《数据结构与算法》课程项目设计报告

1.项目背景介绍与分工说明

1.1 项目背景介绍

随着电子技术的发展，电路设计和分析在现代工程领域中越来越重要。电路的复杂性不断增加，特别是在大规模集成电路和复杂电路网络中，如何快速准确地分析电路的拓扑结构和电气特性成为关键问题。在这些复杂电路中，确定节点之间的最短路径对于优化电路性能、降低能耗以及故障诊断具有重要意义。

本项目旨在开发一个电路最短路径分析系统，通过图结构表示电路的拓扑关系，使用高效的算法计算节点间的最短路径，并通过可视化界面直观展示结果。该系统将为电路设计人员提供一个便捷的工具，帮助他们快速分析和优化电路设计。

本项目已在 GitHub 上开源，访问

https://github.com/mcleoregulus/Project-DataStructuresAndAlgorithms.git 以获取详细信息。

1.2 项目分工说明

徐锦轩（20211069）（队长）：动态图结构和弗洛伊德算法
梅幸元（20211079）（组员）：导入导出配置文件
刘兴焘（20211081）（组员）：电路仿真解算
高语洁（20211080）（组员）：可视化交互界面

2.需求分析

2.1 软件功能整体介绍

本软件旨在实现一个电路仿真与分析系统，用户可以通过图形化界面进行电路的构建、编辑、仿真和分析。系统支持从 CSV 文件导入电路配置信息，导出当前电路状态为 CSV 文件，执行电路仿真以分析各节点电压和支路电流及功率分布，并通过可视化界面展示电路拓扑结构和仿真结果。此外，系统还集成了弗洛伊德算法以计算电路中各节点间的最短路径，为电路设计和优化提供辅助支持。

2.2 软件功能分模块介绍

2.2.1 从 csv 或矩阵导入成图，邻接表导出 csv 电路配置信息

该模块负责电路配置数据的导入和导出功能。用户可以通过指定的 CSV 文件路径，将存储电路拓扑结构和支路参数的配置信息导入到系统中，构建图结构以表示电路。系统支持两种导入方式：直接从 CSV 文件导入或通过矩阵形式导入。导入的电路配置信息包括节点间的连接关系、支路的电阻和电抗值等。

相应的，系统也提供了将当前电路的邻接表导出为 CSV 文件的功能。导出的 CSV 文件包含电路的完整配置信息，方便用户在其他场景下使用或进行备份。导出功能确保了电路数据的可移植性和可维护性。

2.2.2 电路仿真，分析各节点电压值，各支路电流值功率分布

电路仿真模块是系统的核心功能之一。用户可以在构建或导入电路后，执行仿真操作以分析电路的电气特性。仿真过程中，系统将根据电路的拓扑结构和支路参数，计算各节点的电压值、各支路的电流值以及功率分布情况。仿真结果可以直观地展示在用户界面上，用户可以通过节点和支路的标注查看详细的电压和电流等信息，有利于理解电路的运行状态、评估电路性能。

2.2.3 分析各节点和支路的功率分布

在电路仿真器中，功率分布分析是通过`calculatePowerDistribution`方法实现的。该方法基于已计算得到的节点电压和支路电流，通过复数运算计算每条支路的功率。具体来说，对于每条支路，首先获取其电流值，然后计算节点间的电压差，最后通过电压与电流共轭复数的乘积得到复功率。这个复功率的实部表示有功功率，虚部表示无功功率。为了生成新的有向图来表达功率分布，可以创建一个新的图结构，其中节点保持不变，而支路的方向和权重则根据功率的正负来确定：当功率为正时，表示功率从第一个节点流向第二个节点；当功率为负时，则相反。这样，新的有向图就能直观地展示电路中各支路的功率流向和大小，为电路分析和优化提供重要参考。

2.2.4 qt 可视化，交互式生成电路图，信息显示

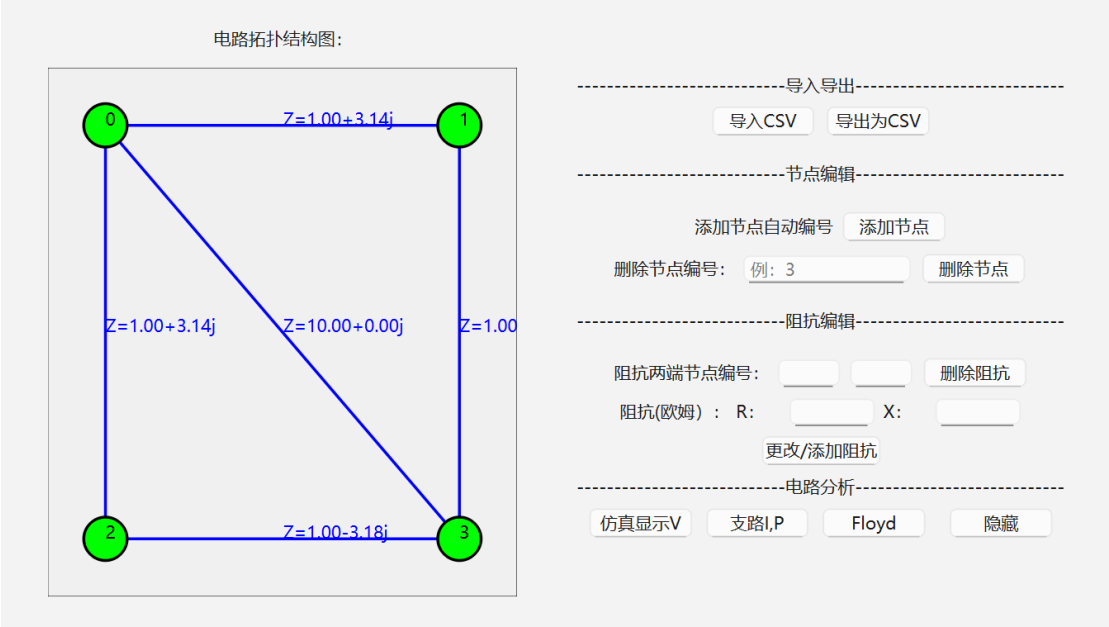


图 1：项目 Qt 交互界面试例

基本交互功能以及控件介绍：
基本功能：可以允许用户通过输入，点击按键等方式执行增加、删除节点操作，增加、删除、修改阻抗操作，导入导出电路拓扑图操作，进行并显示电路仿真计算结果，进行 Floyd 计算等等，并在右侧框图中显示当前的电路拓扑结构以及电路分析结果。

控件介绍：
1. 导入 CSV：按下后程序可以自动导入成员之前编辑好的 config.csv 对应的相对复杂（16 个节点，包含多条相互连接）的连接线路，节省手动添加时间。
2. 导出为 CSV：可以将当前框图中显示的电路导出为 CSV 配置模式并保存，可以在代码中改变打开的 csv 名称再次加载到程序中。
3. 添加节点：点击即可为程序添加一个节点，同时电路拓扑图也会进行相应更新
4. 删除节点编号：选择需要删除的节点编号，注意需要是程序中现存的节点，不然删除无效，输入框返回提示“节点不存在”。如果输入为空，默认删除最后一个节点
5. 删除节点：点击则根据之前输入框里的信息进行节点删除操作，同时展示框电路拓扑结构更新
6. 阻抗两端节点编号：输入想要操作的阻抗两端的节点编号，注意需要小的节点编号在前，大的在后，否则后续操作无效，输入框返回提示“小节点前”。
7. 删除阻抗：根据提供的阻抗编号，验证其是否存在，存在则删除阻抗，不存在则操作无效，输入框返回提示“错误”。
8. R：输入电阻值，X：输入电抗值（单位：欧姆）
9. 更改/添加阻抗：点击则可以读取当前需要修改或添加的阻抗编号，以及用户输入的阻抗值来更改（针对已经存在的阻抗）或添加（针对还不存在的阻抗）阻抗，需要的注意的是若是阻抗两端节点里有当前不存在的节点编号，如此时输入了 6，但只有 4 个节点，程序会自动创建新节点，直到有编号为 6 的新节点存在。
10. 仿真显示 V：点击则会开始进行电路程序仿真计算，同时在节点上方显示当下计算得到的电流（仿真设置为 220V 交流电压源，正极接于 0 节点上，负极则接在当前最后一个节点上，故而 0 节点与最后一个节点悬空可能会导致各节点仿真电压均为 0 或 220，电流与电功率为 0）。点击上文中任意编辑节点与阻抗的按钮都会清空节点电压显示结果
11. 支路 I,P：点击则会在电路拓扑图中显示电路仿真计算得到的各支路电流，损耗功率结果，同样，点击其它任意节点与阻抗按键会清空电流与损耗功率显示结果（若在此之前从来没有点击过仿真显示 V 按钮，仿真未开始，无法显示电流与损耗功率结果）
12. Floyd：点击则会调用编写的 Floyd 函数，在程序输出中（不展示在 Qt 用户交互界面上）输出对应的最短路径计算结果
13. 隐藏：点击可以隐藏当前的电压，电流，损耗功率显示
14. 电路拓扑结构图：绘制当前的电路拓扑图并显示节点编号，进行排列分布，显示节点间阻抗连接情况与阻抗大小（权重），以及在需要时显示进行电路仿真对应的节点电压，中间阻抗上流经的电流大小与损耗的功率大小等

3. 系统设计

3.1 动态图结构实现

3.1.1 Graphl 类

为实现动态图（即顶点数可变的图），将教材上的基于邻接表的静态图的实现作出改进，使用二重指针的单链表表示邻接表，替换原来的单链表数组，从而达到顶点数可变的目的。

LList<LList<Edge> *> *vertex; // 邻接表使用二重指针单链表，构建动态图修改标记数组的实现为单链表，以适配动态的顶点数。

LList<int> *mark; // 动态 mark 数组也需单链表构造函数调用初始化函数，初始化时，可以不指定顶点数，默认创建一个空图。时间复杂度是 O(n)，其中 n 是图的顶点数。

```
void Init(int n = 0)
{
    numVertex = n;
    numEdge = 0;
    mark = new LList<int>(); // Initialize mark array
    for (int i = 0; i < numVertex; i++)
        mark->append(UNVISITED);
    vertex = new LList<LList<Edge> *>();
    for (int i = 0; i < numVertex; i++)
        vertex->append(new LList<Edge>());
}
```

析构函数需要正确释放内存，防止内存泄漏。时间复杂度是 O(n)

```
~GraphI()
{
    delete mark;
    vertex->moveToStart();
    while (vertex->currPos() < vertex->length())
    {
        delete vertex->getValue();
        vertex->next();
    }
    delete vertex;
}
```

由于不再使用数组容器存储各顶点的出边信息，无法再通过索引遍历邻接表，因此需要修改返回第一个邻居和之后的邻居的遍历方法。first 函数的时间复杂度是 O(v)，其中 v 是顶点编号。next 函数的时间复杂度是 O(v+k)，其中 v 是顶点编号，k 是顶点 v 的邻接表长度（即顶点 v 的度数）。（事实上，这里可能造成了重复的光标移动，可以优化）

```
int first(int v)
{ // Return first neighbor of "v"
    vertex->moveToPos(v);
    LList<Edge> *const &currVert = vertex->getValue();
    if (currVert->length() == 0)
        return numVertex;
    // No neighbor
    currVert->moveToStart();
    Edge it = currVert->getValue();
    return it.vertex();
}
int next(int v, int w)
{
    vertex->moveToPos(v);
```

```
        LList<Edge> *const &currVert = vertex->getValue();
        Edge it;
        if (isEdge(v, w))
        {
            if ((currVert->currPos() + 1) < currVert->length())
            {
                currVert->next();
                it = currVert->getValue();
                return it.vertex();
            }
        }
        return n(); // No neighbor
    }
```

由于此程序的应用与电路仿真计算，每条支路的阻抗应为复数，因此使用复数作为图中边的权重。设置边的方法中，增加了一些错误检测以提升程序鲁棒性。同时，如果需要设置的边的顶点编号超过图的顶点数，将调用 addVertex 函数自动增加顶点。还需对邻接表的遍历方式作同步修改。setEdge 函数的时间复杂度是 O(max(i,j)−n+i+k)。

```
    void setEdge(int i, int j, Complex weight)
    {
        if (i == j) {
            cerr << "Error: Invalid Input! setEdge: " << i << " to " << j << endl;
            return;
        }
        if (real(weight) <= 0) {
            cerr << "Error: May not set resistance to negative! setEdge: " << i << " to " << j <<
endl;
            return;
        }

        while (numVertex < i + 1 || numVertex < j + 1) // 自动增加顶点
        {
            addVertex();
        }
        vertex->moveToPos(i);
        LList<Edge> *const &currVert = vertex->getValue();
        Edge currEdge(j, weight);
        if (isEdge(i, j))
        { // Edge already exists in graph
            currVert->remove();
            currVert->insert(currEdge);
        }
        else
        { // Keep neighbors sorted by vertex index
```

```
            numEdge++;
                for (currVert->moveToStart(); currVert->currPos() < currVert->length();
currVert->next())
            {
                Edge temp = currVert->getValue();
                if (temp.vertex() > j)
                    break;
            }
            currVert->insert(currEdge);
        }
    }
```

delEdge 方法同样增加了故障检测，同时更改遍历方式。时间复杂度是 O(i+k)，k 为顶点 i 的度数。

```
    void delEdge(int i, int j) // Delete edge (i, j)
    {
        if (i >= numVertex || j >= numVertex || i < 0 || j < 0 || i == j) {
            cerr << "Error: Invalid Input: Out of Index! delEdge: " << i << " to " << j << endl;
            return;
        }
        vertex->moveToPos(i);
        LList<Edge> *const &currVert = vertex->getValue();
        if (isEdge(i, j))
        {
            currVert->remove();
            numEdge--;
        }
    }
```

isEdge 方法做出类似改变，时间复杂度为 O(i+k)，其中 k 是顶点 i 的邻接表长度。

```
    bool isEdge(int i, int j) // Is (i,j) an edge?
    {
        if (i >= numVertex || j >= numVertex || i < 0 || j < 0 || i == j) {
            cerr << "Error: Invalid Input: Out of Index! isEdge: " << i << " to " << j << endl;
            return false;
        }
        vertex->moveToPos(i);
        LList<Edge> *const &currVert = vertex->getValue();
        Edge it;
        for (currVert->moveToStart();
             currVert->currPos() < currVert->length();
             currVert->next())
        {
            // Check whole list
            Edge temp = currVert->getValue();
```

```
            if (temp.vertex() == j)
                return true;
        }
        return false;
    }
```

weight 方法返回特定边的权重，做出类似优化。时间复杂度为 O(i+k)，其中 k 是顶点 i 的邻接表长度。

```
    Complex weight(int i, int j)
    { // Return weight of (i, j)
        if (i >= numVertex || j >= numVertex || i < 0 || j < 0 || i == j) {
            cerr << "Error: Invalid Input: Out of Index! getWeight: " << i << " to " << j << endl;
            return 0;
        }
        vertex->moveToPos(i);
        LList<Edge> *const &currVert = vertex->getValue();
        Edge curr;
        if (isEdge(i, j))
        {
            curr = currVert->getValue();
            return curr.weight();
        }
        else
            return 0;
    }
```

getMark 和 setMark 方法修改遍历方式和数据更新方式，用于操作标记数组以辅助遍历。getMark 和 setMark 方法的时间复杂度都是 O(v)，其中 v 是顶点编号。

```
    int getMark(int v)
    {
        if (v >= numVertex || v < 0) {
            cerr << "Error: Invalid Input: Out of Index! getMark: " << v << endl;
            return UNVISITED;
        }
        mark->moveToPos(v);
        // mark->print();
        return mark->getValue();
    }
    void setMark(int v, int val)
    {
        if (v >= numVertex || v < 0) {
            cerr << "Error: Invalid Input: Out of Index! setMark: " << v << endl;
            return;
        }
        mark->moveToPos(v);
        mark->remove();
```

```
        mark->insert(val);
    }
```

增加接口用于获取邻接表，用于上层功能实现。

```
    LList<LList<Edge> *> *getAdjList() const { return vertex; }
```

下面是作为动态图新增的方法。addVertex 通过扩展外层单链表的长度以增加指定数量的顶点。时间复杂度为 O(v)，v 为要添加的顶点数，默认为 1。

```
    void addVertex(int v = 1)
    {
        for (int i = 0; i < v; ++i)
        {
            vertex->append(new LList<Edge>());
            mark->append(UNVISITED);
            numVertex++;
        }
    }
```

delVertex 方法用于删除指定顶点，此处分为两种情况，默认删除最后一个顶点，此时直接删除与之有关的所有边，最后删除表示该顶点的单链表。若删除的顶点在中间，还需调整该顶点后续的所有顶点相关的索引，此处时间复杂度较高，为 O(n+m+v)，其中 n 是顶点数，m 是边的数量，v 是要删除的顶点编号。如果要优化算法可以尝试减少不必要的遍历或使用更高效的搜索方法。然而，若使用邻接矩阵实现，时间复杂度达到了 O(n^2)，在顶点数较少且稀疏的电路图中效果不如邻接表实现。

```
    void delVertex(int v = -1)
    {
        v = v == -1 ? numVertex - 1 : v;
        if (v < 0 || v >= numVertex)
        {
            cerr << "Error: delVertex(v) Vertex not found! \n";
            return;
        }

        // 删除与该顶点相关的所有边: 1.遍历所有顶点，删除它们与 v 相连的边
        LList<LList<Edge> *> *const &adjList = getAdjList();
        for (adjList->moveToStart(); adjList->currPos() < adjList->length(); adjList->next())
        {
            if (adjList->currPos() == v)
            {
                continue;
            }

            LList<Edge> *const &currVert = adjList->getValue();
                for (currVert->moveToStart(); currVert->currPos() < currVert->length();
currVert->next())
            {
                Edge temp = currVert->getValue();
```

```
                if (temp.vertex() == v)
                {
                    // 找到与 v 相连的边，删除
                    currVert->remove();
                    numEdge--;
                    break;
                }
            }
        }

        // 2. 删除该顶点的邻接表
        vertex->moveToPos(v);
        numEdge -= vertex->getValue()->length();
        delete vertex->getValue();
        vertex->remove();

        // 3. 删除该顶点的 mark 标记
        mark->moveToPos(v);
        mark->remove();

        // 4. 调整顶点索引：如果删除的不是最后一个顶点，需要调整后续顶点的索引
        if (v != numVertex - 1)
        {
            // 遍历所有顶点的邻接表，调整顶点索引
            for (vertex->moveToStart(); vertex->currPos() < vertex->length(); vertex->next())
            {
                LList<Edge> *const &currVert = adjList->getValue();
                for (currVert->moveToStart(); currVert->currPos() < currVert->length();
currVert->next())
                {
                    Edge temp = currVert->getValue();
                    if (temp.vertex() > v)
                    {
                        // 如果顶点索引大于 v，减 1
                        currVert->remove();
                        currVert->insert(Edge(temp.vertex() - 1, temp.weight()));
                    }
                }
            }
        }

        numVertex--;
    }
```

为实现电路仿真计算的核心功能，现根据邻接表构建导纳矩阵，从而可以在后续通过矩阵求逆和线性运算解出各节点电压和各支路电流。时间复杂度主要由初始化矩阵的 O(n^2) 导致。

```
Complex **getAdmitMatrix()
{
    // 获取邻接表
    LList<LList<Edge> *> *const &adjList = getAdjList();

    // 初始化导纳矩阵
    Complex **admit_matrix = new Complex *[numVertex];
    for (int i = 0; i < numVertex; i++)
    {
        admit_matrix[i] = new Complex[numVertex];
        for (int j = 0; j < numVertex; j++)
        {
            admit_matrix[i][j] = Complex(0, 0); // 初始化为 0
        }
    }

    // 遍历每个顶点
    for (int i = 0; i < numVertex; i++)
    {
        Complex sum = Complex(0, 0); // 对角线元素的和

        // 获取当前顶点的邻接边
        adjList->moveToPos(i);
        LList<Edge> *const &currEdges = adjList->getValue();

        // 遍历邻接边
        currEdges->moveToStart();
        while (currEdges->currPos() < currEdges->length())
        {
            Edge e = currEdges->getValue();
            int j = e.vertex();              // 相连的顶点
            Complex weight = e.weight(); // 边的权重

            // 非对角线元素：负导纳
            admit_matrix[i][j] = Complex(-1, 0) / weight;

            // 累加对角线元素
            sum += Complex(1, 0) / weight;

            currEdges->next();
        }
```

```
            // 对角线元素：总导纳
            admit_matrix[i][i] = sum;
        }
        return admit_matrix;
    }
```

为了方便后续功能实现，增加 getAdjMatrix 方法用于将邻接表转化为邻接矩阵，返回一个二维指针数组，但不存储于图类中，仅在必要时调用。时间复杂度同样主要由初始化矩阵的 O(n^2)导致。

```
    Complex **getAdjMatrix()
    {
        Complex **matrix = new Complex *[numVertex];
        for (int i = 0; i < numVertex; i++)
        {
            matrix[i] = new Complex[numVertex];
            for (int j = 0; j < numVertex; j++)
            {
                matrix[i][j] = Complex(0, 0);
            }
        }

        LList<LList<Edge> *> *const &adjList = getAdjList();
        adjList->moveToStart();
        while (adjList->currPos() < adjList->length())
        {

            LList<Edge> *const &currEdges = adjList->getValue();
            currEdges->moveToStart();
            while (currEdges->currPos() < currEdges->length())
            {
                                    matrix[adjList->currPos()][currEdges->getValue().vertex()]   =
currEdges->getValue().weight();
                currEdges->next();
            }
            adjList->next();
        }
        return matrix;
    }
```

最后是一些辅助功能，例如辅助遍历邻接表、邻接矩阵和导纳矩阵并打印到工作区，以及 setBranch 功能直接设置支路阻抗（无向边）。打印函数的时间复杂度均为 O(n^2)。

3.1.2 Floyd 算法

通过弗洛伊德算法解算多源最短路径问题，并将结果打印到工作区。时间复杂度为

O(n^3)。

```
// 全局变量（或类成员变量），用于存储路径信息
int **path;   // path[i][j] 存储 i 到 j 的最短路径的中间点
double **D;   // D[i][j] 存储 i 到 j 的最短距离

// 递归打印 i 到 j 的最短路径
void prn_pass(int i, int j) {
    if (path[i][j] != -1) {   // 如果有中间点
        prn_pass(i, path[i][j]);   // 递归打印 i 到中间点
        cout << path[i][j] << " -> ";   // 打印中间点
        prn_pass(path[i][j], j);   // 递归打印中间点到 j
    }
}

// 打印所有顶点对的最短路径
void print_allpaths(Graph *G) {
    for (int i = 0; i < G->n(); i++) {
        for (int j = 0; j < G->n(); j++) {
            if (i != j && D[i][j] < INFINITY) {   // 排除自己到自己的情况，并检查可达性
                cout << "Path from " << i << " to " << j << ": ";
                cout << i << " -> ";
                prn_pass(i, j);
                cout << j << " (Impedance: " << D[i][j] << ")" << endl;
            }
        }
    }
}

// Floyd-Warshall 算法实现
void Floyd(Graph *G) {
    int n = G->n();

    // 动态分配 D 和 path 数组
    D = new double*[n];
    path = new int*[n];
    for (int i = 0; i < n; i++) {
        D[i] = new double[n];
        path[i] = new int[n];
    }

    // 初始化 D 和 path
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
```

```
                D[i][j] = 0;    // 自己到自己的距离为 0
            } else {
                Complex w = G->weight(i, j);
                D[i][j] = (w != Complex(0, 0)) ? abs(w) : INFINITY;    // 如果边不存在，设为
INF
            }
            path[i][j] = -1;    // 初始时无中间点
        }
    }

    // Floyd-Warshall 核心算法
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (D[i][k] + D[k][j] < D[i][j]) {
                    D[i][j] = D[i][k] + D[k][j];
                    path[i][j] = k;    // 记录中间点
                }
            }
        }
    }

    // 打印所有最短路径
    print_allpaths(G);

    // 释放动态分配的内存
    for (int i = 0; i < n; i++) {
        delete[] D[i];
        delete[] path[i];
    }
    delete[] D;
    delete[] path;
}
```

首先，初始化距离矩阵 D，设置自顶点到自身的距离为 0，如果有边，则设置权重，否则设置为无穷大（表示不可达）。path 数组初始化为-1，表示没有中间点。

核心算法是使用三重循环进行动态规划更新，检查每一对顶点 i 和 j，通过顶点 k 来更新最短路径。如果通过顶点 k 的路径距离更短，则更新 D[i][j]，并记录中间点 k。通过递归函数 prn_pass 打印从顶点 i 到顶点 j 的具体路径。该函数会递归地打印从 i 到中间点，再从中间点到 j，直到到达源点或目标点。通过动态分配内存存储 D 和 path 数组，并在计算结束后释放内存，避免内存泄漏。

3.1.3 邻接表和邻接矩阵实现的对比

在数据结构中，邻接矩阵和邻接表是两种常见的图表示方法。它们各有优缺点，适用于不同的场景。在电路仿真中，由于电路拓扑结构通常是稀疏的，并且需要动态扩展顶点和边，

邻接表的实现更加高效且适合实际需求。邻接矩阵虽然在稠密图中表现良好，但在稀疏图中会造成空间浪费，且动态扩展复杂度较高，因此使用双重单链表的实现比较合适。

1. 空间复杂度分析

邻接矩阵的空间复杂度为 $O(n^2)$，其中 n 是图的顶点数。这是因为邻接矩阵需要为每个顶点对分配一个存储单元，无论边是否存在。对于稠密图（边数接近 n^2），邻接矩阵的空间利用率较高。然而，对于稀疏图（边数远小于 n^2），邻接矩阵会造成大量的空间浪费，因为大部分存储单元存储的是零值。

邻接表的空间复杂度为 O(n+m)，其中 n 是顶点数，m 是边数。邻接表通过链表存储边信息，每个顶点对应一个链表，链表中存储与该顶点相连的边。对于稀疏图，邻接表的空间利用率更高，因为只存储实际存在的边。在电路仿真中，电路拓扑结构通常是稀疏的，因此邻接表更适合。

2. 时间复杂度分析

邻接矩阵的边查找、插入和删除操作的时间复杂度均为 O(1)，因为可以直接通过矩阵索引访问。然而，遍历所有边的时间复杂度为 $O(n^2)$，因为需要遍历整个矩阵。在实现动态图时，插入或删除顶点等都邻接矩阵需要频繁调整数组的大小，若不借助标准模板库中的 vector 数据容器，实现就比较繁杂，同时导致较高的时间复杂度。

邻接表的边查找、插入和删除操作的时间复杂度为 O(k)，其中 k 是顶点的度数（邻接表长度）。邻接表通过链表实现，动态扩展顶点和边的操作更加高效。对于稀疏图，k 远小于 n，因此邻接表的性能优于邻接矩阵。遍历所有边的时间复杂度为 O(n+m)，因为需要遍历所有顶点和边。

3.2 Utils 工具集合

在电路仿真项目中，为了方便用户从外部数据源导入和导出电路配置信息，我们实现了一系列工具函数。这些工具函数能够将电路配置信息从 CSV 文件导入到图结构中，或者将图结构导出到 CSV 文件。以下是这些工具函数的详细说明。

3.2.1 从 CSV 导入电路配置信息并构建图

为了方便用户从外部数据源导入电路配置信息，我们实现了一个从 CSV 文件导入电路配置信息并构建图的功能。该功能能够读取 CSV 文件中的电路配置信息，并将其转换为图结构中的顶点和边。

功能逻辑实现如下：打开 CSV 文件并读取内容；解析每一行数据，提取顶点对（如 1-2）和对应的电阻（R）与电抗（X）；将顶点对和对应的复数阻抗（R+jX）存储到图中，解析后的数据被存储在一个动态分配的二维数组中，以便后续处理；如果顶点编号超出当前图的顶点范围，则自动扩展图的顶点数。

```
int importFromCSV(const string &filename, Graphl &circuit)
{
    string filepath = "../../data/" + filename + ".csv";
    ifstream file(filepath);
    if (!file.is_open())
    {
        cerr << "Cannot open file: " << filepath << endl;
        return 1;
    }
```

```
string line;
int numBranch = 0;
double **data = new double *[MAX_SIZE];

// 跳过标题行
if (file.good())
{
    getline(file, line); // 读取并丢弃标题行
}

// 按行读取文件
while (getline(file, line) && numBranch < MAX_SIZE)
{
    stringstream ss(line);
    string field;
    string fields[3]; // 只需要存储前三列
    int col = 0;

    // 按逗号分割每行，只读取前三列
    while (getline(ss, field, ',') && col < 3)
    {
        fields[col++] = field;
    }

    // 检查是否至少有三列数据
    if (col < 3)
    {
        continue;
    }

    // 提取第一列中的数字对
    string firstColumn = fields[0];
    size_t dashPos = firstColumn.find('-');

    if (dashPos == string::npos)
    {
        continue;
    }

    string num1Str = firstColumn.substr(0, dashPos);
    string num2Str = firstColumn.substr(dashPos + 1);

    double num1, num2;
    try
```

```
            {
                num1 = stoi(num1Str);
                num2 = stoi(num2Str);
            }
            catch (const exception &e)
            {
                cerr << "Error: Invalid File, cannot convert to number! " << endl;
                continue;
            }

            // 提取第二列和第三列的数字
            double num3, num4;
            try
            {
                num3 = stod(fields[1]);
                num4 = stod(fields[2]);
            }
            catch (const exception &e)
            {
                cerr << "Error: Invalid File, cannot convert to number! " << endl;
                continue;
            }

            // 存储数据
            data[numBranch] = new double[4];
            data[numBranch][0] = num1;
            data[numBranch][1] = num2;
            data[numBranch][2] = num3;
            data[numBranch][3] = num4;
            numBranch++;
        }

        file.close();
        importFromMatrix(data, circuit, numBranch);

        // 释放内存
        for (int i = 0; i < numBranch; i++)
        {
            delete[] data[i];
        }
        delete[] data;

        return 0;
    }
```

3.2.1 将电路图导出到 CSV 文件

为了便于后续分析和验证，我们实现了将电路图的邻接表导出到 CSV 文件的功能。该功能能够将图结构中的顶点和边信息导出到 CSV 文件中，方便用户查看和修改。

功能逻辑实现如下：首先打开指定的 CSV 文件，并写入标题行。获取图的邻接表；遍历邻接表，提取每条边的顶点对和对应的阻抗值；将顶点对和阻抗值写入 CSV 文件，格式与导入时的格式一致。

```
void exportAdjListToCSV(Graphl &circuit, const std::string &filename)
{
    std::string filepath = "../../data/" + filename + ".csv";
    std::ofstream file(filepath);

    if (!file.is_open())
    {
        std::cerr << "Cannot open file for writing: " << filepath << std::endl;
        return;
    }

    file << "Line bus to bus,R (resistance),X (reactance)\n";

    // 获取邻接表，每个顶点对应一个 LList<Edge>*，存储所有邻接边
    LList<LList<Edge> *> *adjList = circuit.getAdjList();

    // 遍历所有顶点
    for (int i = 0; i < circuit.n(); ++i)
    {
        // 移动到相应的顶点 i 位置，获取其邻接链表
        adjList->moveToPos(i);
        LList<Edge> *currEdges = adjList->getValue();

        // 遍历顶点 i 的所有边
        currEdges->moveToStart();
        while (currEdges->currPos() < currEdges->length())
        {
            Edge e = currEdges->getValue();
            int j = e.vertex(); // 边连接到的顶点编号（0-based）

            // 为了防止重复输出无向边，只在 i < j 时输出
            if (i < j && e.weight() != Complex(0, 0))
            {
                file << i + 1 << "-" << j + 1 << ","
                     << std::fixed << std::setprecision(4) << e.weight().real() << ","
                     << e.weight().imag() << "\n";
            }
```

```
                currEdges->next();
            }
        }

        file.close();
        std::cout << "Exported graph to " << filepath << std::endl;
}
```

3.3 电路仿真所需量解算

3.3.1 电路仿真器的基本结构

电路仿真器的基本结构由多个关键成员变量组成，这些变量共同构成了电路仿真的核心数据结构。首先，circuit 作为电路图的引用，存储了电路的拓扑结构和元件参数，是整个仿真过程的基础。节点电压向量 nodeVoltages 使用复数形式存储每个节点的电压值，能够完整表示交流电压的幅值和相位。支路电流向量 branchCurrents 记录了每条支路的电流值，而支路功率向量 branchPowers 则存储了相应的功率信息。此外，通过 voltageSources 和 currentSources 两个映射表，可以灵活地记录和管理电路中的电压源和电流源的位置及其参数值。

```
class CircuitSimulator {
private:
    Graphl &circuit;                                    // 电路图引用
    vector<complex<double>> nodeVoltages;       // 节点电压向量
    vector<complex<double>> branchCurrents;    // 支路电流向量
    vector<complex<double>> branchPowers;       // 支路功率向量
    map<int, complex<double>> voltageSources; // 节点电压源映射
    map<int, complex<double>> currentSources; // 节点电流源映射

public:
    CircuitSimulator(Graphl &cg) : circuit(cg) {
        nodeVoltages.resize(circuit.n(), complex<double>(0, 0));
        initializeVectors();
    }

    void initializeVectors() {
        int n = circuit.n();
        branchCurrents.clear();
        branchPowers.clear();

        // 为每个可能的支路预分配空间
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (circuit.getAdjMatrix()[i][j] != complex<double>(0, 0)) {
                    branchCurrents.push_back(complex<double>(0, 0));
                    branchPowers.push_back(complex<double>(0, 0));
```

```
                }
            }
        }
    }
```

构造函数接收一个电路图引用作为参数，并立即进行必要的初始化工作。首先，根据电路中的节点数量为电压向量分配适当的空间，并将所有初始值设置为零。然后，通过调用initializeVectors()方法，完成对其他向量的初始化工作，确保所有数据结构都处于正确的初始状态，为后续的电路分析做好准备。

3.3.2 核心功能实现

节点电压求解是整个电路分析的核心环节。这个过程首先需要构建导纳矩阵，这是一个n×n 的复数矩阵，其中 n 代表电路中的节点数量。导纳矩阵的对角线元素表示连接到该节点的所有支路导纳之和，而非对角线元素则是对应支路导纳的负值。在构建导纳矩阵的同时，还需要处理电流源和电压源的影响。电流源直接添加到对应节点的电流向量中，而电压源则需要通过修改导纳矩阵的相应行来处理。最后，使用 QR 分解方法求解线性方程组，得到所有节点的电压值，这些值被存储在 nodeVoltages 向量中，为后续的支路电流和功率计算提供基础数据。

```
void solveNodeVoltages() {
    int n = circuit.n();
    MatrixXcd Y = MatrixXcd::Zero(n, n);   // 导纳矩阵
    VectorXcd I = VectorXcd::Zero(n);        // 电流源向量

    // 构建导纳矩阵
    auto admitMatrix = circuit.getAdmitMatrix();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            Y(i, j) = admitMatrix[i][j];
        }
    }

    // 添加电流源
    for (const auto &[node, current] : currentSources) {
        I(node) = current;
    }

    // 处理电压源约束
    for (const auto &[node, voltage] : voltageSources) {
        Y.row(node).setZero();
        Y(node, node) = 1.0;
        I(node) = voltage;
    }

    // 求解方程 Y * V = I
    VectorXcd V = Y.colPivHouseholderQr().solve(I);
```

```
    // 保存结果
    for (int i = 0; i < n; i++) {
        nodeVoltages[i] = V(i);
    }
}
```

支路电流的计算过程基于已经求解得到的节点电压。通过遍历电路中的所有支路，对每一条支路应用欧姆定律，计算其电流值。计算过程中需要考虑支路的阻抗特性，使用复数运算来处理交流电路中的相位关系。为了确保计算的准确性，代码采用了上三角矩阵的遍历方式，避免了对同一条支路的重复计算。计算得到的支路电流值被有序地存储在 branchCurrents 向量中，这些数据不仅反映了电流的大小，还包含了相位信息，为后续的功率计算提供了必要的基础。

```
void calculateBranchCurrents() {
    int n = circuit.n();
    int branchIndex = 0;

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (circuit.getAdjMatrix()[i][j] != complex<double>(0, 0)) {
                complex<double> impedance = circuit.getAdjMatrix()[i][j];
                complex<double> current = (nodeVoltages[i] - nodeVoltages[j]) / impedance;
                branchCurrents[branchIndex++] = current;
            }
        }
    }
}
```

功率分布的计算是电路分析的最后一个重要环节。这个过程基于已经计算得到的节点电压和支路电流，通过复数运算来计算每条支路的功率。在计算过程中，使用电压和电流的共轭复数相乘，得到复功率值。这个复功率的实部代表有功功率，虚部代表无功功率，而模值则代表视在功率。通过这种方式，可以全面了解电路中各支路的功率分布情况，包括有功功率的消耗和无功功率的交换，这对于分析电路的效率和稳定性具有重要意义。

```
void calculatePowerDistribution() {
    int n = circuit.n();
    int branchIndex = 0;

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (circuit.getAdjMatrix()[i][j] != complex<double>(0, 0)) {
                complex<double> current = branchCurrents[branchIndex];
                complex<double> voltage = nodeVoltages[i] - nodeVoltages[j];
                branchPowers[branchIndex] = voltage * conj(current);
                branchIndex++;
            }
```

```
            }
        }
}
```

3.3.3 辅助功能实现

电源设置功能为用户提供了灵活配置电路激励的方式。通过 setVoltageSource 和 setCurrentSource 方法，用户可以方便地设置任意节点的电压源或电流源，支持复数形式的参数设置，能够准确表示交流电源的特性。参考节点的设置通过 setReferenceNode 方法实现，默认选择电路中的最后一个节点作为参考点，并将其电压设置为零，这符合电路分析中的常见做法，为整个电路提供了电压参考基准。

```
void setVoltageSource(int node, complex<double> voltage) {
    voltageSources[node] = voltage;
}

void setCurrentSource(int node, complex<double> current) {
    currentSources[node] = current;
}

void setReferenceNode(int n = -1) {
    n = n == -1 ? circuit.n() - 1 : n;
    setVoltageSource(n, {0, 0});
}
```

结果输出功能为用户提供了直观的电路分析结果展示。通过 printNodeVoltages、printBranchCurrents 和 printBranchPowers 等方法，可以清晰地显示节点电压、支路电流和功率分布的计算结果。输出格式经过精心设计，使用极坐标形式表示复数，保留适当的小数位数，并包含必要的单位信息。这些输出不仅便于用户理解电路的工作状态，也为进一步的电路分析和优化提供了数据支持。

```
void printNodeVoltages() {
    cout << "Node Voltage: " << endl;
    const auto &voltages = getNodeVoltages();
    for (int i = 0; i < voltages.size(); i++) {
        double magnitude = abs(voltages[i]);
        double phase = arg(voltages[i]) * 180 / M_PI;
        cout << fixed << setprecision(4);
        cout << "V" << (i + 1) << " = ";
        cout << magnitude << " |_" << phase << " V" << endl;
    }
}

void printBranchCurrents() {
    cout << "\nBranch current: " << endl;
    const auto &currents = getBranchCurrents();
    int branchIndex = 0;
```

```
    for (int i = 0; i < circuit.n(); i++) {
        for (int j = i + 1; j < circuit.n(); j++) {
            if (circuit.getAdjMatrix()[i][j] != Complex(0, 0)) {
                double magnitude = abs(currents[branchIndex]);
                double phase = arg(currents[branchIndex]) * 180 / M_PI;
                cout << "I" << (i + 1) << "-" << (j + 1) << " = ";
                cout << magnitude << " |_" << phase << " A" << endl;
                branchIndex++;
            }
        }
    }
}

void printBranchPowers() {
    cout << "\nBranch power" << endl;
    const auto &powers = getBranchPowers();
    int branchIndex = 0;
    for (int i = 0; i < circuit.n(); i++) {
        for (int j = i + 1; j < circuit.n(); j++) {
            if (circuit.getAdjMatrix()[i][j] != Complex(0, 0)) {
                double active = real(powers[branchIndex]);
                double reactive = imag(powers[branchIndex]);
                cout << "S" << (i + 1) << "-" << (j + 1) << " = ";
                cout << active << (reactive >= 0 ? "+" : "") << reactive << "j";
                cout << " VA" << endl;
                branchIndex++;
            }
        }
    }
}
```

3.3.4 使用示例

电路仿真器的使用过程如下：首先需要创建电路图对象，设置适当的节点数量。然后，通过 setEdge 方法设置各支路的阻抗参数，支持电阻、电感和电容等不同类型的元件。创建仿真器实例后，可以方便地设置电压源、电流源和参考节点。最后，调用 analyze 方法执行完整的电路分析，并通过相应的打印方法查看分析结果。这个示例展示了如何使用仿真器进行基本的电路分析，包括设置电路参数、执行分析和查看结果等完整流程。

```
// 创建电路图
Graphl circuit(4);   // 4 个节点

// 设置支路阻抗
circuit.setEdge(0, 1, Complex(1, 0));     // 1Ω 电阻
circuit.setEdge(1, 2, Complex(0, 1));     // 1H 电感
circuit.setEdge(2, 3, Complex(0, -1));    // 1F 电容
```

```
// 创建仿真器
CircuitSimulator simulator(circuit);

// 设置电源
simulator.setVoltageSource(0, Complex(10, 0));   // 10V 电压源
simulator.setCurrentSource(1, Complex(0, 1));    // 1A 电流源
simulator.setReferenceNode(3);   // 设置参考节点

// 执行分析
simulator.analyze();

// 输出结果
simulator.printNodeVoltages();
simulator.printBranchCurrents();
simulator.printBranchPowers();
```

这个电路仿真器实现了完整的电路分析功能，不仅支持基本的直流电路分析，还能够处理复杂的交流电路。它提供了节点电压法求解、支路电流计算和功率分布分析等核心功能，并通过友好的接口支持电源设置和结果输出。

3.4 Qt 可视化相关代码设计

下文展示主要控件的代码设计，对于部分控件（如 Floyd，隐藏等）只是简单的操作或是调用函数，不再详细介绍

3.4.1 Qt 的显示结构（.ui 文件）设计

在编辑 Qt 的 ui 文件界面进行界面绘制时的控件放置如下：

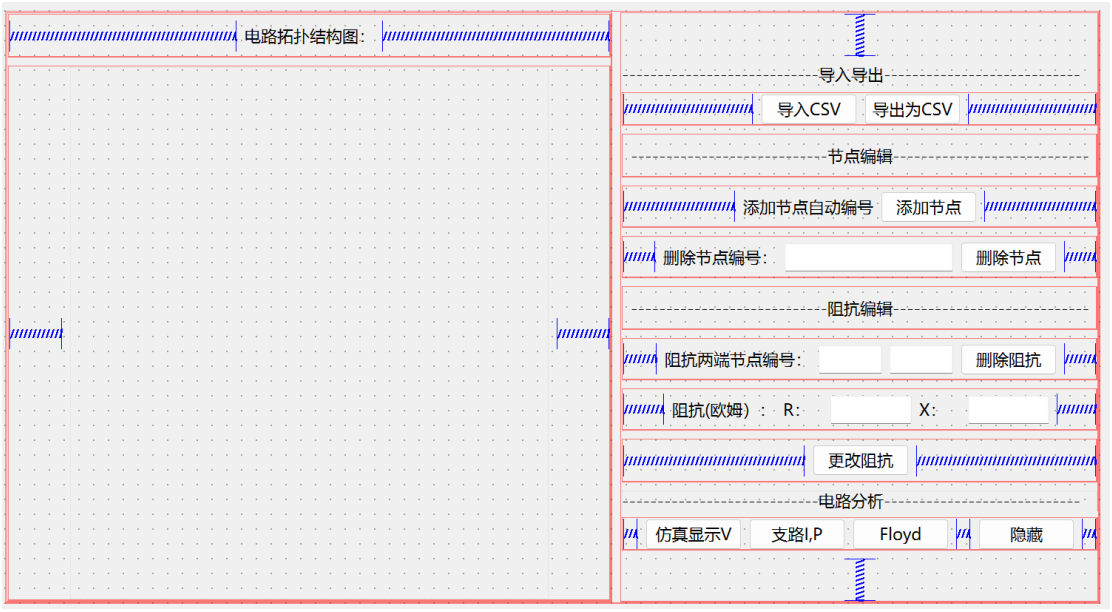


图 2：ui 文件进行界面编辑

设计思路是将主要的展示窗口放置在右侧，用于响应对应的操作，同时将主要的用户操

作空间放在左侧，按导入导出，节点编辑，阻抗编辑，电路分析 4 个模块进行分类，加上了标签以及分隔线方便展示与用户理解，还积极利用 layout 以及伸缩弹簧控件辅助了布局，使整体展示得到优化。

3.4.2 widget 窗口初始化

主要功能：

输入控制：

1. 对节点编号输入框（如删除节点、阻抗端点）限制为整数
2. 对阻抗值输入框（电阻/电抗）允许小数和负数
3. 使用 QRegularExpressionValidator 实现实时输入验证对应

对应代码：

```
//限制输入的数据类型
    QRegularExpression rx("-?\\d{1,3}");   //节点编号仅为整数
    QValidator *validator = new QRegularExpressionValidator(rx, this);
    ui->Delete_txt->setValidator(validator);
    ui->Bus_txt1->setValidator(validator);
    ui->Bus_txt2->setValidator(validator);

    QRegularExpression rx1("^(-?\\d+)(\\.\\d+)?$");   //阻抗可以为负数或小数
    QValidator *validator1 = new QRegularExpressionValidator(rx1, this);
    ui->Resistance_txt->setValidator(validator1);
    ui->Reactance_txt->setValidator(validator1);
```

数据初始化：

1. 自动从"example2.csv"加载初始电路数据
2. 控制台打印邻接表用于调试
3. 立即刷新图形显示组件

对应代码：

```
//初始化矩阵
    importFromCSV(my_circuit,"example2");
    cout << "print Adjust" << endl;
    my_circuit.printAdjList();
    ui->graphWidget->update();
```

用户引导：

1. 在删除节点输入框设置示例提示（"例：3"）
2. 通过输入限制防止无效操作

对应代码：

```
ui->Delete_txt->setPlaceholderText(tr("例：3"));   //设置输入框提示语
```

3.4.3 添加节点按钮

主要功能：

调用 my_circuit.addVertex()添加新节点，并用控制台打印添加成功信息及更新后的邻接表，刷新图形界面显示新节点

代码实现：

```
void Widget::on_Add_Button_clicked()   //当点击了添加节点按钮
{
    //尝试调用函数
    my_circuit.addVertex();
    cout << "Bus added! Print AdjustList" << endl;
    my_circuit.printAdjList();
    ui->graphWidget->showVoltageResults(false);
    ui->graphWidget->showCurrentPowerResults(false);
    ui->graphWidget->update();
}
```

3.4.4 删除节点按钮

主要功能：

1. 获取输入框中的节点编号（转换为整数类型）
2. 调用 delVertex()尝试删除节点：
3. 成功：恢复默认提示文本
4. 失败：显示"节点不存在"错误提示
5. 控制台打印操作结果和更新后的邻接表
6. 清空输入框并隐藏分析结果
7. 刷新电路图显示

代码实现：

```
void Widget::on_Delete_Button_clicked()   //当点击了删除节点按钮
{
    //尝试获取 txt 中的信息
    //QString Deletebus = ui->Delete_txt->text();   //得到输入内容
    int Deletebus = ui->Delete_txt->text().toInt();

    if (my_circuit.delVertex(Deletebus))   {
        ui->Delete_txt->setPlaceholderText(tr("例：3"));   //设置输入框提示语
        qDebug() << Deletebus << " " << Qt::endl;
    }
    else{
        ui->Delete_txt->setPlaceholderText(tr("节点不存在"));   //提示错误
        qDebug() << "节点不存在，重新输入" << " " << Qt::endl;
    }

    cout << "Bus delete! Print AdjustList" << endl;
    my_circuit.printAdjList();

    ui->Delete_txt->clear();   //清空输入框
    ui->graphWidget->showVoltageResults(false);
    ui->graphWidget->showCurrentPowerResults(false);
    ui->graphWidget->update();
```

}

3.4.5 删除阻抗按钮

核心功能：

1. 处理用户点击"删除阻抗"按钮的操作，主要完成以下工作：
2. 获取用户输入的两个节点编号
3. 验证并删除这两个节点之间的阻抗连接
4. 提供操作反馈和界面更新

执行流程：

1. 输入处理阶段
    a. 读取两个输入框的节点编号（bus1 和 bus2）
    b. 验证删除条件
    c. 检查节点顺序（要求 bus2 > bus1）
    d. 检查阻抗是否存在（isEdge 判断）
2. 分支处理
    a. 满足条件时：执行删除操作（delEdge）
    b. 不满足时：显示对应的错误提示
3. 后续处理
    a. 清空输入框内容
    b. 打印操作日志
    c. 更新电路图显示
    d. 隐藏分析结果

用户反馈机制：

1. 通过 placeholderText 提供操作指引和错误提示
2. 控制台输出操作结果和当前邻接表状态
3. 即时更新图形界面反映最新电路状态

代码实现：

```
// 删除阻抗
void Widget::on_Delete_Button_clicked()   //当点击了删除节点按钮
{
    //尝试获取 txt 中的信息
    //QString Deletebus = ui->Delete_txt->text();   //得到输入内容
    int Deletebus = ui->Delete_txt->text().toInt();
    if (ui->Delete_txt->text().isEmpty()) {
        // 输入框为空
        int Deletebus = -1; //默认传参为 1，删除最后一个节点
    }

    if (my_circuit.delVertex(Deletebus))   {
        ui->Delete_txt->setPlaceholderText(tr("例：3"));   //设置输入框提示语
        qDebug() << Deletebus << " " << Qt::endl;
    }
    else{
        ui->Delete_txt->setPlaceholderText(tr("节点不存在"));   //提示错误
```

```
            qDebug() << "节点不存在，重新输入" << " " << Qt::endl;
        }

        cout << "Bus delete! Print AdjustList" << endl;
        my_circuit.printAdjList();

        ui->Delete_txt->clear();   //清空输入框
        ui->graphWidget->showVoltageResults(false);
        ui->graphWidget->showCurrentPowerResults(false);
        ui->graphWidget->update();
}
```

3.4.6 更改阻抗按钮

主要功能：

1. 修改或添加两节点间的阻抗参数
2. 验证节点顺序和电阻值的有效性
3. 提供操作反馈和界面更新

执行流程：

1. 获取用户输入的节点编号和阻抗值（电阻+电抗）
2. 组合成复数形式的阻抗值
3. 验证节点顺序（要求 bus2 > bus1）和电阻非负
4. 通过验证后调用 setEdge 设置双向阻抗
5. 根据操作结果显示相应提示信息
6. 清空所有输入框并刷新界面

用户反馈机制：

- 通过 placeholderText 提示正确输入格式
- 针对不同错误类型显示特定提示：
- 节点顺序错误："小节点前"
- 电阻值为负："电阻>0"
- 控制台输出阻抗修改日志
- 即时更新电路图显示

```
void Widget::on_Change_impedance_clicked()
{
        int bus1 = ui->Bus_txt1->text().toInt();
        int bus2 = ui->Bus_txt2->text().toInt();
        float resistance = ui->Resistance_txt->text().toFloat();
        float racetance = ui->Reactance_txt->text().toFloat();
        complex<double> mycomplex(resistance, racetance);

        if(bus2>bus1){
            if(resistance>=0){
                my_circuit.setEdge(bus1,bus2, mycomplex);
                my_circuit.setEdge(bus2,bus1, mycomplex);
                ui->Bus_txt1->setPlaceholderText(tr("3"));   //设置输入框提示语
```

```
            ui->Bus_txt2->setPlaceholderText(tr("4"));   //设置输入框提示语
        }
        else{
            ui->Resistance_txt->setPlaceholderText(tr("电阻>0"));
        }
    }
    else{
        ui->Bus_txt1->setPlaceholderText(tr("小节点前"));   //设置输入框提示语
        ui->Bus_txt2->setPlaceholderText(tr("小节点前"));   //设置输入框提示语
    }

    ui->Bus_txt1->clear();   //清空输入框
    ui->Bus_txt2->clear();   //清空输入框
    ui->Reactance_txt->clear();   //清空输入框
    ui->Resistance_txt->clear();

    cout << "Impedance change! Print AdjustList" << endl;
    my_circuit.printAdjList();
    ui->graphWidget->showVoltageResults(false);
    ui->graphWidget->showCurrentPowerResults(false);
    ui->graphWidget->update();
}
```

3.4.7 电路分析按钮

主要功能：

1. 执行完整的电路分析计算
2. 计算节点电压、支路电流和功率分布
3. 在界面显示电压分析结果
4. 输出详细分析数据到控制台

执行流程：

1. 创建 CircuitSimulator 实例并设置参考节点
2. 在节点 0 设置 220V 电压源
3. 执行 analyze()进行电路分析
4. 将分析结果传递给图形界面
5. 控制台输出详细的电压、电流和功率数据
6. 图形界面显示电压分布

分析数据输出：

- 节点电压：幅值和相位角
- 支路电流：幅值和相位角
- 支路功率：有功和无功分量

代码实现：

```
void Widget::on_analyseButton_clicked()
{
```

```
CircuitSimulator simulator(my_circuit);
simulator.setVoltageSource(0, Complex(220, 0));
simulator.setReferenceNode();
simulator.analyze();

ui->graphWidget->setAnalysisData(
    simulator.getNodeVoltages(),
    simulator.getBranchCurrents(),
    simulator.getBranchPowers()
    );
ui->graphWidget->showVoltageResults(true);          // 只显示电压
ui->graphWidget->showCurrentPowerResults(false); // 不显示电流和功率

cout << "Circuit analysing! Print parameters:" << endl;

// 看能否打印电流电压矩阵
// 获取节点电压
cout << "Node Voltage: " << endl;
const auto &voltages = simulator.getNodeVoltages();
for (int i = 0; i < voltages.size(); i++)
{
    double magnitude = abs(voltages[i]);              // 模
    double phase = arg(voltages[i]) * 180 / M_PI; // 幅角，转换为度
    cout << fixed << setprecision(4);
    cout << "V" << (i + 1) << " = ";
    // api dosomthing()
    cout << magnitude << " |_" << phase << " V" << endl;
}

// 节点连线上的电流
cout << "\nBranch current: " << endl;
const auto &currents = simulator.getBranchCurrents();
int branchIndex = 0;
for (int i = 0; i < my_circuit.n(); i++)
{
    for (int j = i + 1; j < my_circuit.n(); j++)
    {
        if (my_circuit.getAdjMatrix()[i][j] != Complex(0, 0))
        {
            double magnitude = abs(currents[branchIndex]);              // 模
            double phase = arg(currents[branchIndex]) * 180 / M_PI; // 幅角
            cout << "I" << (i + 1) << "-" << (j + 1) << " = ";
            // api dosomthing()
            cout << magnitude << " |_" << phase << " A" << endl;
```

```
                branchIndex++;
            }
        }
    }

    // 节点连线上的损耗功率
    cout << "\nBranch power" << endl;
    const auto &powers = simulator.getBranchPowers();
    branchIndex = 0;
    for (int i = 0; i < my_circuit.n(); i++)
    {
        for (int j = i + 1; j < my_circuit.n(); j++)
        {
            if (my_circuit.getAdjMatrix()[i][j] != Complex(0, 0))
            {
                double active = real(powers[branchIndex]);     // 有功
                double reactive = imag(powers[branchIndex]); // 无功
                // api dosomthing()
                cout << "S" << (i + 1) << "-" << (j + 1) << " = ";
                // printComplex(powers[branchIndex]);
                cout << active << (reactive >= 0 ? "+" : "") << reactive << "j";
                cout << " VA" << endl;
                branchIndex++;
            }
        }
    }
}
```

3.4.8 paintwidget 绘图组件

功能概述：

paintwidget 类负责电路拓扑结构的可视化展示，主要功能包括：

1. 绘制电路节点和阻抗连接线
2. 显示电压、电流、功率等分析结果
3. 支持动态更新电路图形

核心实现：

电路绘制功能：

```
void paintwidget::drawAdjacencyMatrix(QPainter &painter) {
    // 1. 计算节点分布位置
    for(int i=0; i<nodeCount; ++i){
        nodePositions.push_back(calculateNodePosition(i, nodeCount));
    }

    // 2. 绘制连接线（蓝色）和阻抗值
    painter.setPen(Qt::blue);
```

```
        painter.drawLine(nodePositions[i], nodePositions[j]);
        painter.drawText(midPoint, "Z="+formatComplex(e.weight()));

        // 3. 绘制节点（绿色圆形）
        painter.setBrush(Qt::green);
        painter.drawEllipse(nodePositions[i], 15, 15);
}
```

分析结果显示：

```
// 显示节点电压（红色）
if(showVoltage){
        painter.drawText(voltagePos, "V="+formatComplex(nodeVoltages[i]));
}

// 显示支路电流和功率（红色）
if(showCurrentPower){
        painter.drawText(currentPos, "I="+formatComplex(branchCurrents[branchIndex]));
        painter.drawText(powerPos, "P="+formatComplex(branchPowers[branchIndex]));
}
```

数据更新接口：

```
void paintwidget::setAnalysisData(
        const vector<complex<double>>& voltages,
        const vector<complex<double>>& currents,
        const vector<complex<double>>& powers)
{
        nodeVoltages = voltages;
        branchCurrents = currents;
        branchPowers = powers;
        update(); // 触发重绘
}
```

设计特点

1. 采用网格布局自动计算节点位置
2. 不同数据用颜色区分（蓝色-阻抗，红色-分析结果）
3. 复数数据格式化显示（如"3.00+4.00j"）
4. 支持动态显示/隐藏分析结果

主要功能：

1. 可视化电路拓扑结构
2. 动态显示节点和阻抗连接
3. 支持分析结果的可视化展示（电压/电流/功率）
4. 提供自动布局算法

核心特性：

- 智能节点布局：根据节点数量自动计算位置
- 多图层显示：
- 基础层：节点和阻抗连接
- 分析层：电压/电流/功率数据

● 交互控制：通过标志位控制显示内容

关键实现逻辑：

1. 节点布局算法
    a. 采用网格化布局策略
    b. 根据节点总数计算行列数
    c. 均匀分布节点位置
2. 图形绘制流程
    a. 绘制背景和网格
    b. 绘制阻抗连接线（含阻抗值标注）
    c. 绘制电路节点（圆形+编号）
    d. 叠加分析结果数据
3. 分析结果显示控制
    a. 电压显示：节点上方
    b. 电流/功率显示：连线中间
    c. 通过 showVoltageResults/showCurrentPowerResults 控制可见性

代码实现（核心部分）：

```
// 计算节点位置（自动布局）
QPoint paintwidget::calculateNodePosition(int nodeIndex, int totalNodes) {
    int n = std::ceil(std::sqrt(totalNodes));
    int row = nodeIndex / n;
    int col = nodeIndex % n;
    int margin = 40;
    int x = margin + col * (width() - 2 * margin) / (n - 1);
    int y = margin + row * (height() - 2 * margin) / (n - 1);
    return QPoint(x, y);
}

// 主绘制函数
void paintwidget::drawAdjacencyMatrix(QPainter &painter) {
    // 1. 计算所有节点位置
    for(int i = 0; i < nodeCount; ++i) {
nodePositions.push_back(calculateNodePosition(i, nodeCount));
    }

    // 2. 绘制连接线
painter.setPen(QPen(Qt::blue, 2));
    for(所有有效连接){
painter.drawLine(nodePositions[i], nodePositions[j]);
// 绘制阻抗值
painter.drawText(midPoint, "Z="+formatComplex(impedance));
if(showCurrentPower){
// 绘制电流和功率（红色）
painter.setPen(QPen(Qt::red));
painter.drawText(currentPos, "I="+formatComplex(current)));
```

```
painter.drawText(powerPos, "P="+formatComplex(power)));
}
    }

    // 3. 绘制节点
painter.setPen(QPen(Qt::black, 2));
painter.setBrush(QBrush(Qt::green));
    for(所有节点){
painter.drawEllipse(nodePositions[i], 15, 15);
painter.drawText(nodePositions[i], QString::number(i));
if(showVoltage){
// 显示节点电压
painter.drawText(voltagePos, "V="+formatComplex(voltage)));
}
    }
}

// 复数格式化显示
QString paintwidget::formatComplex(const std::complex<double>& c) {
std::ostringstream oss;
oss << std::fixed << std::setprecision(2)
<< c.real() << (c.imag() >= 0 ? "+" : "") << c.imag() << "j";
    return QString::fromStdString(oss.str());
}
```

优化特性：

1. 抗锯齿渲染保证显示质量
2. 动态间距适应不同窗口尺寸
3. 颜色区分不同信息类型（蓝色-阻抗，红色-分析结果）
4. 格式化显示复数数据（保留 2 位小数）

交互控制接口：

```
void setAnalysisData(...);   // 设置分析数据
void showVoltageResults(bool);   // 控制电压显示
void showCurrentPowerResults(bool); // 控制电流/功率显示
```

该组件通过合理的分层设计和标志位控制，实现了电路拓扑和多种分析结果的高效可视化，同时保持代码结构清晰可维护。

4. 运行效果

4.1 VScode 运行试例

并不作为主要的运行方式推荐，但作为 cpp 交互程序，也展示了项目代码的基本功能。主要的运行文件为项目 src/main/main.cpp

初始显示：

```
当前电路邻接表：
Vertex 0 -> (1, (1,3.14159)) (2, (1,3.14159)) (3, (10,0))
Vertex 1 -> (0, (1,3.14159)) (3, (1,3.14159))
Vertex 2 -> (0, (1,3.14159)) (3, (1,3.18))
Vertex 3 -> (0, (10,0)) (1, (1,3.14159)) (2, (1,3.18))

===== 电路分析系统 =====
1. 添加节点
2. 删除节点
3. 删除阻抗
4. 设置/修改阻抗
5. 执行电路分析
6. 最短路径分析(Floyd)
0. 退出系统
请选择操作(0-6):
```

选择 1：添加节点

输入输出；

```
===== 电路分析系统 =====
1. 添加节点
2. 删除节点
3. 删除阻抗
4. 设置/修改阻抗
5. 执行电路分析
6. 最短路径分析(Floyd)
0. 退出系统
请选择操作(0-6): 1
节点添加成功！当前邻接表：
Vertex 0 -> (1, (1,3.14159)) (2, (1,3.14159)) (3, (10,0))
Vertex 1 -> (0, (1,3.14159)) (3, (1,3.14159))
Vertex 2 -> (0, (1,3.14159)) (3, (1,3.18))
Vertex 3 -> (0, (10,0)) (1, (1,3.14159)) (2, (1,3.18))
Vertex 4 ->

当前电路邻接表：
Vertex 0 -> (1, (1,3.14159)) (2, (1,3.14159)) (3, (10,0))
Vertex 1 -> (0, (1,3.14159)) (3, (1,3.14159))
Vertex 2 -> (0, (1,3.14159)) (3, (1,3.18))
Vertex 3 -> (0, (10,0)) (1, (1,3.14159)) (2, (1,3.18))
Vertex 4 ->
```

可以看到节点成功添加（自动编号）

选择 2：删除节点

```
当前电路邻接表:
Vertex 0 -> (1, (1,3.14159)) (2, (1,3.14159)) (3, (10,0))
Vertex 1 -> (0, (1,3.14159)) (3, (1,3.14159))
Vertex 2 -> (0, (1,3.14159)) (3, (1,3.18))
Vertex 3 -> (0, (10,0)) (1, (1,3.14159)) (2, (1,3.18))
Vertex 4 ->

===== 电路分析系统 =====
1. 添加节点
2. 删除节点
3. 删除阻抗
4. 设置/修改阻抗
5. 执行电路分析
6. 最短路径分析(Floyd)
0. 退出系统
请选择操作(0-6): 2
输入要删除的节点编号(留空删除最后一个节点): 3
节点删除成功！当前邻接表:
Vertex 0 -> (1, (1,3.14159)) (2, (1,3.14159))
Vertex 1 -> (0, (1,3.14159))
Vertex 2 -> (0, (1,3.14159))
Vertex 3 ->
```

可以看到 3 号节点成功删除，删除后节点被重新编号

选择 3：删除阻抗

```
当前电路邻接表:
Vertex 0 -> (1, (1,3.14159)) (2, (1,3.14159))
Vertex 1 -> (0, (1,3.14159))
Vertex 2 -> (0, (1,3.14159))
Vertex 3 ->

===== 电路分析系统 =====
1. 添加节点
2. 删除节点
3. 删除阻抗
4. 设置/修改阻抗
5. 执行电路分析
6. 最短路径分析(Floyd)
0. 退出系统
请选择操作(0-6): 3
输入要删除阻抗的两个节点编号(用空格分隔): 0 2
阻抗删除成功！当前邻接表:
Vertex 0 -> (1, (1,3.14159))
Vertex 1 -> (0, (1,3.14159))
Vertex 2 ->
Vertex 3 ->
```

输入了节点 0,2，可以看到 0,2 间的阻抗被成功删除（因为阻抗只有一个，所以 0->2,2->0 均删除）

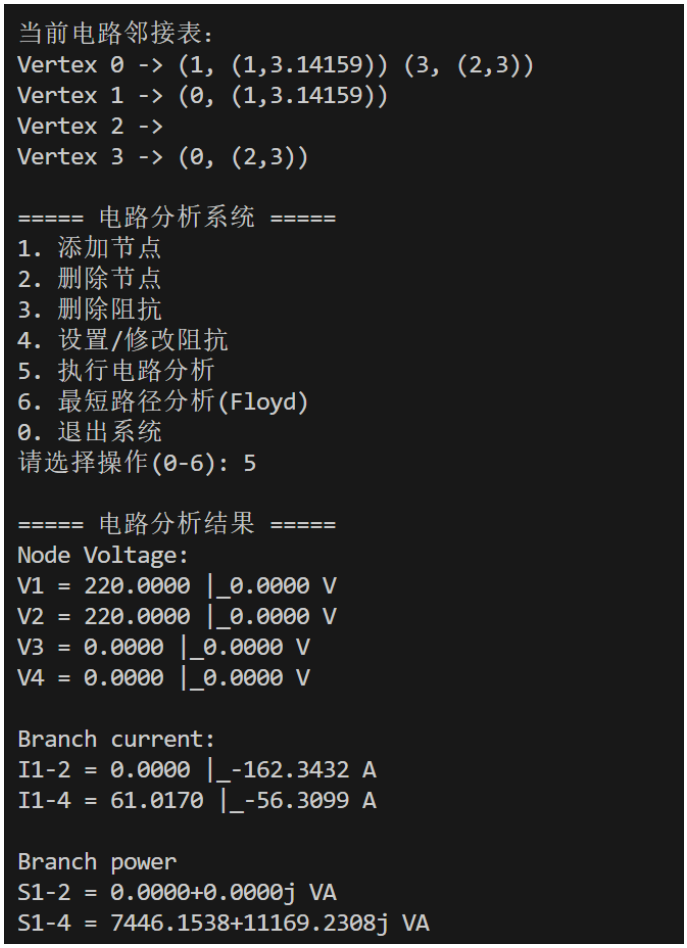
选择 4：设置/更改阻抗

```
当前电路邻接表:
Vertex 0 -> (1, (1,3.14159))
Vertex 1 -> (0, (1,3.14159))
Vertex 2 ->
Vertex 3 ->

===== 电路分析系统 =====
1. 添加节点
2. 删除节点
3. 删除阻抗
4. 设置/修改阻抗
5. 执行电路分析
6. 最短路径分析(Floyd)
0. 退出系统
请选择操作(0-6): 4
输入阻抗的两个节点编号(用空格分隔): 0 3
输入电阻和电抗值(用空格分隔): 2 3
阻抗设置成功！当前邻接表:
Vertex 0 -> (1, (1,3.14159)) (3, (2,3))
Vertex 1 -> (0, (1,3.14159))
Vertex 2 ->
Vertex 3 -> (0, (2,3))
```

输入了节点 0,3，以及它们间的阻抗设置 2+3j，得到更新后的邻接表（0->3,3->0 均自动设置）

选择 5：执行电路分析



输出当前的电路分析结果：各个节点电压，以及阻抗上的电流，损耗功率（默认电源为 220V 交流源，正极接 0 节点，负极接最后一个节点）

选择 6：Floyd

```
当前电路邻接表:
Vertex 0 -> (1, (1.0000,3.1416)) (3, (2.0000,3.0000))
Vertex 1 -> (0, (1.0000,3.1416))
Vertex 2 ->
Vertex 3 -> (0, (2.0000,3.0000))

===== 电路分析系统 =====
1. 添加节点
2. 删除节点
3. 删除阻抗
4. 设置/修改阻抗
5. 执行电路分析
6. 最短路径分析(Floyd)
0. 退出系统
请选择操作(0-6): 6

===== 最短路径分析 =====

Floyd Shortest Paths:
Path from 0 to 1: 0 -> 1 (Impedance: 3.2969)
Path from 0 to 3: 0 -> 3 (Impedance: 3.6056)
Path from 1 to 0: 1 -> 0 (Impedance: 3.2969)
Path from 1 to 3: 1 -> 0 -> 3 (Impedance: 6.9025)
Path from 3 to 0: 3 -> 0 (Impedance: 3.6056)
Path from 3 to 1: 3 -> 0 -> 1 (Impedance: 6.9025)
```

显示最短路径分析，可以据此推导各个支路上的电流流向情况
节点与阻抗输入错误会有相应的错误提示信息，要求用户重新输入，不再赘述，最后可以选择 0 退出程序。

4.2 QtCreator 运行 Qt 交互界面试例

如果有条件运行 Qt 文件，推荐这种展示方式，运行项目\src\Qtproject\release 路径下的 main.exe 文件

4.2.1 界面初始化

界面初始化界面展示如下：

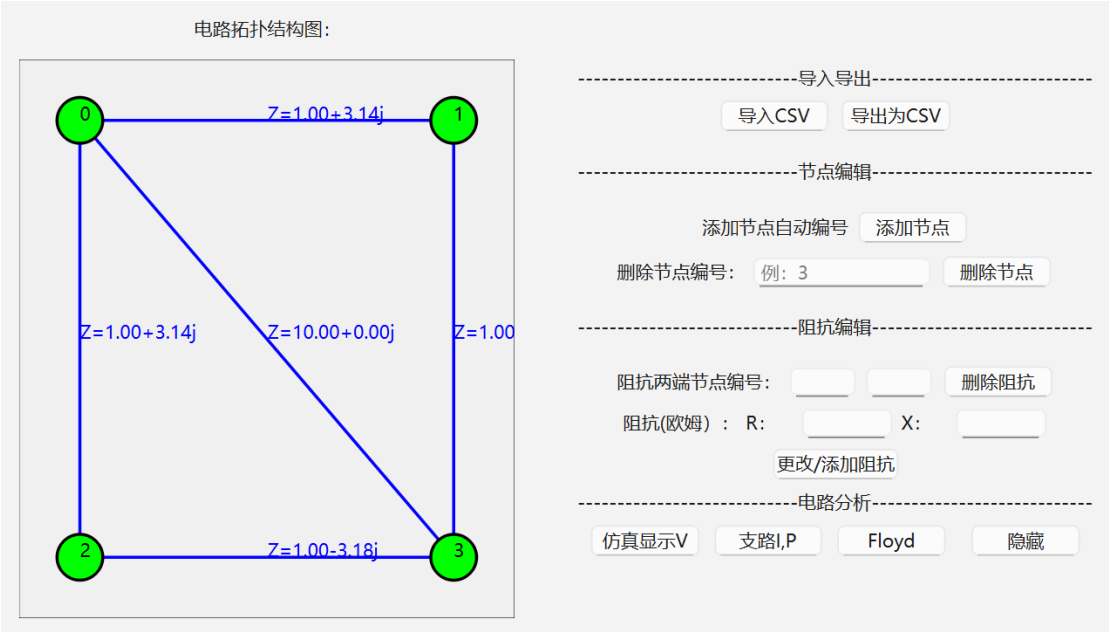


图 3：Qt 界面初始化

左图是根据项目配置的 example.csv 文件转换读取得到的电压节点以及节点间的阻抗连接显示，可以讲对应函数的输入文件名更改以加载别的 csv 配置矩阵，但并不可以在交互界面上修改。

4.2.2 添加节点

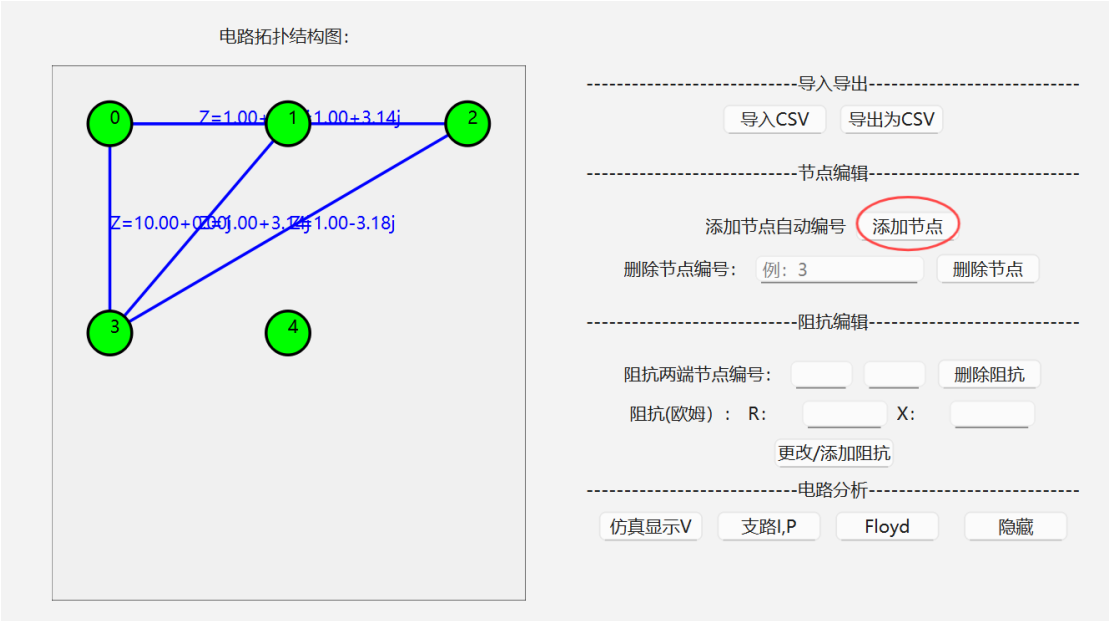


图 4：添加节点效果图

点击如图所示的添加节点位置，程序节点数量由 4 个变为 5 个，同时为了更灵活的展示，排列也发生变化，排列矩阵由原来的 2*2 变为 3*3，但依旧是从左到右，从上到下排列，节点间的连接关系也没有改变。（与此同时输出对应邻接表）

4.2.3 删除节点

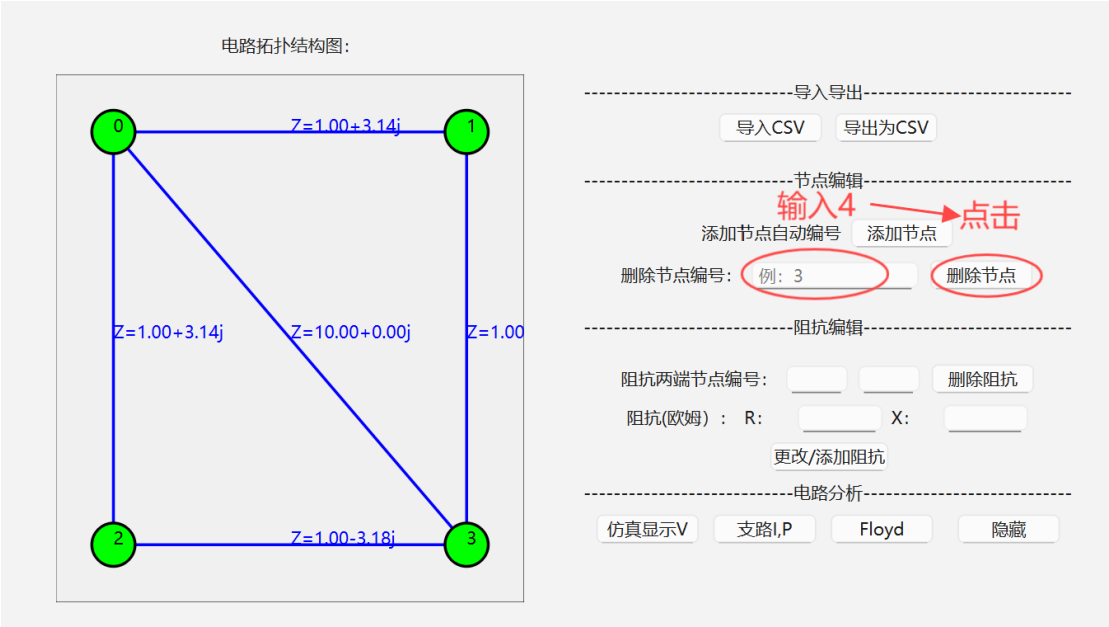


图 5：删除节点效果图

选择输入了节点 4 并点击删除，节点变回 4 个，排布也变回 2*2 矩阵，节点间连接关系与权重与之前相同。（与此同时输出对应邻接表）

4.2.4 删除阻抗

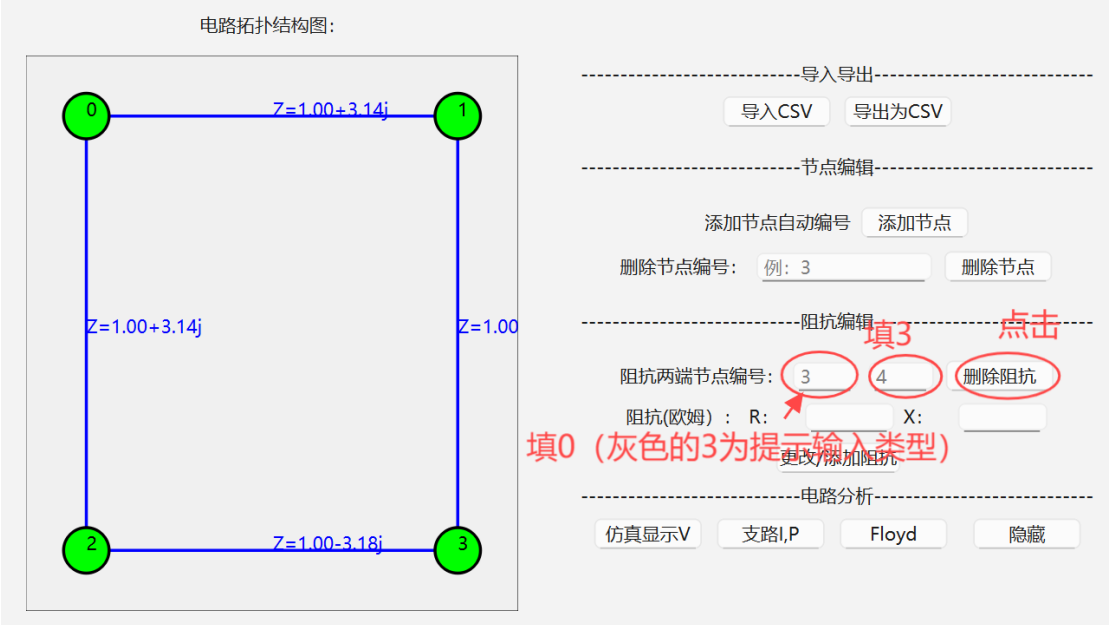


图 6：删除阻抗效果图

选择删除了 0,3 节点间的阻抗，拓扑图更新，阻抗消失（与此同时输出对应邻接表）

4.2.5 更改/添加阻抗

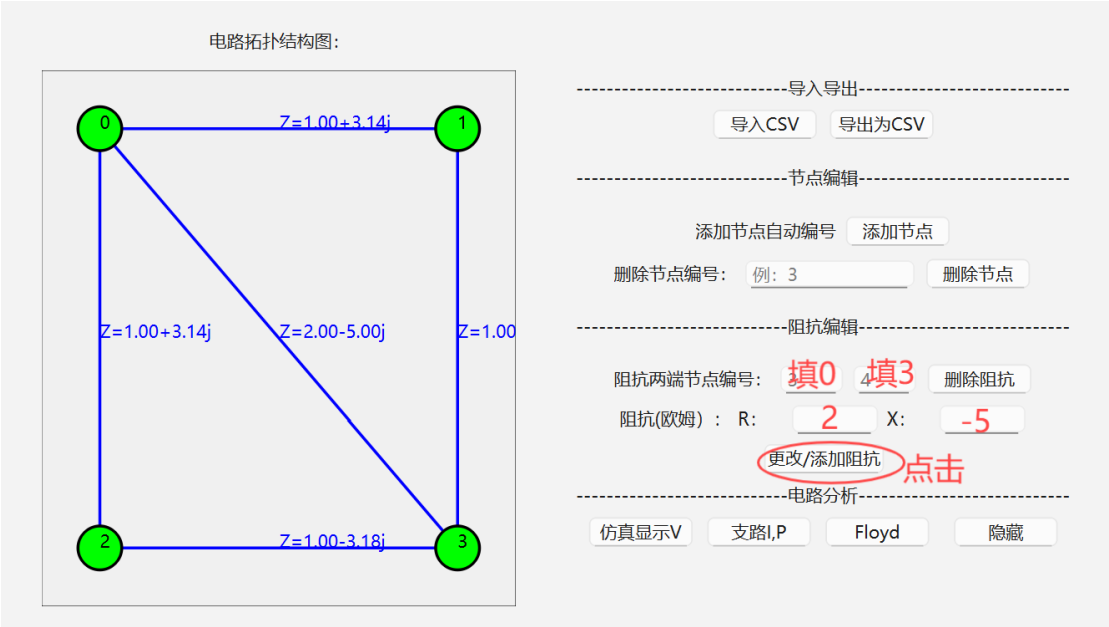


图 7：添加阻抗试例

选择在 0,3 节点间添加了电阻为 2 欧姆，电抗为-5 欧姆的阻抗，界面发生对应更改，与此同时输出对应邻接表（因为程序设置原因电阻输入值不能为负数，负数则程序无效，会在输入窗口返回错误信息）

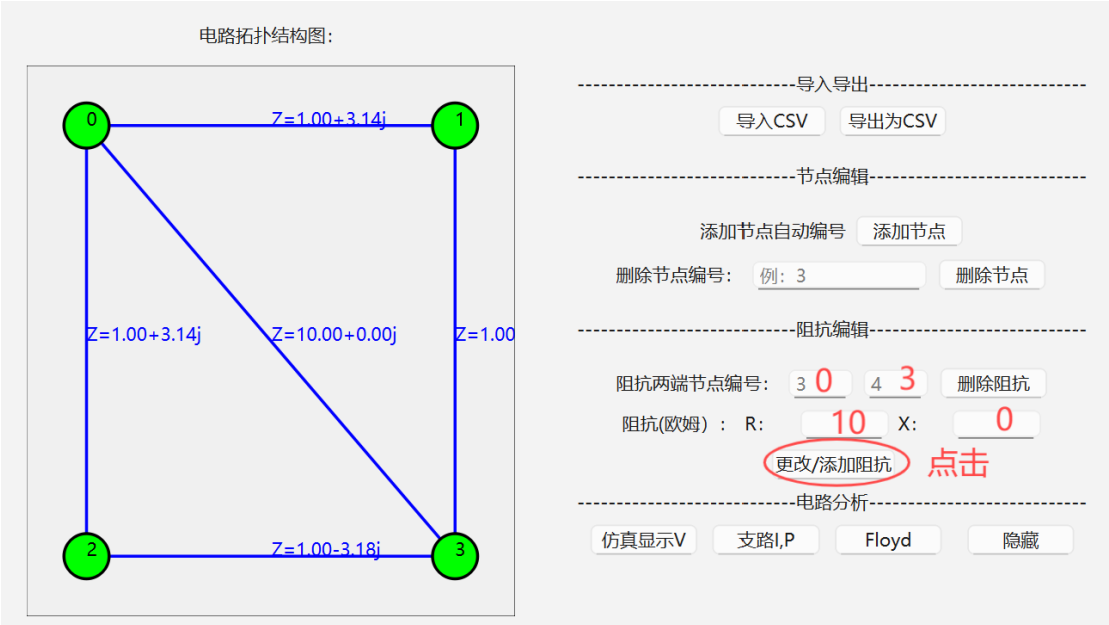


图 8：更改阻抗试例

将 0,3 节点之间的电阻修改回 10+0j 欧姆，电路显示图发生更改，输出对应邻接表

4.2.6 仿真显示 V

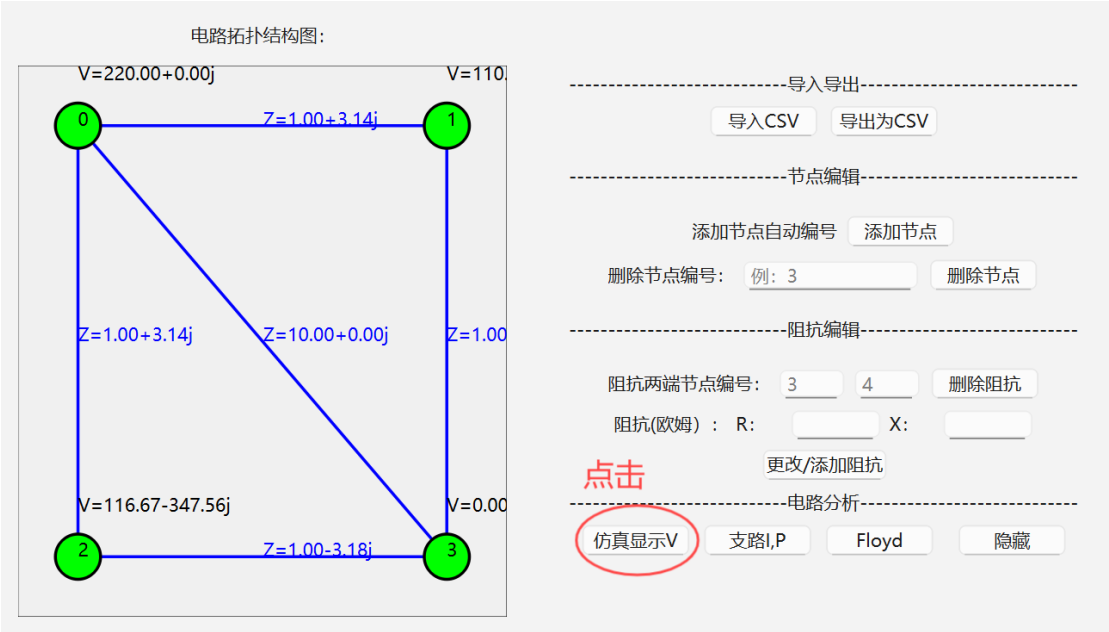


图 9：仿真显示 V 试例

点击仿真显示 V，开始进行电路仿真，默认为 220V 单个交流电压源，正极接待 0 节点上，负极在最后一个节点上（也就是 3 节点），各个节点上电压的值（以复数形式表示）会显示在途中的节点上方，因框的限制原因会有部分数据显示不全，可以在输出中看到完整的电路分析结果：

```
Circuit analysing! Print parameters:
Node Voltage:
V1 = 220.0000 |_0.0000 V
V2 = 110.0000 |_0.0000 V
V3 = 366.6203 |_-71.4432 V
V4 = 0.0000 |_72.8973 V

Branch current:
I1-2 = 33.3646 |_-72.3432 A
I1-3 = 109.9797 |_1.1002 A
I1-4 = 22.0000 |_-0.0000 A
I2-4 = 33.3646 |_-72.3432 A
I3-4 = 109.9797 |_1.1002 A

Branch power
S1-2 = 1113.1977+3497.2107j VA
S1-3 = 12095.5388+37999.2237j VA
S1-4 = 4840.0000+0.0000j VA
S2-4 = 1113.1977+3497.2107j VA
S3-4 = 12095.5388-38463.8133j VA
```

在完成后点击隐藏按钮，可以清空电压显示结果避免界面显示过于拥挤。

4.2.6 支路 I,P

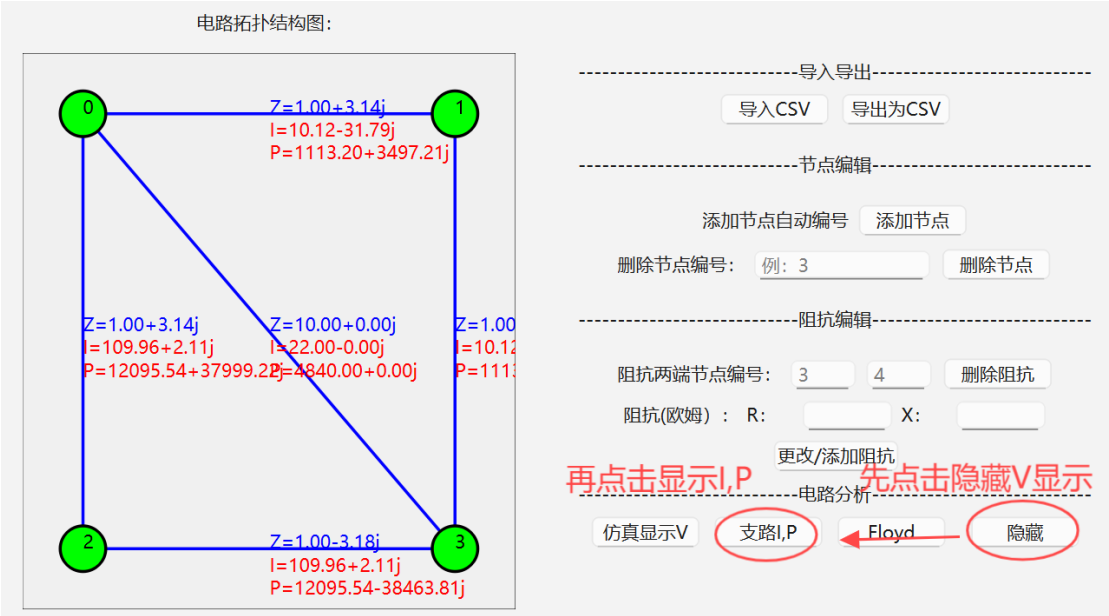


图 10：支路 I,P 显示试例

点击支路 I,P,可以看到阻抗上的电流与损耗功率显示在旁边（红色），可能会有部分显示不全，完整数据可以查看上文中程序输出的内容。

4.2.7 Floyd

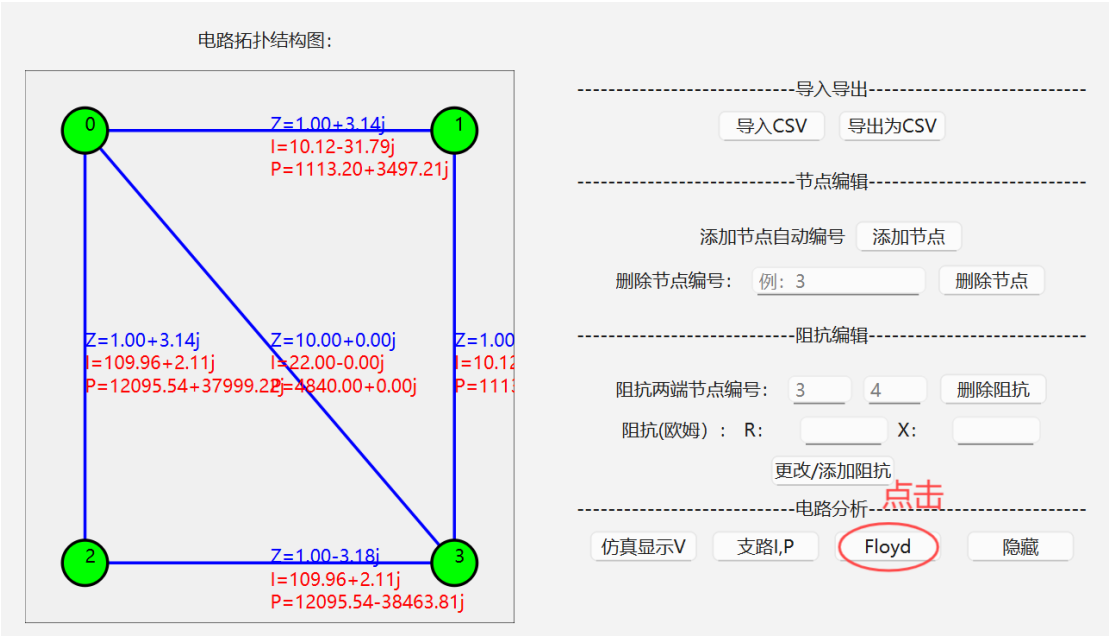


图 11：Floyd 显示

点击 Floyd 可以在输出框中输出完整的 Floyd 分析：

```
Floyd analysis
Path from 0 to 1: 0 -> 1 (Impedance: 3.2969)
Path from 0 to 2: 0 -> 2 (Impedance: 3.2969)
Path from 0 to 3: 0 -> 1 -> 3 (Impedance: 6.5938)
Path from 1 to 0: 1 -> 0 (Impedance: 3.2969)
Path from 1 to 2: 1 -> 0 -> 2 (Impedance: 6.5938)
Path from 1 to 3: 1 -> 3 (Impedance: 3.2969)
Path from 2 to 0: 2 -> 0 (Impedance: 3.2969)
Path from 2 to 1: 2 -> 0 -> 1 (Impedance: 6.5938)
Path from 2 to 3: 2 -> 3 (Impedance: 3.3335)
Path from 3 to 0: 3 -> 1 -> 0 (Impedance: 6.5938)
Path from 3 to 1: 3 -> 1 (Impedance: 3.2969)
Path from 3 to 2: 3 -> 2 (Impedance: 3.3335)
```

4.4.8 导入/导出为 CSV

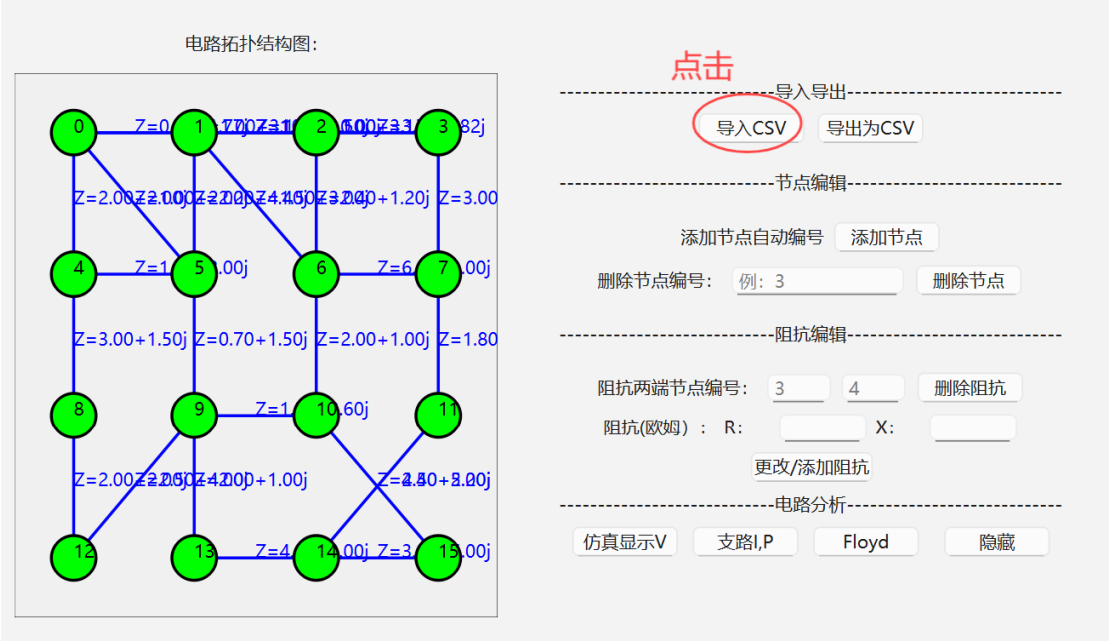


图 12：导入 CSV 展示

点击导入 CSV，自动导入成员之前设置好的相对复杂的电路拓扑结构，可以进行更加复杂的电路分析，点击导出为 CSV,会将当前电路拓扑图导出为 csv 文件存在原来 csv 文件的目录下（data 文件夹下）。

5. 总结

5.1 项目需求实现情况分析

本项目开发了一个电路仿真与分析系统，通过动态图结构、弗洛伊德算法、CSV 导入导出功能以及 Qt 可视化界面，实现了电路的构建、编辑、仿真和分析。

1. 动态图结构实现

成功实现了基于邻接表的动态图结构，支持顶点和边的动态增删操作；图结构能够自动扩展顶点数量，适应不同规模的电路配置；邻接表的实现确保了稀疏图的高效存储和操作，符合电路仿真的实际需求。

2. 弗洛伊德算法

实现了弗洛伊德算法，能够计算电路中各节点间的最短路径；算法结果通过终端输出为电路路径分析提供了支持。

3. CSV 导入导出功能

实现了从 CSV 文件导入电路配置信息的功能，能够快速构建图结构；实现了将当前电路状态导出为 CSV 文件的功能，确保数据的灵活性。

4. 电路仿真与分析

实现了电路仿真功能，能够计算各节点电压、支路电流和功率分布；仿真结果通过可视化界面展示，用户可以直观地查看电路的运行状态。

5. Qt 可视化界面

设计并实现了用户友好的 Qt 界面，支持电路的构建、编辑和仿真结果的展示；界面操作清晰便捷，可以提供良好的用户体验。

5.2 遇到的困难及解决方法

1. 小组协作与代码整合

小组成员在开发过程中，由于各自负责的模块不同，变量类型和代码结构可能存在差异，导致代码整合时出现了兼容性问题。在整合过程中，使用了 Git 进行版本控制和代码管理，解决了代码同步和冲突问题。

2. Qt 导入文件相对路径出错

在开发过程中，发现使用相对路径导入 CSV 文件时，文件无法正确加载。这个问题会发生在部分同学的电脑上，而在其他同学的电脑上却能够正常运行，所以可能需要用户手动添加绝对路径，确保程序能够正常运行。

3. 电路仿真解算中的技术难点之复数运算与相位处理

在电路仿真解算过程中，复数运算和相位处理是一个重要的技术难点。由于交流电路中的电压和电流都是复数形式，需要同时考虑幅值和相位信息。在实现过程中，需要特别注意复数运算的规则，特别是在计算功率时，必须使用电压与电流的共轭复数相乘，才能得到正确的复功率值。此外，在处理相位角时，需要正确地进行弧度与角度的转换，并注意相位角的范围限制。这些复杂的运算不仅增加了代码实现的难度，也对算法的准确性和稳定性提出了更高的要求。为了确保计算结果的准确性，需要在代码中实现严格的数值精度控制和边界条件处理。

4. 电路仿真解算中的技术难点之矩阵求解与数值稳定性

电路仿真中的节点电压法需要求解大型线性方程组，这涉及到矩阵运算和数值稳定性问题。在构建导纳矩阵时，需要正确处理电压源和电流源的约束条件，这可能导致矩阵结构的变化。使用 QR 分解方法求解方程组时，需要特别注意矩阵的条件数，避免出现数值不稳定的情况。此外，在处理大型电路时，矩阵的维数可能很大，这对内存使用和计算效率都提出了挑战。为了提高求解的稳定性和效率，需要在算法实现中采用适当的数值方法，如部分主元选择、矩阵预处理等技术，同时还需要优化数据结构，减少内存占用和提高计算速度。