GraphEditorPlus 作业报告

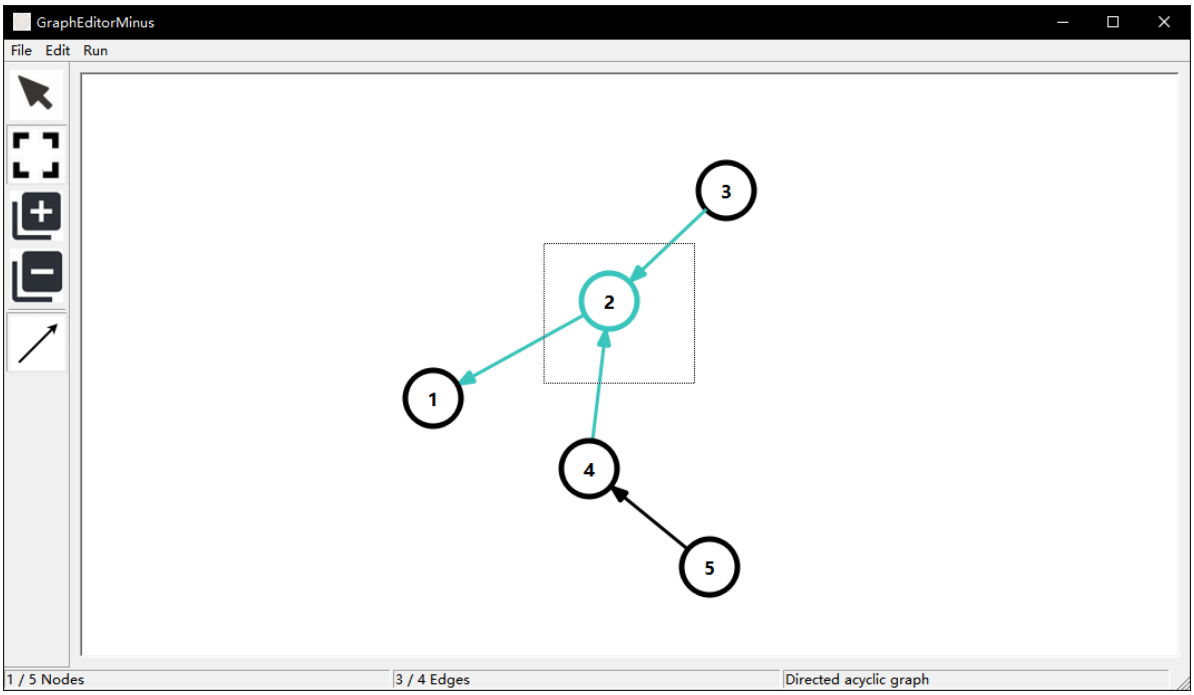
98 组：文豪、郭宇铎、晋禹超

https://github.com/wenhao801/GraphEditorPlus

功能介绍

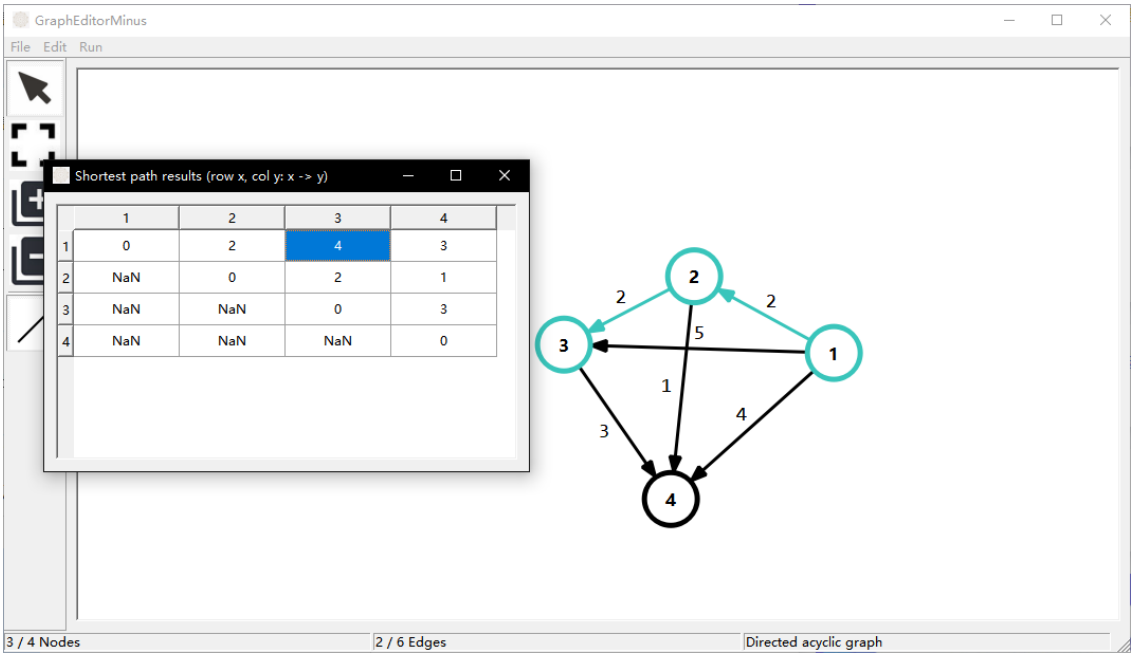
参考 CSAcademy's graph editor、Graphviz 等工具，我们实现了一个可以用作图论研究的图编辑器。

图论中的图定义为集合二元组 (V, E)，其中 V 为点集，E 为边集，$E \subseteq \{(u, v) | u, v \in V\}$。此编辑器对点边支持多种增删改查方式，也可以支持更换颜色、自动布局、运行算法等功能。下面将分各个模块进行介绍。



上方工具栏：

- File，支持 New / Open / Save(As)
- Edit，
  - 通过文本形式加点/加边：生成的点会按照一定规则，随机选择一个较为远离已有点的位置，让图尽量美观。
  - 设置自动编号起点：使用鼠标点击的形式加点时，可以自动编号新加的点。
  - 自动布局：参考 Fruchterman-Reingold 算法，通过模拟点边之间的引力/斥力，将图上的点自动布局为一个较为美观的形式。
  - 控制台：以表格的形式列出当前的所有点边，便于在边有重叠时进行选择/重命名/更改等操作。会按照 被选择先于未被选择、点先于边、边字典序 的优先级依次排序，并按照图的状态实时更新。
- Run，支持对整张图进行最短路、最小生成树（最小生成森林）算法的运行（如果边权都为数字，或默认为1）。运行最短路后会生成一张表格，点击相应格子即可看到一条最短路径（支持负环）；最小生成树则会报告最小边权和，并直接选中一棵树。



左侧模式栏：

- 移动模式：左键点击并拖拽实现单点移动；点击并拖拽空白位置实现画布移动（画布会自动扩展）。被拖拽的点的 z-index 将被置为最高。
- 选择模式：可以框选出若干点/边。
- 添加模式：点击空白位置加点（自动编号）；从已有的点按住鼠标并拖动到另一点上可以加边。
- 删除模式：直接点击相应的点/边，或者框选出若干点/边进行删除。
- 点击有向边按钮也可以切换有向边/无向边。

一些模式下，双击可以改变点的名字/边权。点的名字不能有重复（重复时会提示冲突），但边权可以。

右键菜单：支持对选中的元素进行更改颜色/随机连接成特殊图/……，也支持工具栏中 Edit / Run 相关的功能。

滚轮：Ctrl + 滚轮可以实现缩放。参考了官方文档中的代码。

下方状态栏：可以显示选定的点数/边数，以及整个图的类型（特殊信息），如有向无环图、树、森林……。

各模块与类设计细节

MainWindow 类

作为容纳 MyScene 的主窗口，处理工具栏相关交互。核心功能不在这里，在 MyScene 中。

```
public:
    void switchMode();
    MyScene *scene;
private slots: // 以下函数名都是 Qt Creator 生成的
    void on_actionFrom_Text_triggered();
    void on_actionNew_triggered();
    void on_actionSave_triggered();
    void on_actionOpen_triggered();
    void on_actionSaveAs_triggered();

    void on_actionSet_autoindex_triggered();
    void on_actionAuto_layout_triggered();
```

```
    void on_actionSelection_editor_triggered();
    void on_actionShortest_Path_triggered();
    void on_actionMinimum_spanning_tree_triggered();
```

MyScene 类

MyScene 继承自 QGraphicsScene，即画布，包含了处理加删元素事件、鼠标键盘事件的相关函数和相关成员变量。与其他类的联系有

```
QMap <QString, MyNode*> ids; // 我们令每个点的名字都唯一，因此有名字到 MyNode* 对象的双射。
std::set <MyNode*> nodes; // 存储画布上已有的所有点。
std::set <MyEdge*> edges; // 以及所有边。使用 set 是为了方便地通过指针删除已有对象。

EditWindow editWindow; // Console 的窗口。会随着图的变化而变化。
SPWindow *spwindow; // ShortestPath 的结果窗口。当图的状态改变时，窗口将自动关闭。
```

此外，其他重要的函数和变量有

```
protected:
    void mousePressEvent(QGraphicsSceneMouseEvent *event) override;
    void mouseMoveEvent(QGraphicsSceneMouseEvent *event) override;
    void mouseReleaseEvent(QGraphicsSceneMouseEvent *event) override;
    void mouseDoubleClickEvent(QGraphicsSceneMouseEvent *mouseEvent) override;
    void keyPressEvent(QKeyEvent *keyEvent) override; // 支持 Ctrl + A, Ctrl + S, delete...
    void contextMenuEvent(QGraphicsSceneContextMenuEvent *event) override;
public:
    MyNode* addNode(qreal x, qreal y, QString _name = nullptr);
    void delNode(MyNode *);
    void nameNode(MyNode *, QString); // 以上三个也有 Edge 版本。
    void edgeChangeStart(MyEdge *, MyNode *);
    void edgeChangeEnd(MyEdge *, MyNode *);

    enum CursorMode { MoveMode, SelectMode, AddMode, DeleteMode } curMode = MoveMode; // 当前模式
    void switchMode(CursorMode mode);
    bool directed = 1; // 有向/无向
    void toggleDirect();

    void updateStatusBar();
    void FRlayout(QList <QGraphicsItem*>); // 使用 Fruchterman-Reingold 算法自动布局。
    void showEditConsole();
    void linkChain(QList <QGraphicsItem*>);
    void linkTree(QList <QGraphicsItem*>);
    void linkComplete(QList <QGraphicsItem*>);
    void shortestPath(QList <QGraphicsItem*>); // 使用 Bellman-Ford 算法求最短路（判负环）。
    void closeSPWindow();
    void MST(QList <QGraphicsItem*> items); // 使用 Kruskal 算法求最小生成树。
    void ChangeColor(QList <QGraphicsItem*>, QColor Color); // 以上是 Edit / Run 相关的一些功能，均可顾名思义，不赘述。
```

MyNode 类

MyNode 继承自 QGraphicsEllipseItem。

```
enum { Type = UserType + 1 };
int type() const override { return Type; } // 重载相关函数，用于在使用 QGraphicsItem* 时区分点/边

void updateMode(); // 当模式切换时，调用相关函数使点 可以/不可以 被 移动/选中/……

MyScene *scene;
QGraphicsSimpleTextItem *name = nullptr; // 内部含有的子 Item，用于显示点的名字

QColor color = QColor(0, 0, 0); // 点的颜色
const int penSize = 5, radius = 25; // 画 border 时笔的粗细、圆的半径

std::set <MyEdge*> inEdge, outEdge; // 仍然用 set 存入边/出边。在无向边模式下，这两个不做区分
```

MyEdge 类

MyEdge 继承自 QGraphicsLineItem。箭头的绘制通过重载 paint 函数实现。

边权现在会绘制在有向线段的中点靠右侧。

```
const int penSize = 3;
enum { Type = UserType + 2 };
int type() const override { return Type; }

MyNode *startNode, *endNode; // 起点，终点
QGraphicsSimpleTextItem *weight; // 边权

MyScene *scene;
QColor color = QColor(0, 0, 0); // 边的颜色

void updateMode();
MyNode* ad(MyNode *u) { return u == startNode ? endNode : startNode; } // 便于从一点找到另一点
```

MyView 类

继承自 QGraphicsView 类。这个类的创建只是为了实现滚轮缩放。

```
protected:
    void wheelEvent(QWheelEvent *event) override;
private:
    int _numScheduledScalings = 0; // 向当前方向（指放大/缩小）计划进行缩放多少次
    double curFactor = 1; // 当前倍率，为了美观，控制在 [1/5, 5]
private slots:
    void scalingTime(qreal x); // 每帧动画缩放一次
    void animFinished();
```

EditWindow 类

继承自窗口 QWidget，用来作为 Console 的窗口。

```
public:
    ConsoleTable *tableWidget; // 里面的表格
    QList <QGraphicsItem*> items; // 涉及到的所有点边
    MyScene *scene;
    void updateTable(); // 核心函数：排序、填表
    bool userEditing = 0; // 填完表之后，用户用右键修改表，就在图上做相应的修改（若修改非法，就不动）。
protected:
    void closeEvent(QCloseEvent *event) override; // 关闭窗口时改为 hide 而非 delete
private slots:
    void toggleSelection(QTableWidgetItem *item); // 当选中一行时触发，更改图上已选中的元素
    void editItem(QTableWidgetItem *item); // 用右键编辑时触发
```

ConsoleTable 类

继承自 QTableWidget。创建这个类只是为了区分左右键，将相应的信号传给 EditWindow。全部代码如下：

```
void ConsoleTable::mousePressEvent(QMouseEvent *event) {
    if (event->button() == Qt::RightButton) {
        QTableWidgetItem *item = itemAt(event->pos());
        if (item != nullptr) {
            editItem(item);
        }
    }
    QTableWidget::mousePressEvent(event);
}
```

SPWindow 类

继承自窗口 QWidget。创建这个类是为了保存最短路的结果，以便在用户点击时显示。

```
public:
    MyScene *scene;
    QTableWidget tableWidget;
    QList <MyNode*> V;
    QList <MyEdge*> E;
    QMap <QPair<MyNode*, MyNode*>, MyEdge*> from, first; // 存了以每个点为起点时，每个点第一次被更新是由哪条边，以便可视化负环。
    QMap <QPair<MyNode*, MyNode*>, QString> ans;

    bool updateSP(QList <QGraphicsItem*> items); // 核心函数，使用 Bellman-Ford 算法求最短路。
protected:
    void closeEvent(QCloseEvent *event) override;

private slots:
    void selectCell(int r, int c); // 当选中格子时，给出最短路方案。
```

InsertFromText 类

用作从文本插入图，核心即

```
void InsertFromText::on_buttonBox_accepted()
{
    QString text = ui->plainTextEdit->document()->toPlainText();
    scene->insertFromText(text);
}
```

使用 Qt Creator 做了窗口的相关排版，后来发现可以直接使用代码实现。

小组分工情况

文豪：算法相关工作，例如在程序中实现 Bellman-Ford、Kruskal 算法，以及实现 FR 自动布局算法等。

晋禹超：界面设计相关工作，如窗口的布局、点边的绘制和个性化设置等。

郭宇铎：交互窗口相关工作，例如存读档、与用户交互（弹出窗口）等。

项目总结与反思

经过几个月的学习与实践，我们基本实现了最初规划的功能。

多人协作使用 Qt 编写图形化界面程序，不仅需要学会查阅官方文档、使用 Git 等版本控制工具，还需要思考最佳的交互方式、处理用户不合法的输入等各种问题。例如，用户选择一个点后，点需要被置于画布的最上层；针对多条边重叠的问题，我们需要引入 Console 来更好地编辑；运行算法后，运行结果有“时效性”；算法题中往往保证边权是正整数，然而实际应用中，我们甚至难以保证边权是数……相比起做算法题，编写多入口多出口的应用程序需要考虑很多核心算法之外的事情，我们的程序设计能力也因这次作业变得更为全面。

功能方面，虽然我们大体实现了一个流畅的编辑体验，但各个方面仍有很大的改进空间，例如

- 不支持撤销操作。撤销操作往往需要记录操作栈或记录每一时刻的状态，实现起来较为困难。
- 不支持复制粘贴。这是点编号的唯一性所带来的不便之处，若后续想扩展出计算图模拟功能，应该做相关改进。
- 点、边个性化能力不强。从一开始 MyNode 和 MyEdge 就分别继承自 椭圆、直线的 Item，这使得我们难以做出矩形点或曲线边，也就同样带来了两点间有多条边时的难以显示问题。如果可以高度个性化点和边，可以很方便地实现各种算法、数据结构的可视化。

综合来看，我们通过这次作业，不仅学习到了图论方面的相关知识，还实践了真实桌面端程序开发，通过解决种种问题全面锤炼了程序设计能力，收获颇丰。