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 2

Выполнил студент гр. ИВТ-23-1б
Попонин Михаил Александрович

Проверил:
Доцент каф. ИТАС
Полякова Ольга Андреевна

(оценка) (подпись)

(дата)

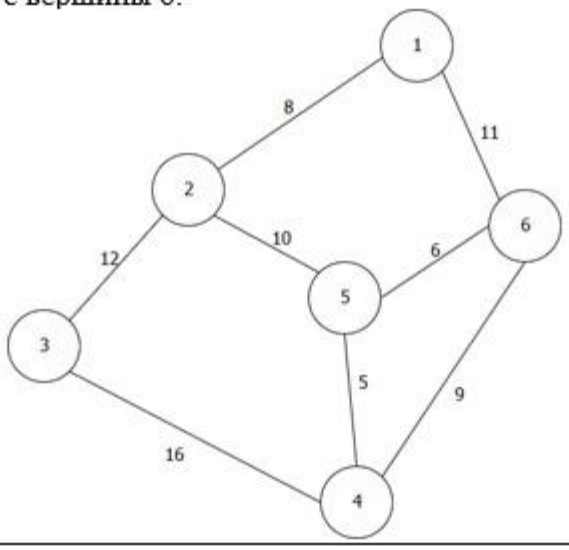
г. Пермь, 2024

Цель и задачи работы

Целью данной работы является получение навыков работы с графами

Вариант 2:

Реализовать граф, а также алгоритм Дейкстры, выполнив все необходимые действия.
Выполнение начать с вершины 6.



UML диаграмма

На рисунке 1 изображена диаграмма класса

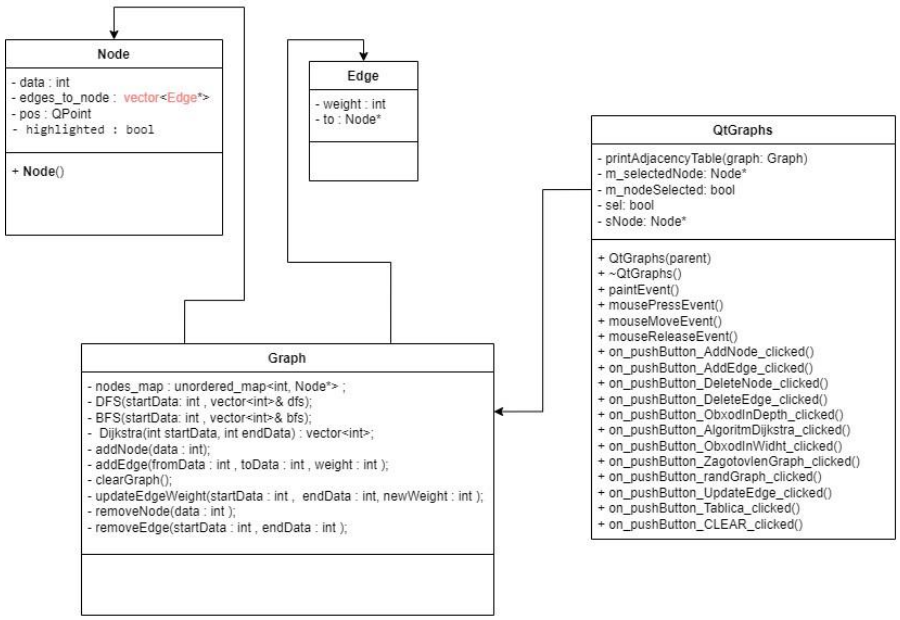


Рисунок 1

Код программы

Таблица 1 – файл QtGraphs.h

```
#pragma once
#include <QWidget>
#include <QMouseEvent>
#include <ui_QtGraphs.h>
#include <unordered_map>
#include <unordered_set>
#include <vector>
#include <stack>
#include <iostream>
using namespace std;
class Edge;
class Node;
class Graph;
class Node
{
public:
    int data;
    vector<Edge*> edges_to_node;
    QPoint pos;
    bool highlighted;
    Node()
    {
        pos = QPoint(600 + (rand()%600 - 300), 300 + (rand()%600 - 300));
    }
};
class Edge
{
public:
    int weight;
    Node* to;
};
class Graph
{
public:
    unordered_map<int, Node*> nodes_map;

    void DFS(int startData, vector<int>& dfs);
    void BFS(int startData, vector<int>& bfs);
    vector<int> Dijkstra(int startData, int endData);
    void addNode(int data);
    void addEdge(int fromData, int toData, int weight);
    void clearGraph();
    void updateEdgeWeight(int startData, int endData, int newWeight);
    void removeNode(int data);
```

```
    void removeEdge(int startData, int endData);
};
class QtGraphs : public QMainWindow
{
    Q_OBJECT
public:
    QtGraphs(QWidget* parent = nullptr);
    ~QtGraphs();
    Graph graph;
protected:
    void paintEvent(QPaintEvent* event) override;
    void mousePressEvent(QMouseEvent* event) override;
    void mouseMoveEvent(QMouseEvent* event) override;
    void mouseReleaseEvent(QMouseEvent* event) override;
private:
    Ui::QtGraphs ui;
    Node* m_selectedNode;
    bool m_nodeSelected;
    void printAdjacencyTable(const Graph& graph)
    {
        for (const auto& pair : graph.nodes_map)
        {
            int nodeData = pair.first;
            Node* node = pair.second;
            std::cout << "Vertex " << nodeData << ": ";
            std::unordered_set<int> printedNodes;
            for (Edge* edge : node->edges_to_node)
            {
                if (printedNodes.find(edge->to->data) == printedNodes.end())
                {
                    std::cout << "[" << edge->to->data << ", " << edge->weight << "] ";
                    printedNodes.insert(edge->to->data);
                }
            }
            std::cout << std::endl;
        }
    }
    bool sel = 0;
    Node* sNode;
    void on_pushButton_AddNode_clicked();
    void on_pushButton_AddEdge_clicked();
    void on_pushButton_DeleteNode_clicked();
    void on_pushButton_DeleteEdge_clicked();
    void on_pushButton_ObxodInDepth_clicked();
    void on_pushButton_AlgoritmDijkstra_clicked();
    void on_pushButton_ObxodInWidht_clicked();
    void on_pushButton_ZagotovlenGraph_clicked();
    void on_pushButton_randGraph_clicked();
```

```
    void on_pushButton_UpdateEdge_clicked();
    void on_pushButton_Tablica_clicked();
    void on_pushButton_CLEAR_clicked();
};
```

Таблица 2 – файл main.cpp

```
#include "QtGraphs.h"
#include <QApplication>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QtGraphs w;
    w.show();
    return a.exec();
}
```

Таблица 3 – файл QtGraphs.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>
void Graph::addNode(int data)
{
    if (nodes_map.find(data) == nodes_map.end())
    {
        Node* newNode = new Node;
        newNode->data = data;
        nodes_map[data] = newNode;
    }
}
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()
{
    for (auto& pair : nodes_map)
    {
        Node* node = pair.second;
        delete node;
    }
    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())
    {
        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;
        }
    }
}
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);
            }
        }
    }
    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)
{
   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;
         }
      }
   }
   auto it = nodes_map.find(data);
   if (it != nodes_map.end())
   {
      delete it->second;
      nodes_map.erase(it);
   }
}
void Graph::removeEdge(int startData, int endData)
{
   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;
    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;
        }
    }
}
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 });
            }
        }
    }
    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);
    connect(ui.pushButton_AddNode, &QPushButton::clicked, this,
&QtGraphs::on_pushButton_AddNode_clicked);
    connect(ui.pushButton_AddEdge, &QPushButton::clicked, this,
&QtGraphs::on_pushButton_AddEdge_clicked);
    connect(ui.pushButton_DeleteNode, &QPushButton::clicked, this,
&QtGraphs::on_pushButton_DeleteNode_clicked);
    connect(ui.pushButton_DeleteEdge, &QPushButton::clicked, this,
&QtGraphs::on_pushButton_DeleteEdge_clicked);
    connect(ui.pushButton_ObxodInDepth, &QPushButton::clicked, this,
&QtGraphs::on_pushButton_ObxodInDepth_clicked);
    connect(ui.pushButton_AlgoritmDijkstra, &QPushButton::clicked, this,
&QtGraphs::on_pushButton_AlgoritmDijkstra_clicked);
    connect(ui.pushButton_ObxodInWidht, &QPushButton::clicked, this,
&QtGraphs::on_pushButton_ObxodInWidht_clicked);
    connect(ui.pushButton_ZagotovlenGraph, &QPushButton::clicked, this,
&QtGraphs::on_pushButton_ZagotovlenGraph_clicked);
    connect(ui.pushButton_randGraph, &QPushButton::clicked, this,
&QtGraphs::on_pushButton_randGraph_clicked);
    connect(ui.pushButton_UpdateEdge, &QPushButton::clicked, this,
&QtGraphs::on_pushButton_UpdateEdge_clicked);
    connect(ui.pushButton_CLEAR, &QPushButton::clicked, this,
&QtGraphs::on_pushButton_CLEAR_clicked);
    connect(ui.pushButton_Tablica, &QPushButton::clicked, this,
&QtGraphs::on_pushButton_Tablica_clicked);
}
QtGraphs::~QtGraphs()
{
```

```
}
void QtGraphs::paintEvent(QPaintEvent* event)
{
   QPainter painter(this);
   QFont font = painter.font();
   font.setPointSize(16);
   painter.setFont(font);
   painter.setPen(QPen(Qt::black, 2));
   for (const auto& pair : graph.nodes_map) {
      Node* node = pair.second;
      for (Edge* edge : node->edges_to_node) {
         QPoint pos_f;
         QPoint pos_t;
         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 = 20;
         QPointF arrowP1 = pos_t + QPointF(sin(angle - M_PI / 12) * arrowSize, cos(angle -
M_PI / 12) * arrowSize);
         QPointF arrowP2 = pos_t + QPointF(sin(angle + M_PI / 12) * arrowSize, cos(angle +
M_PI / 12) * arrowSize);
         QPolygonF arrowHead;
         arrowHead << pos_t << arrowP1 << arrowP2;
         QPainterPath path;
         path.moveTo(pos_t);
         path.lineTo(arrowP1);
         path.lineTo(arrowP2);
         painter.fillPath(path, Qt::magenta);
         painter.drawPolygon(arrowHead);
      }
   }
   painter.setBrush(Qt::NoBrush);
   painter.setPen(QPen(Qt::black, 2));
   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::magenta);
        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;
        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_AddNode_clicked()
{
    QString text = ui.lineEdit_addDelNodeValue->text();
    if (text.isEmpty())
    {
        return;
    }
    int nodeValue = text.toInt();
```

```
    graph.addNode(nodeValue);
    ui.lineEdit_addDelNodeValue->clear();
    update();
    ui.statusbar->showMessage("Вершина добавлена!");
}
void QtGraphs::on_pushButton_AddEdge_clicked() {
    if (ui.LineEdit_FirstNode->text().isEmpty() or ui.LineEdit_SecondNode->text().isEmpty() or
ui.lineEdit_Weight->text().isEmpty()) {
        return;
    }
    int fromNode = ui.LineEdit_FirstNode->text().toInt();
    int toNode = ui.LineEdit_SecondNode->text().toInt();
    int weight = ui.lineEdit_Weight->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_FirstNode->clear();
        ui.lineEdit_Weight->clear();
        ui.LineEdit_SecondNode->clear();
        update();
        ui.statusbar->showMessage("Грань добавлена!");
    }
}
void QtGraphs::on_pushButton_DeleteNode_clicked()
{
    if (ui.lineEdit_addDelNodeValue->text().isEmpty())
    {
        return;
    }
    int del = ui.lineEdit_addDelNodeValue->text().toInt();
    graph.removeNode(del);
    ui.lineEdit_addDelNodeValue->clear();
    update();
    ui.statusbar->showMessage("Вершина удалена!");
}
void QtGraphs::on_pushButton_DeleteEdge_clicked()
{
    if (ui.LineEdit_FirstNode->text().isEmpty() or ui.LineEdit_SecondNode->text().isEmpty()) {
        return;
    }
    int s = ui.LineEdit_FirstNode->text().toInt();
    int f = ui.LineEdit_SecondNode->text().toInt();
    graph.removeEdge(s, f);
    ui.LineEdit_FirstNode->clear();
    ui.LineEdit_SecondNode->clear();
    update();
    ui.statusbar->showMessage("Грань удалена!");
```

```
}
void QtGraphs::on_pushButton_ObxodInDepth_clicked()
{
   ui.textBrowser->clear();
   if (ui.lineEditAlgoritmObxodaInDepth->text().isEmpty()) {
      return;
   }
   vector<int> dfsv;
   int s = ui.lineEditAlgoritmObxodaInDepth->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(", ");
         }
      }
      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.lineEditAlgoritmObxodaInDepth->clear();
            update();
            idx = 0;
         }
         ui.statusbar->showMessage("Алгоритм обхода в глубину выполнен!");
      });
      timer->start(666);
   }
}
void QtGraphs::on_pushButton_AlgoritmDijkstra_clicked()
{
```

```
    ui.textBrowser->clear();
    if (ui.lineEditAlgoritmDijkstraFirst->text().isEmpty() or ui.lineEditAlgoritmDijkstraSecond-
>text().isEmpty())
    {
        return;
    }
    int s = ui.lineEditAlgoritmDijkstraFirst->text().toInt();
    int f = ui.lineEditAlgoritmDijkstraSecond->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->setText(resultString);
                timer->stop();
                timer->deleteLater();
                sel = 0;
                ui.lineEditAlgoritmDijkstraFirst->clear();
                ui.lineEditAlgoritmDijkstraSecond->clear();
                update();
                idx = 0;
            }
            ui.statusbar->showMessage("Алгоритм Дейкстры выполнен!");
        });
        timer->start(500);
    }
```

```
}
void QtGraphs::on_pushButton_ObxodInWidht_clicked()
{
   if (ui.lineEdit_ObxodInWidht->text().isEmpty())
   {
      return;
   }
   vector<int> dfsv;
   int s = ui.lineEdit_ObxodInWidht->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(", ");
         }
      }
      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_ObxodInWidht->clear();
            update();
            idx = 0;
         }
         ui.statusbar->showMessage("Алгоритм обхода в ширину выполнен!");
      });
      timer->start(500);
   }
}
```

```
void QtGraphs::on_pushButton_ZagotovlenGraph_clicked()
{
   graph.addNode(1);
   graph.addNode(2);
   graph.addNode(3);
   graph.addNode(4);
   graph.addNode(5);
   graph.addNode(6);

   graph.addEdge(1, 2, 8);
   graph.addEdge(1, 6, 11);

   graph.addEdge(6, 5, 6);
   graph.addEdge(6, 4, 9);
   graph.addEdge(6, 1, 11);

   graph.addEdge(4, 5, 5);
   graph.addEdge(4, 3, 16);
   graph.addEdge(4, 6, 9);

   graph.addEdge(3, 2, 12);
   graph.addEdge(3, 4, 16);


   graph.addEdge(2, 1, 8);
   graph.addEdge(2, 3, 12);
   graph.addEdge(2, 5, 10);

   graph.addEdge(5, 2, 10);
   graph.addEdge(5, 4, 5);
   graph.addEdge(5, 6, 6);

   update();
   ui.statusbar->showMessage("Загатовленный граф призван!");
}
void QtGraphs::on_pushButton_randGraph_clicked()
{
   for (int i = 0; i < 10; i++)
   {
      int c = rand() % 10;
      int b = rand() % 10;
      graph.addNode(c);
      graph.addNode(b);
      int chance = rand() % 5;
      if (!(chance == 0))
      {
         int m = rand() % 10;
         graph.addEdge(c,b,m);
```

```
            if(chance == 1)
            {
                graph.addEdge(c,b,m);
            }
        }
    }
    update();
    ui.statusbar->showMessage("Случайный граф призван!");
}
void QtGraphs::on_pushButton_UpdateEdge_clicked()
{
    if (ui.LineEdit_FirstNode->text().isEmpty() or ui.lineEdit_Weight->text().isEmpty() or
ui.LineEdit_SecondNode->text().isEmpty())
    {
        return;
    }
    int s = ui.LineEdit_FirstNode->text().toInt();
    int t = ui.LineEdit_SecondNode->text().toInt();
    int w = ui.lineEdit_Weight->text().toInt();
    graph.updateEdgeWeight(s, t, w);
    ui.LineEdit_FirstNode->clear();
    ui.lineEdit_Weight->clear();
    ui.LineEdit_SecondNode->clear();
    update();
    ui.statusbar->showMessage("Грань обновлена!");
}
void QtGraphs::on_pushButton_CLEAR_clicked()
{
    graph.clearGraph();
    update();
}
void QtGraphs::on_pushButton_Tablica_clicked()
{
    printAdjacencyTable(graph);
}
```

Демонстрация работы программы:

https://youtu.be/S6xSP47UG08?si=6zBMYStq-RsDpEmu

Скриншоты работы программы: