Министерство науки и высшего образования Российской Федерации Федеральное государственное автономное учреждение высшего образования «Пермский национальный исследовательский политехнический университет»

ПНИПУ

**Лабораторная работа  
Графы**

Выполнил:   
студент группы ИВТ-23-2Б   
Чудинов Данил Николаевич

Проверила:   
доцент кафедры ИТАС   
О.А. Полякова

Пермь, 2024 г.

**«Графы»**

**Анализ задачи:**

Реализовать алгоритмы для собственного, придуманного самим автором, варианта двунаправленного графа, имеющего не менее 6 вершин.

Алгоритмы:

1. Обход в ширину.

2. Обход в глубину.

3. Алгоритм Дейкстры.

Необходимо реализовать функции для редактирования графа:

- Создание новой вершины.

- Удаление вершины.

- Добавление и удаление ребра.

- Редактирование весов ребер.

- Редактирование матрицы смежности (или инцидентности в зависимости от реализации).

- Реализовать вывод графа.

**Код на языке C++:**

#include <iostream>

#include <vector>

#include <queue>

#include <string>

using namespace std;

template <class T>

class Graph {

private:

vector<T> vetrexList;

vector<vector<int>> adjMatrix;

int size;

queue<T> VertsQueue;

vector<int> labelList;

public:

inline Graph<T>(const int& ksize) {//Конструктор графа, в параметр передаётся размер графа(то есть кол-во вершин)

this->size = ksize;

labelList.resize(ksize, 1000000);

this->adjMatrix = vector<vector<T>>(ksize, vector<T>(ksize));

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

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

this->adjMatrix[i][j] = 0;

};

};

}

inline bool isfull() { //Проверка графа на то, что он заполнен

return this->vetrexList.size() == this->size;

}

inline bool isEmpty() {//Проверка графа на то, что он пуст

return this->vetrexList.size() == 0;

}

inline void insertVetrex(const T& vert) {//Функция, которая добавляет вершину

if (this->isfull()) {

cout << "Невозможно добавить вершину." << endl;

return;

}

this->vetrexList.push\_back(vert);

}

inline int GetVertPos(const T& g) {//Получение индекса вершин

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

if (this->vetrexList[i] == g) {

return i;

}

}

return -1;

}

inline int GetAmountVerts() {//Получение количества вершин

return this->vetrexList.size();

}

inline int GetWeight(const T& g1, const T& g2) {//Получение веса между вершинами

if (this->isEmpty()) {

return 0;

}

int g1\_p = this->GetVertPos(g1);

int g2\_p = this->GetVertPos(g2);

if (g1\_p == -1 || g2\_p == -1) {

cout << "Одного из выбранных узлов в графе не существует!";

return 0;

}

return this->adjMatrix[g1\_p][g2\_p];

}

vector<T> GetNbrs(const T& vetrex) {//Получение вектора соседей

vector<T> nbrsList;

int pos = this->GetVertPos(vetrex);

if (pos != -1) {

for (int i = 0; i < this->vetrexList.size(); i++) {

if (this->adjMatrix[pos][i] != 0) {

nbrsList.push\_back(this->vetrexList[i]);

}

}

}

return nbrsList;

}

void InsertEdge(const T& vetrex1, const T& vetrex2, int weight = 1) {//Вставка ребра для неориентированного графа

if (GetVertPos(vetrex1) != (-1) && this->GetVertPos(vetrex2) != (-1)) {

int vertPos1 = GetVertPos(vetrex1);

int vertPos2 = GetVertPos(vetrex2);

if (this->adjMatrix[vertPos1][vertPos2] != 0 && this->adjMatrix[vertPos2][vertPos1] != 0) {

cout << "Ребро между вершинами уже есть" << endl;

return;

}

else {

this->adjMatrix[vertPos1][vertPos2] = weight;

this->adjMatrix[vertPos2][vertPos1] = weight;

}

}

else {

cout << "Какой-либо вершины нет в графе" << endl;

return;

}

}

void PrintMatrix() {//Печать матрицы смежности графа

if (!this->isEmpty()) {

cout << "Матрица смежности: " << endl;

cout << "- ";

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

cout <<" " << vetrexList[i] << " ";

}

cout << endl;

for (int i = 0; i < this->vetrexList.size(); i++) {

cout << this->vetrexList[i] << " ";

for (int j = 0; j < this->vetrexList.size(); j++) {

cout << " " << this->adjMatrix[i][j] << " ";

}

cout << endl;

}

}

else {

cout << "Граф пуст" << endl;

}

}

int GetAmoutEdges() {//Получение количества ребер для неориентированного графа

int amount = 0;

if (!this->isEmpty()) {

for (int i = 0; i < this->vetrexList.size(); i++) {

for (int j = 0; j < this->vetrexList.size(); j++) {

if (this->adjMatrix[i][j] != 0) {

amount++;

}

}

}

}

return amount / 2;

}

T& front() {//Получение первого элемента в очереди

return VertsQueue.front();

}

void removeVertex(const T& vertex) {

int pos = GetVertPos(vertex);

if (pos == -1) {

cout << "Вершины " << vertex << " нет в графе." << endl;

return;

}

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

if (adjMatrix[pos][i] != 0) removeEdge(vertex, vetrexList[i]);

if (adjMatrix[i][pos] != 0) removeEdge(vetrexList[i], vertex);

}

vetrexList.erase(vetrexList.begin() + pos);

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

adjMatrix[i].erase(adjMatrix[i].begin() + pos);

}

adjMatrix.erase(adjMatrix.begin() + pos);

size--;

cout << "Вершина " << vertex << " удалена." << endl;

}

void removeEdge(const T& vertex1, const T& vertex2) {//Функция, которая удаляет ребро

int pos1 = GetVertPos(vertex1);

int pos2 = GetVertPos(vertex2);

if (pos1 == -1 || pos2 == -1) {

cout << "Одной из вершин нет в графе." << endl;

return;

}

if (adjMatrix[pos1][pos2] == 0) {

cout << "Ребра между вершинами " << vertex1 << " и " << vertex2 << " нет." << endl;

return;

}

adjMatrix[pos1][pos2] = 0;

adjMatrix[pos2][pos1] = 0;

cout << "Ребро между вершинами " << vertex1 << " и " << vertex2 << " удалено." << endl;

}

void editEdgeWeight(const T& vertex1, const T& vertex2, int newWeight) {//Функция, которая меняет вес между ребрами

int pos1 = GetVertPos(vertex1);

int pos2 = GetVertPos(vertex2);

if (pos1 == -1 || pos2 == -1) {

cout << "Одной из вершин нет в графе." << endl;

return;

}

if (adjMatrix[pos1][pos2] == 0) {

cout << "Ребра между вершинами " << vertex1 << " и " << vertex2 << " нет." << endl;

return;

}

adjMatrix[pos1][pos2] = newWeight;

adjMatrix[pos2][pos1] = newWeight;

cout << "Вес ребра между вершинами " << vertex1 << " и " << vertex2 << " изменен на " << newWeight << "." << endl;

}

void DFS(T& startVertex, bool\* visitedVerts) {//Обход графа в глубину

cout << "Вершина " << startVertex << " посещена " << endl;

visitedVerts[this->GetVertPos(startVertex)] = true;

vector<T> neighbors = this->GetNbrs(startVertex);

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

if (!visitedVerts[this->GetVertPos(neighbors[i])]) {

this->DFS(neighbors[i], visitedVerts);

}

}

}

void BFS(T& startVertex, bool\* visitedVerts) {//Обход графа в ширину

if (visitedVerts[this->GetVertPos(startVertex)] == false) {

this->VertsQueue.push(startVertex);

cout << "Вершина " << startVertex << " обработана" << endl;

visitedVerts[this->GetVertPos(startVertex)] = true;

}

vector<T> neighbors = this->GetNbrs(startVertex);

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

if (!visitedVerts[this->GetVertPos(neighbors[i])]) {

this->VertsQueue.push(neighbors[i]);

visitedVerts[this->GetVertPos(neighbors[i])] = true;

cout << "Вершина " << neighbors[i] << " обработана" << endl;

}

}

if (this->VertsQueue.empty())

return;

T nextVertex = VertsQueue.front();

VertsQueue.pop();

BFS(nextVertex, visitedVerts);

}

void FillLabels(T& startVertex) {

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

labelList[i] = 1000000; // инициализация большими значениями

}

int pos = GetVertPos(startVertex);

labelList[pos] = 0;

}

bool AllVisiited(vector<bool>& visitedVerts) {

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

if (!visitedVerts[i]) { // проверка на непосещенные вершины

return false;

}

}

return true;

}

void Dijkstra(const T& startVertex) {//Алгортим обхода Дейкстры

int n = this->GetAmountVerts();

vector<int> dist(n, INT\_MAX); // Инициализация расстояний до всех вершин как бесконечности

vector<bool> visited(n, false); // Инициализация всех вершин как непосещенных

int startIdx = this->GetVertPos(startVertex); // Получение индекса начальной вершины

dist[startIdx] = 0; // Расстояние до самого себя равно 0

for (int i = 0; i < n - 1; ++i) {

// Находим вершину с минимальным расстоянием от начальной вершины, из еще не посещенных вершин

int minDist = INT\_MAX, minIdx{};

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

if (!visited[j] && dist[j] <= minDist) {

minDist = dist[j];

minIdx = j;

}

}

// Помечаем выбранную вершину как посещенную

visited[minIdx] = true;

// Обновляем значение расстояния для всех соседей выбранной вершины

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

if (!visited[j] && this->adjMatrix[minIdx][j] && dist[minIdx] != INT\_MAX

&& dist[minIdx] + this->adjMatrix[minIdx][j] < dist[j]) {

dist[j] = dist[minIdx] + this->adjMatrix[minIdx][j];

}

}

}

// Выводим полученные расстояния от начальной вершины до всех остальных

cout << "Вершина \tРасстояние от начальной вершины\n";

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

cout << this->vetrexList[i] << "\t\t" << dist[i] << "\n";

}

}

};

int main()

{

system("chcp 1251>NULL");

Graph<int> graph(6);

int Verts, Edges, vertex, sourceVertex, targetVetrex,Weight,vertex1;

bool\* visitedVerts = new bool[6];

bool\* visitedVerts1 = new bool[6];

fill(visitedVerts, visitedVerts + 20, false);

cout << "Введите количество вершин: " << endl;

cin >> Verts;

cout << "Введите количество ребер графа: " << endl;

cin >> Edges;

cout << endl;

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

cout << "Вершина: ";

cin >> vertex;

graph.insertVetrex(vertex);

cout << endl;

}

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

cout << "Исходная вершина: ";

cin >> sourceVertex;

cout << endl;

cout << "Конечная вершина: ";

cin >> targetVetrex;

cout << endl;

cout << "Вес ребра: ";

cin >> Weight;

cout << endl;

int\* targetVerPtr = &targetVetrex;

graph.InsertEdge(sourceVertex, targetVetrex, Weight);

}

cout << endl;

graph.PrintMatrix();

int N;

cout << "Введите вершину, с которой начать обход: ";

cin >> vertex;

cout << endl;

cout << "Обход в глубину: " << endl;

graph.DFS(vertex, visitedVerts);

cout << "Введите вершину, с которой начать обход: ";

cin >> vertex;

cout << endl;

cout << "Обход в ширину: " << endl;

graph.BFS(vertex, visitedVerts);

cout << "Введите вершину, с которой начать обход: ";

cin >> vertex;

cout << endl;

cout << "Обход дейкстры: " << endl;

graph.Dijkstra(vertex);

delete[] visitedVerts;

cout << "Введите вершину, которую хотите удалить: ";

cin >> N;

graph.removeVertex(N);

graph.PrintMatrix();

cout << "Текущий размер графа: "<< graph.GetAmountVerts();

cout << endl;

int g, k;

cout << "Введите кол-во вершин, которые вы хотите добавить: ";

cin >> g;

cout << "Введите кол-во ребёр, которые хотите добавить: ";

cin >> k;

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

cout << "Вершина: ";

cin >> vertex;

graph.insertVetrex(vertex);

cout << endl;

}

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

cout << "Исходная вершина: ";

cin >> sourceVertex;

cout << endl;

cout << "Конечная вершина: ";

cin >> targetVetrex;

cout << endl;

cout << "Вес ребра: ";

cin >> Weight;

cout << endl;

int\* targetVerPtr = &targetVetrex;

graph.InsertEdge(sourceVertex, targetVetrex, Weight);

}

graph.PrintMatrix();

cout << "Введите номера вершин, между которыми нужно удалить ребро: ";

cin >> vertex;

cin >> vertex1;

graph.removeEdge(vertex, vertex1);

cout << "Граф после удаления ребра: ";

cout << endl;

graph.PrintMatrix();

cout << endl << endl;

cout << "Введите номера вершин, между которыми нужно изменить вес ребра: ";

cin >> vertex;

cin >> vertex1;

cout << endl << endl;

cout << "Введите нужный вес: ";

cin >> Weight;

graph.editEdgeWeight(vertex, vertex1, Weight);

cout << "Граф после изменения веса ребра: ";

cout << endl;

graph.PrintMatrix();

return 0;

}

**Визуализация:**

**Main.cpp**

[#include](https://vk.com/im?sel=459190403&st=%23include) "mainwindow.h"  
[#include](https://vk.com/im?sel=459190403&st=%23include) "graph.h"  
  
[#include](https://vk.com/im?sel=459190403&st=%23include) <QApplication>  
  
int main(int argc, char \*argv[])  
{  
QApplication a(argc, argv);  
Graph w;  
[w.show](https://vk.com/away.php?to=http%3A%2F%2Fw.show&cc_key=)();  
return a.exec();  
}

**Mainwindow.cpp**

[#include](https://vk.com/im?sel=459190403&st=%23include) "mainwindow.h"  
[#include](https://vk.com/im?sel=459190403&st=%23include) "ui\_mainwindow.h"  
  
MainWindow::MainWindow(QWidget \*parent)  
: QMainWindow(parent)  
, ui(new Ui::MainWindow)  
{  
ui->setupUi(this);  
}  
  
MainWindow::~MainWindow()  
{  
delete ui;  
}

**Mainwindow.h**

[#ifndef](https://vk.com/im?sel=459190403&st=%23ifndef) MAINWINDOW\_H  
[#define](https://vk.com/im?sel=459190403&st=%23define) MAINWINDOW\_H  
  
[#include](https://vk.com/im?sel=459190403&st=%23include) <QMainWindow>  
  
QT\_BEGIN\_NAMESPACE  
namespace Ui { class MainWindow; }  
QT\_END\_NAMESPACE  
  
class MainWindow : public QMainWindow  
{  
Q\_OBJECT  
  
public:  
MainWindow(QWidget \*parent = nullptr);  
~MainWindow();  
  
private:  
Ui::MainWindow \*ui;  
};  
[#endif](https://vk.com/im?sel=459190403&st=%23endif) // MAINWINDOW\_H

**Mainwindow.ui**

<?xml version="1.0" encoding="UTF-8"?>  
<ui version="4.0">  
<class>MainWindow</class>  
<widget class="QMainWindow" name="MainWindow">  
<property name="geometry">  
<rect>  
<x>0</x>  
<y>0</y>  
<width>894</width>  
<height>655</height>  
</rect>  
</property>  
<property name="windowTitle">  
<string>MainWindow</string>  
</property>  
<widget class="QWidget" name="centralwidget">  
<layout class="QGridLayout" name="gridLayout"/>  
</widget>  
<widget class="QMenuBar" name="menubar">  
<property name="geometry">  
<rect>  
<x>0</x>  
<y>0</y>  
<width>894</width>  
<height>26</height>  
</rect>  
</property>  
</widget>  
<widget class="QStatusBar" name="statusbar"/>  
</widget>  
<resources/>  
<connections/>  
</ui>

**Graph.cpp**

#include "graph.h"

#include <QTextStream>

#include <QMessageBox>

#include <map>

QString str;//для функций

Graph::Graph(QWidget \*parent) : QGraphicsView(parent) {

scene = new QGraphicsScene(this);

scene->setItemIndexMethod(QGraphicsScene::NoIndex);

scene->setSceneRect(-500, -450, 900, 900);

setScene(scene);

setCacheMode(CacheBackground);

setViewportUpdateMode(BoundingRectViewportUpdate);

setRenderHint(QPainter::Antialiasing);

setTransformationAnchor(AnchorUnderMouse);

scale(qreal(0.8), qreal(0.8));

setMinimumSize(800, 900);

}

void Graph::itemMoved() {

if(!timerId) {

timerId = startTimer(1000 / 25);

}

}

void Graph::timerEvent(QTimerEvent \*event) {

Q\_UNUSED(event);

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

for(Vertex \*temp : qAsConst(vertexList)){

temp->calculateForces();

}

bool itemsMoved = false;

for(Vertex \*temp : qAsConst(vertexList)){

if(temp->advancePosition()){

itemsMoved = true;

}

}

if(!itemsMoved){

killTimer(timerId);

timerId = 0;

}

}

void Graph::drawBackground(QPainter \*painter, const QRectF &rect){

Q\_UNUSED(rect);

QRectF sceneRect = this->sceneRect();

QRectF rightShadow(sceneRect.right(), sceneRect.top() + 5, 5, sceneRect.height());

QRectF bottomShadow(sceneRect.left() + 5, sceneRect.bottom(), sceneRect.width(), 5);

if (rightShadow.intersects(rect) || rightShadow.contains(rect)){

painter->fillRect(rightShadow, Qt::darkGray);

}

if (bottomShadow.intersects(rect) || bottomShadow.contains(rect)){

painter->fillRect(bottomShadow, Qt::darkGray);

}

QLinearGradient gradient(sceneRect.topLeft(), sceneRect.bottomRight());

gradient.setColorAt(0, Qt::white);

gradient.setColorAt(1, Qt::lightGray);

painter->fillRect(rect.intersected(sceneRect), gradient);

painter->setBrush(Qt::NoBrush);

painter->drawRect(sceneRect);

QRectF textRect(sceneRect.left() + 4, sceneRect.top() + sceneRect.height()/9 + 35, sceneRect.width() - 4, 20);

QFont font = painter->font();

font.setBold(true);

font.setPointSize(14);

painter->setFont(font);

painter->setPen(Qt::lightGray);

createTabWidget(rect);

}

void Graph::createTabWidget(const QRectF &rect) {

Q\_UNUSED(rect);

QRectF sceneRect = this->sceneRect();

QWidget \*CommandsWidget = new QWidget;

CommandsWidget->setGeometry(sceneRect.left() + 1, sceneRect.top() + 3, sceneRect.width()/6, sceneRect.height() - 2);

QVBoxLayout \*vertexTabLayout = new QVBoxLayout(CommandsWidget);

QFont textfont = QFont();

textfont.setPointSize(12);

//кнопки для вершины

QLabel \*l = new QLabel(tr("⚪:"));

auto \*f = new QFont;

f->setPixelSize(40);

l->setFont(\*f);

vertexTabLayout->addWidget(l);

//кнопка добавления вершины

QPushButton \*addVertexButton = new QPushButton();

addVertexButton->setText(tr("➕"));

addVertexButton->setFont(textfont);

addVertexButton->setMaximumSize(QSize(120, 120));

vertexTabLayout->addWidget(addVertexButton);

connect(addVertexButton, &QPushButton::clicked, this, &Graph::insertVertex);

//кнопка удаления вершины

QPushButton \*eraseVertexButton = new QPushButton();

eraseVertexButton->setText(tr("🗑"));

eraseVertexButton->setFont(textfont);

textfont.setPointSize(12);

eraseVertexButton->setMaximumSize(QSize(120, 120));

vertexTabLayout->addWidget(eraseVertexButton);

connect(eraseVertexButton, &QPushButton::clicked, this, &Graph::createEraseVertexWindow);

//кнопки для ребра

QLabel \*l2 = new QLabel(tr("━▶:"));

auto \*f2 = new QFont;

f2->setPixelSize(40);

l2->setFont(\*f2);

vertexTabLayout->addWidget(l2);

//кнопка добавления ребра

QPushButton \*addEdgeButton = new QPushButton();

addEdgeButton->setText(tr("➕"));

addEdgeButton->setFont(textfont);

addEdgeButton->setMaximumSize(QSize(120, 120));

connect(addEdgeButton, &QPushButton::clicked, this, &Graph::createAddEdgeWindow);

vertexTabLayout->addWidget(addEdgeButton);

//кнопка для обновления веса вершины

QPushButton \*updateWeightButton = new QPushButton();

updateWeightButton->setText(tr("🔁"));

updateWeightButton->setFont(textfont);

updateWeightButton->setMaximumSize(QSize(120, 120));

connect(updateWeightButton, &QPushButton::clicked, this, &Graph::createUpdateWeightWindow);

vertexTabLayout->addWidget(updateWeightButton);

//удаление вершины

QPushButton \*eraseEdgeButton = new QPushButton();

eraseEdgeButton->setText(tr("🗑"));

eraseEdgeButton->setFont(textfont);

eraseEdgeButton->setMaximumSize(QSize(120, 120));

connect(eraseEdgeButton, &QPushButton::clicked, this, &Graph::createEraseEdgeWindow);

vertexTabLayout->addWidget(eraseEdgeButton);

//кнопки для графа

QLabel \*l3 = new QLabel(tr("-----"));

auto \*f3 = new QFont;

f3->setPixelSize(40);

l3->setFont(\*f3);

vertexTabLayout->addWidget(l3);

//кнопка функции графа

QPushButton \*FuncButton = new QPushButton();

FuncButton->setText(tr("🔎"));

FuncButton->setFont(textfont);

FuncButton->setMaximumSize(QSize(120, 120));

connect(FuncButton, &QPushButton::clicked, this, &Graph::createFunctionWindow);

vertexTabLayout->addWidget(FuncButton);

//кнопка удаления графа

QPushButton \*clearButton = new QPushButton();

clearButton->setText(tr("🗑"));

clearButton->setFont(textfont);

clearButton->setMaximumSize(QSize(120, 120));

connect(clearButton, &QPushButton::clicked, this, &Graph::clear);

vertexTabLayout->addWidget(clearButton);

//кнопка вывода матрицы смежности

QPushButton \*AdjMatrixButton = new QPushButton();

AdjMatrixButton->setText(tr("ℹ️"));

AdjMatrixButton->setFont(textfont);

AdjMatrixButton->setMaximumSize(QSize(120, 120));

connect(AdjMatrixButton, &QPushButton::clicked, this, &Graph::createAdjMatrixWindow);

vertexTabLayout->addWidget(AdjMatrixButton);

scene->addWidget(CommandsWidget);

}

void Graph::createAdjMatrixWindow() {//вывод матрицы смежности

int rows = vertexList.size() + 1;

int cols = vertexList.size() + 1;

QTableWidget \*AdjMatrixWindow = new QTableWidget(rows, cols);

AdjMatrixWindow->setColumnWidth(0, 30);

AdjMatrixWindow->setRowHeight(0, 30);

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

AdjMatrixWindow->setRowHeight(1+i, 30);

AdjMatrixWindow->setItem(0, i+1, new QTableWidgetItem(QString::number(i)));

AdjMatrixWindow->item(0, i+1)->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);

}

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

AdjMatrixWindow->setItem(1+i, 0, new QTableWidgetItem(QString::number(i)));

AdjMatrixWindow->item(1+i, 0)->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);

for (Edge \*edge : vertexList[i]->getEdges()) {

int pos = edge->destVertex()->getIndex();

AdjMatrixWindow->setItem(1+i, pos+1, new QTableWidgetItem(QString::number(edge->getWeight())));

AdjMatrixWindow->item(1+i, pos+1)->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);

}

AdjMatrixWindow->setColumnWidth(i+1, 30);

}

AdjMatrixWindow->setWindowTitle("Матрица смежности");

AdjMatrixWindow->show();

}

int Graph::getVertexNum() {//получение размера списка вершин

return vertexList.size();

}

void Graph::scaleView(qreal scaleFactor) {

qreal factor = transform().scale(scaleFactor, scaleFactor).mapRect(QRectF(0, 0, 1, 1)).width();

if(factor < 0.07 || factor > 100){

return;

}

scale(scaleFactor, scaleFactor);

}

void Graph::insertVertex() {//вставка вершины

int vertexIndex = vertexList.size();

vertexList.append(new Vertex(this, vertexIndex));

vertexList[vertexIndex]->setPos(0, 0);

scene->addItem(vertexList[vertexIndex]);

vertexNum++;

}

void Graph::updateIndex() {

for(int i = 0; i < vertexList.size(); i++) {//исправил

vertexList[i]->setIndex(i);//Правильнее было бы назвать Name или Number

}

}

void Graph::eraseVertex(int vertexIndex) {//удаление вершины

Vertex \*temp = vertexList[vertexIndex];

for(Edge \*edge : temp->getEdges()) {//удаление всех дуг, связанных с вершиной

eraseEdge(vertexIndex, edge->destVertex()->getIndex());

}

temp->clearEdge();

scene->removeItem(temp);

edgeNum -= vertexList[vertexIndex]->getEdges().size()\*2;//изменяем количество дуг

vertexList.erase(vertexList.begin() + vertexIndex);

updateIndex();

vertexNum--;//изменяем количество вершин

}

void Graph::insertEdge(int source, int dest, double weight){//Вставка дуги

if(checkAdjacent(source, dest)) {//если уже соединены - выход

return;

}

//создаём двунаправленную стрелку

scene->addItem(new Edge(vertexList[source], vertexList[dest], weight));//убрать вес

scene->addItem(new Edge(vertexList[dest], vertexList[source], weight));

edgeNum += 2;

}

void Graph::updateWeight(int source, int dest, double weight){//изменение веса

if (!checkAdjacent(source, dest)){//если не соединены - выход

return;

}

vertexList[source]->updateWeight(dest, weight);

vertexList[dest]->updateWeight(source, weight);

}

void Graph::eraseEdge(int source, int dest) {//удаление дуги

if (!checkAdjacent(source, dest)) {//если вершины не соединены - выход

return;

}

//удаление для начальной вершины

for (Edge \*edge : vertexList[source]->getEdges()) {

if(edge->destVertex()->getIndex() == dest) {

vertexList[source]->eraseEdge(dest);

scene->removeItem(edge);

}

}

//удаление для конечной вершины

for (Edge \*edge : vertexList[dest]->getEdges()){

if(edge->destVertex()->getIndex() == source){

vertexList[dest]->eraseEdge(source);

scene->removeItem(edge);

}

}

edgeNum -= 2;

}

bool Graph::checkAdjacent(int source, int dest) {//Проверка, соединены ли вершины

return vertexList[source]->pathExist(dest);

}

int Graph::getDegree(int vertexIndex) {

return vertexList[vertexIndex]->getDegree();

}

void Graph::clear() {//полное удаление графа

while (!vertexList.empty()) {

eraseVertex(0);

}

}

int \*\* Graph::GetAdjMatrix() {

int \*\*AdjMatrix = new int\*[vertexNum] {};

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

AdjMatrix[i] = new int[vertexNum]{};

}

for (Vertex\* i : vertexList) {

for (Edge\* j: i->getEdges()) {

AdjMatrix[i->getIndex()][j->destVertex()->getIndex()] = j->getWeight();

}

}

return AdjMatrix;

}

void Graph::RemoveAdjMatrix(int \*\* AdjMatrix) {

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

delete[] AdjMatrix[i];

}

delete[] AdjMatrix;

}

void Graph::createEraseVertexWindow() {//Окно для удаления выбранной вершины

window = new QWidget;

input1 = new QLineEdit;

QLabel \*label = new QLabel;

label->setFrameStyle(QFrame::Box | QFrame::Plain);

QPushButton \*okButton = new QPushButton(tr("OK"));

QGridLayout \*layout = new QGridLayout;

layout->addWidget(new QLabel(tr("Введите номер вершины:")), 0, 0);

layout->addWidget(input1, 0, 1);

layout->addWidget(okButton, 1, 1, Qt::AlignRight);

layout->setSizeConstraint(QLayout::SetFixedSize);

window->setLayout(layout);

window->setWindowTitle("Удаление вершины");

connect(okButton, &QPushButton::clicked, this, &Graph::eraseVertexSlot);

window->show();

}

void Graph::eraseVertexSlot() {//удаление выбранной вершины

int srcIndex = input1->text().toInt();

window->close();

if (srcIndex >= vertexList.size()){

return;

}

eraseVertex(srcIndex);

}

void Graph::createAddEdgeWindow() {//окно для добавления дуги

window = new QWidget;

input1 = new QLineEdit;

input2 = new QLineEdit;

input3 = new QLineEdit;

QLabel \*label = new QLabel;

label->setFrameStyle(QFrame::Box | QFrame::Plain);

QPushButton \*okButton = new QPushButton(tr("OK"));

QGridLayout \*layout = new QGridLayout;

layout->addWidget(new QLabel(tr("Введите начальную вершину:")), 0, 0);

layout->addWidget(input1, 0, 1);

layout->addWidget(new QLabel(tr("Введите конечную вершину:")), 1, 0);

layout->addWidget(input2, 1, 1);

layout->addWidget(new QLabel(tr("Введите вес дуги:")), 2, 0);

layout->addWidget(input3, 2, 1);

layout->addWidget(okButton, 3, 1, Qt::AlignRight);

layout->setSizeConstraint(QLayout::SetFixedSize);

window->setLayout(layout);

window->setWindowTitle("Добавление дуги");

connect(okButton, &QPushButton::clicked, this, &Graph::addEdgeSlot);

window->show();

}

void Graph::addEdgeSlot() {//добавление дуги

int srcIndex = input1->text().toInt();

int destIndex = input2->text().toInt();

double edgeWeight = input3->text().toDouble();

if (srcIndex >= vertexList.size() destIndex >= vertexList.size() edgeWeight <= 0){

return;

}

insertEdge(srcIndex, destIndex, edgeWeight);

}

void Graph::createEraseEdgeWindow() {//окно для удаления выбранной дуги

window = new QWidget;

input1 = new QLineEdit;

input2 = new QLineEdit;

QLabel \*label = new QLabel;

label->setFrameStyle(QFrame::Box | QFrame::Plain);

QPushButton \*okButton = new QPushButton(tr("OK"));

QGridLayout \*layout = new QGridLayout;

layout->addWidget(new QLabel(tr("Номер начальной вершины:")), 0, 0);

layout->addWidget(input1, 0, 1);

layout->addWidget(new QLabel(tr("Номер конечной вершины:")), 1, 0);

layout->addWidget(input2, 1, 1);

layout->addWidget(okButton, 2, 1, Qt::AlignRight);

layout->setSizeConstraint(QLayout::SetFixedSize);

window->setLayout(layout);

window->setWindowTitle("Удаление дуги");

connect(okButton, &QPushButton::clicked, this, &Graph::eraseEdgeSlot);

window->show();

}

void Graph::eraseEdgeSlot() {//удаление выбранной дуги

int srcIndex = input1->text().toInt();

int destIndex = input2->text().toInt();

window->close();

if (srcIndex >= vertexList.size() || destIndex >= vertexList.size()){

return;

}

eraseEdge(srcIndex, destIndex);

}

void Graph::createUpdateWeightWindow() {//окно для изменения веса

window = new QWidget;

input1 = new QLineEdit;

input2 = new QLineEdit;

input3 = new QLineEdit;

QLabel \*label = new QLabel;

label->setFrameStyle(QFrame::Box | QFrame::Plain);

QPushButton \*okButton = new QPushButton(tr("OK"));

QGridLayout \*layout = new QGridLayout;

layout->addWidget(new QLabel(tr("Введите номер начальной вершины:")), 0, 0);

layout->addWidget(input1, 0, 1);

layout->addWidget(new QLabel(tr("Введите номер конечной вершины:")), 1, 0);

layout->addWidget(input2, 1, 1);

layout->addWidget(new QLabel(tr("Новый вес:")), 2, 0);

layout->addWidget(input3, 2, 1);

layout->addWidget(okButton, 3, 1, Qt::AlignRight);

layout->setSizeConstraint(QLayout::SetFixedSize);

window->setLayout(layout);

window->setWindowTitle("Изменение веса дуги");

connect(okButton, &QPushButton::clicked, this, &Graph::updateWeightSlot);

window->show();

}

void Graph::updateWeightSlot() {//изменение веса

int srcIndex = input1->text().toInt();

int destIndex = input2->text().toInt();

double edgeWeight = input3->text().toDouble();

window->close();

if(srcIndex >= vertexList.size() destIndex >= vertexList.size() edgeWeight <= 0){

return;

}

updateWeight(srcIndex, destIndex, edgeWeight);

}

void Graph::createFunctionWindow() {

window = new QWidget;

input1 = new QLineEdit;

input5 = new QLineEdit;

QLabel \*label = new QLabel;

label->setText("1. DFS\n2. BFS\n3. Алгоритм Дейкстры\n4. Задача Коммивояжёра\n");

QPushButton \*okButton = new QPushButton(tr("OK"));

QGridLayout \*layout = new QGridLayout;

layout->addWidget(label, 0, 0);

layout->addWidget(new QLabel(tr("Номер функции:")), 1, 0);

layout->addWidget(input5, 1, 1);

layout->addWidget(new QLabel(tr("Номер начальной вершины:")), 2, 0);

layout->addWidget(input1, 2, 1);

layout->addWidget(okButton, 3, 0, Qt::AlignRight);

layout->setSizeConstraint(QLayout::SetFixedSize);

window->setLayout(layout);

window->setWindowTitle("Выбор функции");

connect(okButton, &QPushButton::clicked, this, &Graph::StartSelectedFunction);

window->show();

}

void Graph::runDFS(int temp, bool\* visited) {

if (!visited[temp]) {

visited[temp] = true;

str += "Вершина " + QString::number(temp) + " посещена\n";

}

Vertex \*tempVertex = vertexList[temp];

for (Edge \*tempEdge : tempVertex->getEdges()) {//проход по соседям

if (!visited[tempEdge->destVertex()->getIndex()]) {

runDFS(tempEdge->destVertex()->getIndex(), visited);

}

}

}

void Graph::runBFS(int index, bool\* visited, QQueue<int>\* bfsQueue) {

if (visited[index] == false) {

bfsQueue->push\_back(index);

str += "Вершина " + QString::number(index) + " обработана\n";

visited[index] = true;

}

Vertex \*tempVertex = vertexList[index];

bfsQueue->pop\_front();

for (Edge \*tempEdge : tempVertex->getEdges()) {

if(!visited[tempEdge->destVertex()->getIndex()]) {

str += "Вершина " + QString::number(tempEdge ->destVertex() ->getIndex()) + " обработана\n";

visited[tempEdge->destVertex()->getIndex()] = true;

bfsQueue->push\_back(tempEdge->destVertex()->getIndex());

}

}

if (!bfsQueue->empty()) {

runBFS(bfsQueue->front(), visited, bfsQueue);

}

}

void Graph::runDijkstra(int vertexIndex) {

int\*\* AdjMatrix = GetAdjMatrix();

QVector <int> distance(vertexNum, INT\_MAX);

distance[vertexIndex] = 0;

map<int, int> m;

m[0] = vertexIndex;

while(!m.empty()) {

int m\_lenght = (\*m.begin()).first;

int m\_vertex = (\*m.begin()).second;

m.erase(m.begin());

if (m\_lenght <= distance[m\_vertex]) {

distance[m\_vertex] = m\_lenght;

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

if (distance[i] > distance[m\_vertex] + AdjMatrix[m\_vertex][i] && AdjMatrix[m\_vertex][i] != 0) {

distance[i] = distance[m\_vertex] + AdjMatrix[m\_vertex][i];

m[distance[i]] = i;

}

}

}

}

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

if (distance[i] != INT\_MAX && i != vertexIndex) {

str += "Расстояние от вершины " + QString::number(vertexIndex) + " до вершины " + QString::number(i) + " cоставляет " + QString::number(distance[i]) + '\n';

}

else if (i != vertexIndex) {

str += "Вершины " + QString::number(vertexIndex) + " и " + QString::number(i) + " не соединены\n";

}

}

RemoveAdjMatrix(AdjMatrix);

}

void Graph::runTSP() {

int \*\*AdjMatrix = GetAdjMatrix();//получение матрицы смежности

//проверка на возможность выполнения функции

bool FlagToContinue = (vertexNum > 2);

for (int i = 0; i < vertexNum && FlagToContinue; i++) {

FlagToContinue = (vertexList[i]->getEdges().size() > 1);

}

if (FlagToContinue) {

int \*\* OrigianalAdjMatrix = GetAdjMatrix();

int m = INT\_MAX, Sum = 0;

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

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

if (AdjMatrix[i][j] == 0) {

AdjMatrix[i][j] = m;

}

}

}

QMap<int,int> roads = {};

for (int z = 0; z < vertexNum; z++) {

int maxi = 0, maxj = 0;

int di[vertexNum], dj[vertexNum];

int maxScore = 0;

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

di[i] = dj[i] = m;

}

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

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

if (AdjMatrix[i][j] < di[i]) {

di[i] = AdjMatrix[i][j];

}

}

}

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

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

if (AdjMatrix[i][j] != m) {

AdjMatrix[i][j] -= di[i];

}

}

}

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

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

if (AdjMatrix[j][i] < dj[i]) {

dj[i] = AdjMatrix[j][i];

}

}

}

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

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

if (AdjMatrix[j][i] != m) {

AdjMatrix[j][i] -= dj[i];

}

}

}

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

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

int imin = m, jmin = m;

if (AdjMatrix[i][j] == 0) {

for (int k = 0; k < vertexNum; k++) {

if (AdjMatrix[k][j] < imin && i != k) {

imin = AdjMatrix[k][j];

}

if (AdjMatrix[i][k] < jmin) {

jmin = AdjMatrix[i][k];

}

}

if (maxScore < imin + jmin) {

maxScore = imin + jmin;

maxi = i;

maxj = j;

}

}

}

}

roads[maxj] = maxi;

AdjMatrix[maxj][maxi] = m;

Sum += OrigianalAdjMatrix[maxi][maxj];

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

AdjMatrix[i][maxj] = m;

AdjMatrix[maxi][i] = m;

}

}

int i = 0;

str += QString::number(i);

while(FlagToContinue) {

str += "-> " + QString::number(roads[i]);

i = roads[i];

FlagToContinue = (i != 0);

}

str += " Сумма = " + QString::number(Sum);

RemoveAdjMatrix(OrigianalAdjMatrix);

}

else {

str += "Решение задачи Коммивояжёра невозможно!\nКаждая вершина графа должна иметь как минимум два ребра!";

}

RemoveAdjMatrix(AdjMatrix);

}

void Graph::StartSelectedFunction() {

int index = input1->text().toInt();

int NumOfFunction = input5->text().toInt();

if (index >= 0 && index < vertexList.size() && NumOfFunction >= 0 && NumOfFunction < 5) {

enum Functions {

DFS = 1, BFS, Dijkstra, Travel

};

switch (NumOfFunction) {

case(Functions::DFS): {

bool\* visited = new bool[vertexList.size()];

runDFS(index, visited);

createDFSWindow();

delete[] visited;

break;

}

case(Functions::BFS): {

bool\* visited = new bool[vertexList.size()];

runBFS(index, visited, new QQueue<int>);

createBFSWindow();

delete[] visited;

break;

}

case(Functions::Dijkstra): {

runDijkstra(index);

createDijkstraWindow();

break;

}

case(Functions::Travel): {

runTSP();

createTSPWindow();

break;

}

}

}

}

void Graph::createDFSWindow() {

window = new QWidget;

QGridLayout \*layout = new QGridLayout;

layout->addWidget(new QLabel(str), 0, 0);

str = "";

window->setLayout(layout);

window->setWindowTitle("DFS");

window->show();

}

void Graph::createBFSWindow() {

window = new QWidget;

QGridLayout \*layout = new QGridLayout;

layout->addWidget(new QLabel(str), 0, 0);

str = "";

window->setLayout(layout);

window->setWindowTitle("BFS");

window->show();

}

void Graph::createDijkstraWindow() {

window = new QWidget;

QGridLayout \*layout = new QGridLayout;

layout->addWidget(new QLabel(str), 0, 0);

str = "";

window->setLayout(layout);

window->setWindowTitle("Алгоритм Дейкстры");

window->show();

}

void Graph::createTSPWindow() {

window = new QWidget;

QGridLayout \*layout = new QGridLayout;

layout->addWidget(new QLabel(str), 0, 0);

str = "";

window->setLayout(layout);

window->setWindowTitle("Решение задачи Коммивояжёра");

window->show();

}

**Работа программы:**











