Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение

высшего образования

**«Пермский национальный** **исследовательский политехнический университет»**

Электротехнический факультет

Кафедра «Информационные технологии и автоматизированные системы»

направление подготовки: 09.03.01– «Информатика и вычислительная техника»

**Лабораторная работа**

**по теме**

**«Графы»**

Выполнил студент гр. ИВТ-24-1б

Оглезнев Никита Михайлович

Проверил:

доц. каф. ИТАС

Яруллин Денис Владимирович

(оценка) (подпись)

(дата)

г. Пермь, 2024

1. **Постановка задачи**

Целью данной работы является получение практических навыков работы с графами, включая их визуализацию и применение алгоритмов поиска кратчайших путей.

Задачей является реализация ориентированного графа с возможностью динамического изменения структуры, визуализации, выполнения алгоритма Флойда, а также обхода графа.

1. **Анализ задачи**

Для решения поставленной задачи определены следующие компоненты:

* Класс GraphWindow – основное окно приложения, содержащее графическую сцену для отображения графа, таблицу смежности и элементы управления.
* Класс ResultWindow – вспомогательное окно для отображения матрицы кратчайших путей
* Класс CustomGraphicsView – виджет для отображения графической сцены
* Функции добавления/удаления вершин, обработки изменений в таблице смежности, реализация алгоритма Флойда, обход графа

1. **Реализация**
   1. **Структура данных**

Граф представлен матрицей смежности (adjacencyMatrix), где значение adjacencyMatrix[i][j] соответствует весу ребра из вершины i в j. Вершины хранятся в контейнере QMap<int, QString> с генерацией имён (v1. v2…) вершин графа.

* 1. **Визуализация графа**

Вершины графа отображаются в виде кругов, размещающихся по окружности, ребра рисуются линиями с направлением графа, вес отображается на экране.

Для отрисовки используется QGraphicsScene и элементы QGraphicsEllioseItem для вершин, QGraphicsLineItem для рёбер.

1. **Код программы**

#include <QApplication>

#include <QMainWindow>

#include <QGraphicsScene>

#include <QGraphicsView>

#include <QTableWidget>

#include <QPushButton>

#include <QVBoxLayout>

#include <QHBoxLayout>

#include <QWidget>

#include <QGraphicsEllipseItem>

#include <QGraphicsLineItem>

#include <QGraphicsTextItem>

#include <QInputDialog>

#include <QMap>

#include <QHeaderView>

#include <QtMath>

#include <limits>

#include <QQueue>

#include <QTimer>

#include <QTextEdit>

#include <QMouseEvent>

class CustomGraphicsView : public QGraphicsView {

public:

    CustomGraphicsView(QGraphicsScene\* scene, QWidget\* parent = nullptr)

        : QGraphicsView(scene, parent) {}

protected:

    void wheelEvent(QWheelEvent\* event) override {

        event->ignore();

    }

};

class ResultWindow : public QDialog {

public:

    ResultWindow(const QVector<QVector<int>>& matrix, const QString& title, QWidget\* parent = nullptr)

        : QDialog(parent, Qt::Window | Qt::WindowTitleHint | Qt::CustomizeWindowHint) {

        setWindowTitle(title);

        QTableWidget\* table = new QTableWidget(matrix.size(), matrix.size());

        table->setEditTriggers(QAbstractItemView::NoEditTriggers);

        table->horizontalHeader()->setSectionResizeMode(QHeaderView::Stretch);

        table->verticalHeader()->setSectionResizeMode(QHeaderView::Stretch);

        table->setSizeAdjustPolicy(QTableWidget::AdjustToContents);

        for(int i = 0; i < matrix.size(); ++i) {

            for(int j = 0; j < matrix.size(); ++j) {

                QTableWidgetItem\* item = new QTableWidgetItem();

                item->setTextAlignment(Qt::AlignCenter);

                item->setFlags(item->flags() & ~Qt::ItemIsEditable);

                if(i == j) item->setBackground(QColor(255, 200, 200));

                if(matrix[i][j] == std::numeric\_limits<int>::max()/2)

                    item->setText("-");

                else if(matrix[i][j] < 0)

                    item->setText("X");

                else

                    item->setText(QString::number(matrix[i][j]));

                table->setItem(i, j, item);

            }

        }

        QVBoxLayout\* layout = new QVBoxLayout(this);

        layout->addWidget(table);

        setMinimumSize(400, 400);

    }

protected:

    void mousePressEvent(QMouseEvent\* event) override {

        if(event->button() == Qt::LeftButton) {

            m\_dragging = true;

            m\_dragPosition = event->globalPosition().toPoint() - frameGeometry().topLeft();

            event->accept();

        }

    }

    void mouseMoveEvent(QMouseEvent\* event) override {

        if(m\_dragging && (event->buttons() & Qt::LeftButton)) {

            move(event->globalPosition().toPoint() - m\_dragPosition);

            event->accept();

        }

    }

    void mouseReleaseEvent(QMouseEvent\* event) override {

        Q\_UNUSED(event);

        m\_dragging = false;

    }

private:

    bool m\_dragging = false;

    QPoint m\_dragPosition;

};

class GraphWindow : public QMainWindow {

    Q\_OBJECT

public:

    GraphWindow(QWidget \*parent = nullptr) : QMainWindow(parent) {

        setupUI();

    }

private slots:

    void addVertex() {

        int newSize = adjacencyMatrix.size() + 1;

        adjacencyMatrix.resize(newSize);

        for(auto& row : adjacencyMatrix) row.resize(newSize);

        vertexNames[newSize-1] = QString("v%1").arg(newSize);

        updateTable();

        visualizeGraph();

    }

    void removeVertex() {

        if(vertexNames.isEmpty()) return;

        bool ok;

        QString vertex = QInputDialog::getItem(this, "Удаление вершины",

                                               "Выберите вершину:", vertexNames.values(), 0, false, &ok);

        if(!ok) return;

        int idx = vertexNames.key(vertex);

        adjacencyMatrix.remove(idx);

        for(auto& row : adjacencyMatrix) row.remove(idx);

        vertexNames.remove(idx);

        QMap<int, QString> newNames;

        int newIdx = 0;

        for(auto& name : vertexNames.values()) {

            newNames[newIdx++] = name;

        }

        vertexNames = newNames;

        updateTable();

        visualizeGraph();

    }

    void cellChanged(int row, int col) {

        if(row == col) return;

        QString text = matrixTable->item(row, col)->text();

        bool ok;

        int value = text.toInt(&ok);

        if(ok) {

            if(value < 0) {

                adjacencyMatrix[row][col] = -value;

                adjacencyMatrix[col][row] = -value;

            } else {

                adjacencyMatrix[row][col] = value;

                adjacencyMatrix[col][row] = 0;

            }

        } else {

            adjacencyMatrix[row][col] = 0;

            adjacencyMatrix[col][row] = 0;

        }

        updateTable();

        visualizeGraph();

    }

    void runFloyd() {

        if(adjacencyMatrix.isEmpty()) return;

        int n = adjacencyMatrix.size();

        QVector<QVector<int>> dist(n, QVector<int>(n));

        for(int i = 0; i < n; ++i) {

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

                dist[i][j] = (i == j) ? 0 :

                                 (adjacencyMatrix[i][j] > 0) ? adjacencyMatrix[i][j] :

                                 std::numeric\_limits<int>::max()/2;

            }

        }

        for(int k = 0; k < n; ++k)

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

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

                    if(dist[i][k] + dist[k][j] < dist[i][j])

                        dist[i][j] = dist[i][k] + dist[k][j];

        ResultWindow\* w = new ResultWindow(dist, "Матрица кратчайших путей");

        w->show();

    }

    void traverseGraph() {

        logText->clear();

        if(vertexNames.isEmpty()) return;

        bool ok;

        QString startVertex = QInputDialog::getItem(this, "Выбор вершины",

                                                    "Выберите стартовую вершину:",

                                                    vertexNames.values(), 0, false, &ok);

        if(!ok) return;

        int startIdx = vertexNames.key(startVertex);

        QQueue<int> queue;

        QVector<bool> visited(adjacencyMatrix.size(), false);

        queue.enqueue(startIdx);

        visited[startIdx] = true;

        QTimer\* timer = new QTimer(this);

        connect(timer, &QTimer::timeout, [=]() mutable {

            if(queue.isEmpty()) {

                timer->stop();

                timer->deleteLater();

                resetEdgeColors();

                resetVertexColors();

                logText->append("◼ Обход завершен!");

                return;

            }

            int current = queue.dequeue();

            highlightVertex(current, QColor("#90EE90"));

            logText->append("● Посещена вершина: " + vertexNames[current]);

            // Process neighbors with delays

            QTimer::singleShot(800, [=]() mutable {

                for(int neighbor = 0; neighbor < adjacencyMatrix.size(); ++neighbor) {

                    if(adjacencyMatrix[current][neighbor] > 0 && !visited[neighbor]) {

                        highlightEdge(current, neighbor, QColor("#FF6347"));

                        QTimer::singleShot(800, [=]() mutable {

                            visited[neighbor] = true;

                            queue.enqueue(neighbor);

                            highlightVertex(neighbor, QColor("#90EE90"));

                        });

                    }

                }

            });

        });

        timer->start(1500);

        resetEdgeColors();

        resetVertexColors();

    }

private:

    void setupUI() {

        QWidget\* centralWidget = new QWidget;

        QHBoxLayout\* mainLayout = new QHBoxLayout(centralWidget);

        scene = new QGraphicsScene(this);

        CustomGraphicsView\* view = new CustomGraphicsView(scene);

        view->setRenderHint(QPainter::Antialiasing);

        mainLayout->addWidget(view, 3);

        QWidget\* rightPanel = new QWidget;

        QVBoxLayout\* rightLayout = new QVBoxLayout(rightPanel);

        rightLayout->setSpacing(5);

        QPushButton\* addBtn = new QPushButton("Добавить вершину");

        QPushButton\* removeBtn = new QPushButton("Удалить вершину");

        QPushButton\* floydBtn = new QPushButton("Алгоритм Флойда");

        QPushButton\* traverseBtn = new QPushButton("Обход графа");

        matrixTable = new QTableWidget;

        matrixTable->setStyleSheet("QTableWidget {font-size: 10pt;}");

        matrixTable->horizontalHeader()->setSectionResizeMode(QHeaderView::Stretch);

        matrixTable->verticalHeader()->setSectionResizeMode(QHeaderView::Stretch);

        matrixTable->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);

        connect(matrixTable->horizontalHeader(), &QHeaderView::geometriesChanged, this, [this]() {

            matrixTable->setFixedHeight(matrixTable->width());

        });

        logText = new QTextEdit;

        logText->setReadOnly(true);

        logText->setPlaceholderText("Журнал обхода...");

        logText->setStyleSheet("color: gray;");

        logText->setMinimumHeight(200);

        rightLayout->addWidget(addBtn);

        rightLayout->addWidget(removeBtn);

        rightLayout->addWidget(floydBtn);

        rightLayout->addWidget(traverseBtn);

        rightLayout->addWidget(matrixTable);

        rightLayout->addWidget(logText);

        mainLayout->addWidget(rightPanel, 1);

        setCentralWidget(centralWidget);

        connect(addBtn, &QPushButton::clicked, this, &GraphWindow::addVertex);

        connect(removeBtn, &QPushButton::clicked, this, &GraphWindow::removeVertex);

        connect(floydBtn, &QPushButton::clicked, this, &GraphWindow::runFloyd);

        connect(traverseBtn, &QPushButton::clicked, this, &GraphWindow::traverseGraph);

        connect(matrixTable, &QTableWidget::cellChanged, this, &GraphWindow::cellChanged);

    }

    void updateTable() {

        matrixTable->blockSignals(true);

        matrixTable->setRowCount(adjacencyMatrix.size());

        matrixTable->setColumnCount(adjacencyMatrix.size());

        QStringList headers;

        for(int i = 0; i < adjacencyMatrix.size(); i++)

            headers << vertexNames[i];

        matrixTable->setHorizontalHeaderLabels(headers);

        matrixTable->setVerticalHeaderLabels(headers);

        for(int i = 0; i < adjacencyMatrix.size(); ++i) {

            for(int j = 0; j < adjacencyMatrix.size(); ++j) {

                QTableWidgetItem\* item = new QTableWidgetItem();

                item->setTextAlignment(Qt::AlignCenter);

                if(i == j) {

                    item->setText("-");

                    item->setBackground(QColor(255, 200, 200));

                    item->setFlags(item->flags() & ~Qt::ItemIsEditable);

                }

                else {

                    if(adjacencyMatrix[i][j] > 0) {

                        item->setText(QString::number(adjacencyMatrix[i][j]));

                    } else {

                        if(adjacencyMatrix[j][i] > 0) {

                            item->setText("-");

                        } else {

                            item->setText("");

                        }

                    }

                }

                matrixTable->setItem(i, j, item);

            }

        }

        matrixTable->blockSignals(false);

    }

    void visualizeGraph() {

        scene->clear();

        nodeItems.clear();

        edgeTexts.clear();

        const int radius = 30;

        const int spacing = 200;

        QVector<QPointF> positions;

        for(int i = 0; i < adjacencyMatrix.size(); ++i) {

            double angle = 2 \* M\_PI \* i / adjacencyMatrix.size();

            QPointF pos(

                spacing \* cos(angle) + 300,

                spacing \* sin(angle) + 300

                );

            positions.append(pos);

        }

        for(int i = 0; i < adjacencyMatrix.size(); ++i) {

            for(int j = 0; j < adjacencyMatrix.size(); ++j) {

                if(adjacencyMatrix[i][j] > 0 && i != j) {

                    QLineF line(

                        positions[i] + QPointF(radius/2, radius/2),

                        positions[j] + QPointF(radius/2, radius/2)

                        );

                    QGraphicsLineItem\* edge = scene->addLine(line, QPen(Qt::gray, 1));

                    edge->setZValue(0);

                    QGraphicsTextItem\* text = scene->addText(QString::number(adjacencyMatrix[i][j]));

                    text->setDefaultTextColor(Qt::darkBlue);

                    QPointF textPos = (line.p1() + line.p2()) / 2;

                    text->setPos(textPos.x() - 10, textPos.y() - 10);

                    text->setZValue(4);

                    edgeTexts.append(text);

                    bool isBidirectional = (adjacencyMatrix[j][i] > 0);

                    drawArrow(scene, line, radius, isBidirectional);

                }

            }

        }

        for(int i = 0; i < adjacencyMatrix.size(); ++i) {

            QGraphicsEllipseItem\* node = scene->addEllipse(

                positions[i].x(), positions[i].y(),

                radius, radius,

                QPen(Qt::black, 1.5),

                Qt::white

                );

            node->setZValue(2);

            nodeItems.append(node);

            QGraphicsTextItem\* text = scene->addText(vertexNames[i]);

            text->setDefaultTextColor(Qt::black);

            QRectF textRect = text->boundingRect();

            text->setPos(

                positions[i].x() + radius/2 - textRect.width()/2,

                positions[i].y() + radius/2 - textRect.height()/2

                );

            text->setZValue(3);

        }

    }

    void drawArrow(QGraphicsScene\* scene, const QLineF& line, int radius, bool isBidirectional) {

        const qreal arrowSize = 8;

        QPointF diff = line.p2() - line.p1();

        qreal length = qSqrt(diff.x()\*diff.x() + diff.y()\*diff.y());

        if(length < 1e-6) return;

        QPointF unit = diff / length;

        QPointF baseEnd = line.p2() - unit \* (radius / 2);

        QPolygonF arrowEnd;

        arrowEnd << baseEnd

                 << baseEnd + QPointF(-unit.x()\*arrowSize - unit.y()\*arrowSize,

                                      -unit.y()\*arrowSize + unit.x()\*arrowSize)

                 << baseEnd + QPointF(-unit.x()\*arrowSize + unit.y()\*arrowSize,

                                      -unit.y()\*arrowSize - unit.x()\*arrowSize);

        scene->addPolygon(arrowEnd, QPen(Qt::gray), QBrush(Qt::gray))->setZValue(1);

        if(isBidirectional) {

            QPointF baseStart = line.p1() + unit \* (radius / 2);

            QPolygonF arrowStart;

            arrowStart << baseStart

                       << baseStart + QPointF(unit.x()\*arrowSize - unit.y()\*arrowSize,

                                              unit.y()\*arrowSize + unit.x()\*arrowSize)

                       << baseStart + QPointF(unit.x()\*arrowSize + unit.y()\*arrowSize,

                                              unit.y()\*arrowSize - unit.x()\*arrowSize);

            scene->addPolygon(arrowStart, QPen(Qt::gray), QBrush(Qt::gray))->setZValue(1);

        }

    }

    void highlightVertex(int idx, QColor color) {

        if(idx >= 0 && idx < nodeItems.size())

            nodeItems[idx]->setBrush(color);

    }

    void highlightEdge(int from, int to, QColor color) {

        QList<QGraphicsItem\*> items = scene->items();

        for(QGraphicsItem\* item : items) {

            if(auto line = dynamic\_cast<QGraphicsLineItem\*>(item)) {

                QPointF p1 = line->line().p1();

                QPointF p2 = line->line().p2();

                QPointF fromPos = nodeItems[from]->rect().center() + nodeItems[from]->pos();

                QPointF toPos = nodeItems[to]->rect().center() + nodeItems[to]->pos();

                if(p1 == fromPos && p2 == toPos) {

                    line->setPen(QPen(color, 2));

                }

            }

        }

    }

    void resetEdgeColors() {

        QList<QGraphicsItem\*> items = scene->items();

        for(QGraphicsItem\* item : items) {

            if(auto line = dynamic\_cast<QGraphicsLineItem\*>(item)) {

                line->setPen(QPen(Qt::gray, 1));

            }

        }

    }

    void resetVertexColors() {

        for(auto& node : nodeItems) {

            node->setBrush(Qt::white);

        }

    }

    QGraphicsScene\* scene;

    QTableWidget\* matrixTable;

    QTextEdit\* logText;

    QVector<QVector<int>> adjacencyMatrix;

    QMap<int, QString> vertexNames;

    QVector<QGraphicsEllipseItem\*> nodeItems;

    QVector<QGraphicsTextItem\*> edgeTexts;

};

int main(int argc, char \*argv[]) {

    QApplication app(argc, argv);

    GraphWindow window;

    window.setWindowTitle("Graph lab");

    window.resize(1280, 720);

    window.show();

    return app.exec();

}

#include "main.moc"

1. **UML и результат**



