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 дисциплине**

**«Теория алгоритмов и структуры данных»**

**на тему**

**«Графы»**

**Вариант 7**

Выполнил студент гр. ИВТ-23-1б

Бушмаков Кирилл Дмитриевич

Проверил:

Доцент каф. ИТАС

Яруллин Денис Владимирович

(оценка) (подпись)

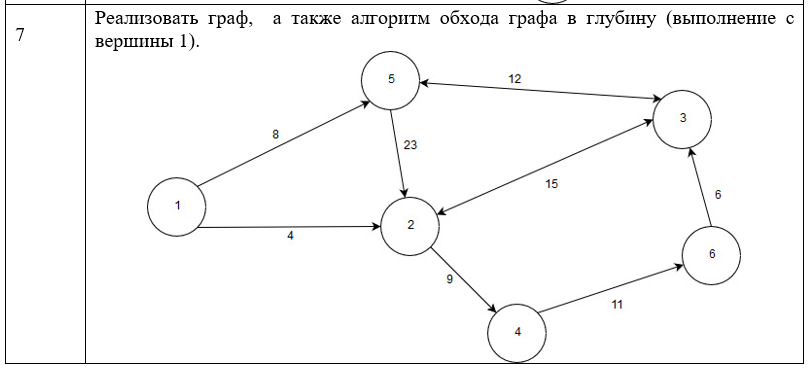
(дата)

г. Пермь, 2024

**Цель и задачи работы**

Целью данной работы является получение навыков работы с графами

**Вариант 7:**



**UML диаграмма**

На рисунке 1 изображена диаграмма класса

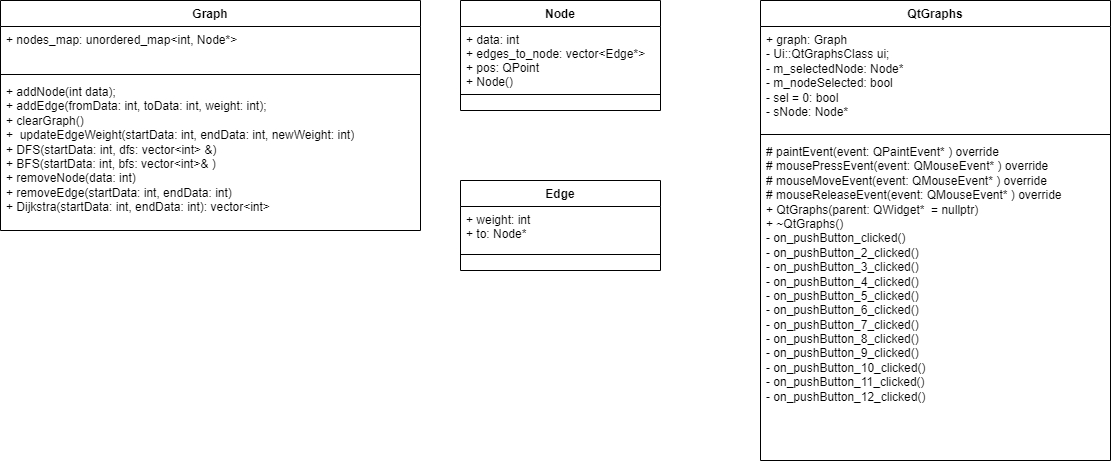


Рисунок 1

**Код программы**

**Файл QtGraphs.h**

#pragma once

#include <QWidget>

#include <QMouseEvent>

#include <ui\_QtGraphs.h>

#include <unordered\_map>

#include <vector>

#include <stack>

using namespace std;

class Edge;

class Node;

class Graph;

class Node //класс для узла графа

{

public:

int data;

vector<Edge\*> edges\_to\_node;

QPoint pos;

Node()

{

pos = QPoint(400, 200);

}

};

class Edge //класс для ребра графа

{

public:

int weight;

Node\* to;

};

class Graph

{

public:

unordered\_map<int, Node\*> nodes\_map;

void addNode(int data);//добавление узла с новым значением

void addEdge(int fromData, int toData, int weight);//добавление ребра с весом, от заданного узла к другому заданному узлу

void clearGraph();//очитка графа

void updateEdgeWeight(int startData, int endData, int newWeight);//редактирование веса ребра, при отсутсвиии ребра ничего не проиходит

void DFS(int startData, vector<int> &dfs);//обход в глубину, возвращает вектор в которм элементы идут в собраном порядке

void BFS(int startData, vector<int>& bfs);//обход в ширину, возвращает вектор в которм элементы идут в собраном порядке

void removeNode(int data);//удлаение узла по заданному значению

void removeEdge(int startData, int endData);//удаление ребра от заданного узла до заданного узла

vector<int> Dijkstra(int startData, int endData);//алгоритм Дейкстры ищущий кратчайший путь от узла к узлу, возвращает вектор который является путем, вектор пуст -> пути нет

};

class QtGraphs : public QMainWindow

{

Q\_OBJECT

public:

QtGraphs(QWidget\* parent = nullptr);

~QtGraphs();

Graph graph;

protected:

void paintEvent(QPaintEvent\* event) override;//переопределние отрисовки

void mousePressEvent(QMouseEvent\* event) override;//переопределние событий мыши для возможности перемещать узлы графа

void mouseMoveEvent(QMouseEvent\* event) override;

void mouseReleaseEvent(QMouseEvent\* event) override;//переопределние событий мыши для возможности перемещать узлы графа, конец

private:

Ui::QtGraphsClass ui;

Node\* m\_selectedNode;

bool m\_nodeSelected;

bool sel = 0;

Node\* sNode;

void on\_pushButton\_clicked();//функции выполняемые при нажатии на соответствующие кнопки, начало

void on\_pushButton\_2\_clicked();

void on\_pushButton\_3\_clicked();

void on\_pushButton\_4\_clicked();

void on\_pushButton\_5\_clicked();

void on\_pushButton\_6\_clicked();

void on\_pushButton\_7\_clicked();

void on\_pushButton\_8\_clicked();

void on\_pushButton\_9\_clicked();

void on\_pushButton\_10\_clicked();

void on\_pushButton\_11\_clicked();

void on\_pushButton\_12\_clicked();//функции выполняемые при нажатии на соответствующие кнопки, конец

};

**Файл QtGraph.cpp**

#include "QtGraphs.h"

#include <QPainter>

#include <vector>

#include <QLineEdit>

#include <QPushButton>

#include <cmath>

#include <unordered\_set>

#include <chrono>

#include <thread>

#include <QTimer>

#include <queue>

#include <limits>

#include <iostream>

const double M\_PI = 3.1415;

void Graph::addNode(int data) {

if (nodes\_map.find(data) == nodes\_map.end()) {

Node\* newNode = new Node;

newNode->data = data;

nodes\_map[data] = newNode;

}

else {

}

}

void Graph::addEdge(int fromData, int toData, int weight) {

for (Edge\* edge : nodes\_map[fromData]->edges\_to\_node) {

if (edge->to == nodes\_map[toData]) {

return;

}

}

Edge\* newEdge = new Edge();

newEdge->to = nodes\_map[toData];

newEdge->weight = weight;

nodes\_map[fromData]->edges\_to\_node.push\_back(newEdge);

}

void Graph::clearGraph()

{

// Удаление всех узлов

for (auto& pair : nodes\_map)

{

Node\* node = pair.second;

delete node;

}

// Очистка хэш-таблицы

nodes\_map.clear();

}

void Graph::updateEdgeWeight(int startData, int endData, int newWeight) {

if (nodes\_map.find(startData) == nodes\_map.end() || nodes\_map.find(endData) == nodes\_map.end()) {

// Один из узлов не существует

return;

}

Node\* startNode = nodes\_map[startData];

Node\* endNode = nodes\_map[endData];

for (Edge\* edge : startNode->edges\_to\_node) {

if (edge->to == endNode) {

edge->weight = newWeight;

return;

}

}

// Ребро не найдено

}

void Graph::DFS(int startData, vector<int>& dfs)

{

stack<Node\*> nodeStack;

nodeStack.push(nodes\_map[startData]);

unordered\_set<int> visited;

visited.insert(startData);

while (!nodeStack.empty())

{

Node\* currentNode = nodeStack.top();

nodeStack.pop();

dfs.push\_back(currentNode->data);

for (Edge\* edge : currentNode->edges\_to\_node)

{

if (visited.find(edge->to->data) == visited.end())

{

nodeStack.push(edge->to);

visited.insert(edge->to->data);

}

}

}

for (auto const& pair : nodes\_map)

{

if (visited.find(pair.first) == visited.end())

{

dfs.push\_back(pair.first);

visited.insert(pair.first);

}

}

}

void Graph::BFS(int startData, vector<int>& bfs)

{

queue<Node\*> q;

unordered\_map<int, bool> visited;

Node\* startNode = nodes\_map[startData];

q.push(startNode);

visited[startData] = true;

while (!q.empty())

{

Node\* currentNode = q.front();

q.pop();

bfs.push\_back(currentNode->data);

for (Edge\* edge : currentNode->edges\_to\_node)

{

Node\* neighborNode = edge->to;

if (!visited[neighborNode->data])

{

visited[neighborNode->data] = true;

q.push(neighborNode);

}

}

}

// Проверяем не посещенные узлы и запускаем для них обход в ширину

for (const auto& pair : nodes\_map)

{

Node\* node = pair.second;

if (!visited[node->data])

{

q.push(node);

visited[node->data] = true;

while (!q.empty())

{

Node\* currentNode = q.front();

q.pop();

bfs.push\_back(currentNode->data);

for (Edge\* edge : currentNode->edges\_to\_node)

{

Node\* neighborNode = edge->to;

if (!visited[neighborNode->data])

{

visited[neighborNode->data] = true;

q.push(neighborNode);

}

}

}

}

}

}

void Graph::removeNode(int data)

{

// Удаление всех ребер, ведущих к узлу с заданным значением

for (auto& pair : nodes\_map)

{

Node\* node = pair.second;

vector<Edge\*> edges\_to\_remove;

for (Edge\* edge : node->edges\_to\_node)

{

if (edge->to->data == data)

{

edges\_to\_remove.push\_back(edge);

}

}

for (Edge\* edge : edges\_to\_remove)

{

auto it = find(node->edges\_to\_node.begin(), node->edges\_to\_node.end(), edge);

if (it != node->edges\_to\_node.end())

{

node->edges\_to\_node.erase(it);

delete edge; // Освобождение памяти

}

}

}

// Удаление самого узла из nodes\_map

auto it = nodes\_map.find(data);

if (it != nodes\_map.end())

{

delete it->second; // Освобождение памяти

nodes\_map.erase(it);

}

}

void Graph::removeEdge(int startData, int endData)

{

// Проверка наличия узлов с заданными значениями в графе

auto startNodeIt = nodes\_map.find(startData);

auto endNodeIt = nodes\_map.find(endData);

if (startNodeIt == nodes\_map.end() || endNodeIt == nodes\_map.end())

{

return;

}

Node\* startNode = startNodeIt->second;

Node\* endNode = endNodeIt->second;

// Поиск и удаление ребра между узлами

Edge\* edgeToRemove = nullptr;

for (Edge\* edge : startNode->edges\_to\_node)

{

if (edge->to->data == endData)

{

edgeToRemove = edge;

break;

}

}

if (edgeToRemove)

{

auto it = find(startNode->edges\_to\_node.begin(), startNode->edges\_to\_node.end(), edgeToRemove);

if (it != startNode->edges\_to\_node.end())

{

startNode->edges\_to\_node.erase(it);

delete edgeToRemove; // Освобождение памяти

}

}

else

{

}

}

vector<int> Graph::Dijkstra(int startData, int endData)

{

unordered\_map<int, int> dist;

unordered\_map<int, int> prev;

vector<int> result;

for (auto& pair : nodes\_map)

{

dist[pair.first] = INT\_MAX;

prev[pair.first] = -1;

}

dist[startData] = 0;

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

pq.push({ 0, startData });

while (!pq.empty())

{

int u = pq.top().second;

pq.pop();

if (u == endData)

break;

for (Edge\* edge : nodes\_map[u]->edges\_to\_node)

{

int v = edge->to->data;

int alt = dist[u] + edge->weight;

if (alt < dist[v])

{

dist[v] = alt;

prev[v] = u;

pq.push({ alt, v });

}

}

}

// Восстановление пути

for (int at = endData; at != -1; at = prev[at])

{

result.push\_back(at);

}

reverse(result.begin(), result.end());

if (result[0] == endData) { result.pop\_back(); }

return result;

}

QtGraphs::QtGraphs(QWidget\* parent)

: QMainWindow(parent)

{

ui.setupUi(this);

//подключение кнопок

connect(ui.pushButton, &QPushButton::clicked, this, &QtGraphs::on\_pushButton\_clicked);

connect(ui.pushButton\_2, &QPushButton::clicked, this, &QtGraphs::on\_pushButton\_2\_clicked);

connect(ui.pushButton\_4, &QPushButton::clicked, this, &QtGraphs::on\_pushButton\_3\_clicked);

connect(ui.pushButton\_3, &QPushButton::clicked, this, &QtGraphs::on\_pushButton\_4\_clicked);

connect(ui.pushButton\_5, &QPushButton::clicked, this, &QtGraphs::on\_pushButton\_5\_clicked);

connect(ui.pushButton\_6, &QPushButton::clicked, this, &QtGraphs::on\_pushButton\_6\_clicked);

connect(ui.pushButton\_7, &QPushButton::clicked, this, &QtGraphs::on\_pushButton\_7\_clicked);

connect(ui.pushButton\_8, &QPushButton::clicked, this, &QtGraphs::on\_pushButton\_8\_clicked);

connect(ui.pushButton\_9, &QPushButton::clicked, this, &QtGraphs::on\_pushButton\_9\_clicked);

connect(ui.pushButton\_10, &QPushButton::clicked, this, &QtGraphs::on\_pushButton\_10\_clicked);

connect(ui.pushButton\_11, &QPushButton::clicked, this, &QtGraphs::on\_pushButton\_11\_clicked);

connect(ui.pushButton\_12, &QPushButton::clicked, this, &QtGraphs::on\_pushButton\_12\_clicked);

}

QtGraphs::~QtGraphs()

{

}

void printAdjacencyMatrix(const Graph& graph) {

// Подсчитываем количество вершин

int numVertices = graph.nodes\_map.size();

// Создаем статический двумерный массив для матрицы смежности и заполняем его нулями

int adjacencyMatrix[100][100] = { 0 }; // Предполагаем, что граф не будет содержать более 100 вершин

// Заполняем матрицу смежности

for (const auto& pair : graph.nodes\_map) {

Node\* node = pair.second;

for (Edge\* edge : node->edges\_to\_node) {

adjacencyMatrix[node->data-1][edge->to->data-1] = edge->weight;

}

}

// Выводим матрицу смежности

std::cout << "Adjacency Matrix:" << std::endl;

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

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

std::cout << adjacencyMatrix[i][j] << " ";

}

std::cout << std::endl;

}

}

void QtGraphs::paintEvent(QPaintEvent\* event) {

QPainter painter(this);

QFont font = painter.font();

font.setPointSize(16);

painter.setFont(font);

for (const auto& pair : graph.nodes\_map) {

Node\* node = pair.second;

for (Edge\* edge : node->edges\_to\_node) {

QPoint pos\_f;

QPoint pos\_t;

int d = 20 \* sin(atan(1));

double angles = atan2(-(edge->to->pos.y() - node->pos.y()), (edge->to->pos.x()-node->pos.x()));

pos\_f = QPoint(node->pos.x()+20\*cos(angles), node->pos.y() - 20\*sin(angles));

pos\_t = QPoint(edge->to->pos.x()- 20 \* cos(angles), edge->to->pos.y() + 20\*sin(angles));

painter.drawLine(pos\_f, pos\_t);

int x\_t = pos\_f.x() + 4 \* (pos\_t.x() - pos\_f.x()) / 5;

int y\_t = pos\_f.y() - 4 \* (pos\_f.y() - pos\_t.y()) / 5;

painter.drawText(x\_t-10, y\_t+10, QString::number(edge->weight));

QLine line(pos\_f, pos\_t);

double angle = atan2(-line.dy(), line.dx())-M\_PI/2;

double arrowSize = 15;

double arrowLength = 20;

QPointF arrowP1 = pos\_t + QPointF(sin(angle - M\_PI / 3) \* arrowSize, cos(angle - M\_PI / 3) \* arrowSize);

QPointF arrowP2 = pos\_t + QPointF(sin(angle + M\_PI / 3) \* arrowSize, cos(angle + M\_PI / 3) \* arrowSize);

QPolygonF arrowHead;

arrowHead << pos\_t << arrowP1 << arrowP2;

painter.drawPolygon(arrowHead);

}

}

for (const auto& pair : graph.nodes\_map) {

Node\* node = pair.second;

painter.drawEllipse(node->pos, 20, 20);

painter.drawText(node->pos.x() - 9, node->pos.y() + 8, QString::number(node->data));

}

if (sel) {

painter.drawEllipse(100,100, 40, 40);

painter.setBrush(Qt::green);

painter.drawEllipse(sNode->pos, 20, 20);

painter.drawText(sNode->pos.x() - 9, sNode->pos.y() + 8, QString::number(sNode->data));

}

}

void QtGraphs::mousePressEvent(QMouseEvent\* event)

{

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

m\_nodeSelected = false; // Сброс флага выбранного узла

for (const auto& pair : graph.nodes\_map) {

Node\* node = pair.second;

if ((event->pos() - node->pos).manhattanLength() < 30) {

m\_selectedNode = node;

m\_nodeSelected = true;

break;

}

}

update();

}

}

void QtGraphs::mouseMoveEvent(QMouseEvent\* event)

{

if (m\_nodeSelected && m\_selectedNode) {

m\_selectedNode->pos = event->pos();

update();

}

}

void QtGraphs::mouseReleaseEvent(QMouseEvent\* event)

{

if (event->button() == Qt::LeftButton && m\_nodeSelected) {

m\_nodeSelected = false;

m\_selectedNode = nullptr;

update();

}

}

void QtGraphs::on\_pushButton\_clicked() {

QString text = ui.lineEdit->text();

if (text.isEmpty()) {

return;

}

int nodeValue = text.toInt();

graph.addNode(nodeValue);

ui.lineEdit->clear();

update();

}

void QtGraphs::on\_pushButton\_2\_clicked() {

if (ui.lineEdit\_2->text().isEmpty() or ui.lineEdit\_3->text().isEmpty() or ui.lineEdit\_4->text().isEmpty()) {

return;

}

int fromNode = ui.lineEdit\_2->text().toInt();

int toNode = ui.lineEdit\_3->text().toInt();

int weight = ui.lineEdit\_4->text().toInt();

if (graph.nodes\_map.find(fromNode) != graph.nodes\_map.end() && graph.nodes\_map.find(toNode) != graph.nodes\_map.end()) {

graph.addEdge(fromNode, toNode, weight);

ui.lineEdit\_2->clear();

ui.lineEdit\_4->clear();

ui.lineEdit\_3->clear();

update();

}

else {

}

}

void QtGraphs::on\_pushButton\_3\_clicked()

{

if (ui.lineEdit\_5->text().isEmpty()) {

return;

}

int del = ui.lineEdit\_5->text().toInt();

graph.removeNode(del);

ui.lineEdit\_5->clear();

update();

}

void QtGraphs::on\_pushButton\_4\_clicked()

{

if (ui.lineEdit\_7->text().isEmpty() or ui.lineEdit\_6->text().isEmpty()) {

return;

}

int s = ui.lineEdit\_7->text().toInt();

int f = ui.lineEdit\_6->text().toInt();

graph.removeEdge(s, f);

ui.lineEdit\_7->clear();

ui.lineEdit\_6->clear();

update();

}

void QtGraphs::on\_pushButton\_5\_clicked()

{

ui.textBrowser->clear();

if (ui.lineEdit\_8->text().isEmpty()) {

return;

}

vector<int> dfsv;

int s = ui.lineEdit\_8->text().toInt();

if (graph.nodes\_map.find(s) != graph.nodes\_map.end()) {

graph.DFS(s, dfsv);

QString resultString;

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

resultString.append(QString::number(dfsv[i]));

if (i < dfsv.size() - 1) {

resultString.append(", ");

}

}

//ui.textBrowser->setText(resultString);

static int idx = 0;

QTimer\* timer = new QTimer(this);

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

if (dfsv.size() != 0 and idx < dfsv.size()) {

Node\* nod = graph.nodes\_map[dfsv[idx]];

sNode = nod;

sel = 1;

update();

idx++;

}

else {

ui.textBrowser->setText(resultString);

timer->stop();

timer->deleteLater();

sel = 0;

ui.lineEdit\_8->clear();

update();

idx = 0;

}

});

timer->start(500);

}

}

void QtGraphs::on\_pushButton\_6\_clicked()

{

ui.textBrowser\_2->clear();

if (ui.lineEdit\_9->text().isEmpty() or ui.lineEdit\_9->text().isEmpty()) {

return;

}

int s = ui.lineEdit\_9->text().toInt();

int f = ui.lineEdit\_10->text().toInt();

if (graph.nodes\_map.find(s) != graph.nodes\_map.end() and graph.nodes\_map.find(f) != graph.nodes\_map.end()) {

vector<int> di = graph.Dijkstra(s,f);

QString resultString;

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

resultString.append(QString::number(di[i]));

if (i < di.size() - 1) {

resultString.append("->");

}

}

static int idx = 0;

QTimer\* timer = new QTimer(this);

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

if (di.size() != 0 and idx < di.size()) {

Node\* nod = graph.nodes\_map[di[idx]];

sNode = nod;

sel = 1;

update();

idx++;

}

else {

ui.textBrowser\_2->setText(resultString);

timer->stop();

timer->deleteLater();

sel = 0;

ui.lineEdit\_9->clear();

ui.lineEdit\_10->clear();

update();

idx = 0;

}

});

timer->start(500);

}

}

void QtGraphs::on\_pushButton\_7\_clicked()

{

if (ui.lineEdit\_11->text().isEmpty()) {

return;

}

vector<int> dfsv;

int s = ui.lineEdit\_11->text().toInt();

if (graph.nodes\_map.find(s) != graph.nodes\_map.end()) {

graph.BFS(s, dfsv);

QString resultString;

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

resultString.append(QString::number(dfsv[i]));

if (i < dfsv.size() - 1) {

resultString.append(", ");

}

}

//ui.textBrowser\_2->setText(resultString);

static int idx = 0;

QTimer\* timer = new QTimer(this);

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

if (dfsv.size() != 0 and idx < dfsv.size()) {

Node\* nod = graph.nodes\_map[dfsv[idx]];

sNode = nod;

sel = 1;

update();

idx++;

}

else {

ui.textBrowser\_3->setText(resultString);

timer->stop();

timer->deleteLater();

sel = 0;

ui.lineEdit\_11->clear();

update();

idx = 0;

}

});

timer->start(500);

}

}

void QtGraphs::on\_pushButton\_8\_clicked()

{

graph.addNode(1);

graph.addNode(2);

graph.addNode(3);

graph.addNode(4);

graph.addNode(5);

graph.addNode(6);

graph.addEdge(1, 2, 4);

graph.addEdge(1,5,8);

graph.addEdge(5,3,12);

graph.addEdge(3,5,12);

graph.addEdge(5,2,23);

graph.addEdge(2,4,9);

graph.addEdge(4,6,11);

graph.addEdge(2,3,15);

graph.addEdge(3,2,15);

graph.addEdge(6,3,6);

update();

}

void QtGraphs::on\_pushButton\_9\_clicked()

{

graph.addNode(1);

graph.addNode(2);

graph.addNode(3);

graph.addNode(4);

graph.addNode(5);

graph.addNode(6);

graph.addNode(7);

graph.addNode(8);

graph.addNode(9);

graph.addNode(10);

graph.addEdge(1,2,8);

graph.addEdge(2,3,9);

graph.addEdge(3,2,6);

graph.addEdge(1,4,9);

graph.addEdge(4,5,3);

graph.addEdge(4,3,12);

graph.addEdge(1,6,10);

graph.addEdge(6,7,16);

graph.addEdge(7,5,16);

graph.addEdge(8,9,1);

graph.addEdge(9,8,2);

graph.addEdge(5,7,89);

graph.addEdge(4,1,5);

graph.addNode(11);

graph.addEdge(8, 11, 69);

graph.addEdge(11, 9, 44);

update();

}

void QtGraphs::on\_pushButton\_10\_clicked()

{

if (ui.lineEdit\_12->text().isEmpty() or ui.lineEdit\_13->text().isEmpty() or ui.lineEdit\_14->text().isEmpty()) {

return;

}

int s = ui.lineEdit\_12->text().toInt();

int t = ui.lineEdit\_14->text().toInt();

int w = ui.lineEdit\_13->text().toInt();

graph.updateEdgeWeight(s, t, w);

ui.lineEdit\_12->text().clear();

ui.lineEdit\_14->text().clear();

ui.lineEdit\_13->text().clear();

update();

}

void QtGraphs::on\_pushButton\_11\_clicked()

{

graph.clearGraph();

update();

}

void QtGraphs::on\_pushButton\_12\_clicked()

{

printAdjacencyMatrix(graph);

}

**Файл main.cpp**

#include "QtGraphs.h"

#include <QtWidgets/QApplication>

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

{

QApplication a(argc, argv);

QtGraphs w;

w.show();

return a.exec();

}

**Демонстрация работы программы:**

https://youtu.be/BNtogfsi3uM