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**

**«Информатика»**

**на тему**

**«Графы и задача коммивояжера»**

Выполнил студент гр. ИВТ-23-1б

Бакин Владислав Артемович

Проверил:

доц. каф. ИТАС

Полякова Ольга Андреевна

Яруллин Денис Владимирович

(оценка) (подпись)

(дата)

г. Пермь, 2024

# Графы и задача коммивояжёра

# Постановка задачи

Разработать приложение с графическим интерфейсом для генерации графов и выполнения обходов по ним

# Функционал

1. Добавление узла
2. Перемещение узла
3. Удаление узла
4. Добавление ребра
5. Удаление ребра
6. Диалоговое окно для выполнения обходов
7. Диалоговое окно для решения задачи коммивояжёра

# Код программы

### Заголовочные файлы

// Диалоговое окно добавления ребра  
  
#ifndef ADDEDGE\_H  
#define ADDEDGE\_H  
  
#include <QDialog>  
#include <QGraphicsScene>  
  
#include "graph.h"  
  
namespace Ui {  
 class AddNewEdgeWindow;  
}  
  
class AddNewEdgeWindow : public QDialog  
{  
 Q\_OBJECT  
  
public:  
 explicit AddNewEdgeWindow(Graph& graph, QWidget \*parent = nullptr);  
 ~AddNewEdgeWindow();  
  
private slots:  
 void addNewEdge();  
  
private:  
 Graph& m\_graph;  
 Ui::AddNewEdgeWindow \*ui;  
};  
  
#endif // ADDEDGE\_H

// Диалоговое окно удаления ребра  
  
#ifndef DELETEEDGE\_H  
#define DELETEEDGE\_H  
  
#include <QDialog>  
  
#include "graph.h"  
  
namespace Ui {  
 class DeleteEdgeWindow;  
}  
  
class DeleteEdgeWindow : public QDialog  
{  
 Q\_OBJECT  
  
public:  
 explicit DeleteEdgeWindow(Graph& graph, QWidget \*parent = nullptr);  
 ~DeleteEdgeWindow();  
  
public slots:  
 void delEdge();  
  
private:  
 Graph& m\_graph;  
 Ui::DeleteEdgeWindow \*ui;  
};  
  
#endif // DELETEEDGE\_H

// Диалоговое окно удаления узла  
  
#ifndef DELETENODE\_H  
#define DELETENODE\_H  
  
#include <QDialog>  
#include <QGraphicsView>  
  
#include "graph.h"  
  
namespace Ui {  
 class DeleteNodeWindow;  
}  
  
class DeleteNodeWindow : public QDialog  
{  
 Q\_OBJECT  
  
public:  
 explicit DeleteNodeWindow(Graph& graph, QGraphicsScene\* scene, QGraphicsView\* printGraph, QGraphicsItemGroup\*\* matrixOfGroups, QWidget \*parent = nullptr);  
 ~DeleteNodeWindow();  
  
public slots:  
 void delNode();  
  
private:  
 Graph& m\_graph;  
 QGraphicsScene\* m\_scene;  
 QGraphicsView\* m\_printGraph;  
 QGraphicsItemGroup\*\* m\_matrixOfGroups;  
 Ui::DeleteNodeWindow \*ui;  
};  
  
#endif // DELETENODE\_H

#ifndef GRAPH\_H  
#define GRAPH\_H  
  
#include <QGraphicsEllipseItem>  
#include <QDebug>  
#include <vector>  
#include <queue>  
  
class Graph  
{  
private:  
 int sizeOfMatrix; // размер матрицы смежности  
 int\*\* matrix; // матрица смежности  
 QGraphicsEllipseItem\*\* matrixOfEllipses; // массив указателей на эллипсы  
 QGraphicsItemGroup\*\* matrixOfGroups; // массив указателей на группы узлов  
 std::vector<QGraphicsItemGroup\*> vectorOfArrows; // вектор указателей на группы стрелок  
 int curSerialNumber; // порядковый номер нового узла  
  
public:  
 Graph();  
 ~Graph();  
  
 // Геттеры:  
 int getSize(); // возвращает размер матрицы смежности (с учетом координатной строки и столбца)  
 int getSerialNumber(); // возвращает текущий порядковый номер узла (не самый большой порядковый номер, а тот который будет следущий при добавлении)  
 int\*\* getMatrix(); // возвращает матрицу смежности  
 QGraphicsEllipseItem\*\* getMatrixOfEllipses(); // возвращает массив указателей на объект эллипса  
 QGraphicsItemGroup\*\* getMatrixOfGroups(); // возвращает массив указателей на группы узлов (эллипс и текст - порядковый номер)  
 std::vector<QGraphicsItemGroup\*>& getVectorOfArrows(); // возвращает вектор указателей на группы ребер  
  
 // Изменение матрицы смежности:  
 void addEdge(int, int, int); // добавление ребра в матрицу смежности  
 void delEdge(int, int); // удаление ребра из матрицы смежности  
 void delNode(int); // удаление узла из матрицу смежности  
  
 void resizeMatrix(int); // увеличение матрицы (при добавлении узлов)  
 void increaseSerialNumber(); // увеличение порядкового номера (при добавлении узлов)  
 void addNodeOnLastPos(QGraphicsEllipseItem\*, QGraphicsItemGroup\*); // добавление координатной ячейки в матрице смежности + изменение matrixOfEllipses и matrixOfGroups  
 void updateMatrixOfEllipses(int, QGraphicsEllipseItem\*); // обновление matrixOfEllipses  
 void updateMatrixOfGroups(int, QGraphicsItemGroup\*); // обновление matrixOfGroups  
 void updateVector(std::vector<QGraphicsItemGroup\*>&); // обновление вектора стрелок  
  
 // Для отладки:  
 void printMatrix(); // вывод матрицы смежности в консоль  
  
 // Обходы и задача коммивояжера  
 std::vector<int> dfs(int);  
 void dfs(int, std::vector<int>&);  
 std::vector<int> bfs(int);  
 std::vector<std::vector<int>> floyd();  
 std::vector<int> dijkstra(int);  
 std::vector<int> tsp(int, int);  
  
};  
  
#endif // GRAPH\_H

#ifndef GRAPHGENERATOR\_H  
#define GRAPHGENERATOR\_H  
  
#include <QMainWindow>  
#include <QDebug>  
#include <QGraphicsSceneMouseEvent>  
#include <QtMath>  
  
#include "graph.h"  
#include "addedge.h"  
#include "deleteedge.h"  
#include "deletenode.h"  
#include "searchmenu.h"  
#include "tsp.h"  
#include "graph.h"  
  
QT\_BEGIN\_NAMESPACE  
namespace Ui { class GraphGenerator; }  
QT\_END\_NAMESPACE  
  
class GraphGenerator : public QMainWindow  
{  
 Q\_OBJECT  
  
public:  
 GraphGenerator(QWidget \*parent = nullptr);  
 ~GraphGenerator();  
  
 QPolygonF createArrowHead(const QPointF& startPoint, const QPointF& endPoint); // визуализация наконечника стрелки  
 void updateSerialNumbers(); // обновление нумерации узлов, если удаляется узел  
  
public slots:  
 // Визуализация графа:  
 void addNode(); // добавление узла  
 void delNode(); // удаление узла  
 void addEdge(); // добавление ребра  
 void delEdge(); // удаление ребра  
 void updateArrows(); // на случай, если стрелки автометически не обновились при перемещении  
 bool eventFilter(QObject \*object, QEvent \*event); // проверка на то, что был сдвинут узел (для обновления стрелок)  
  
 // Обходы графа и задача коммивояжера:  
 void openMenu(); // обход в ширину, глубину, алгоритм Флойда, алгоритм Дейкстры  
 void tsp(); // задача коммивояжера  
  
private:  
 Graph graph;  
 // "Окно" для визуализации:  
 QGraphicsScene \*scene;  
 QGraphicsView \*printGraph;  
 Ui::GraphGenerator \*ui;  
};  
#endif // GRAPHGENERATOR\_H

// Диалоговое окно - меню содержащее алгоритмы поиска по графу  
  
#ifndef SEARCHMENU\_H  
#define SEARCHMENU\_H  
  
#include <QDialog>  
  
#include "graph.h"  
  
namespace Ui {  
 class output;  
}  
  
class output : public QDialog  
{  
 Q\_OBJECT  
  
public:  
 explicit output(Graph\* graph, QWidget \*parent = nullptr);  
 ~output();  
  
public slots:  
 // Обходы:  
 void dfs(); // обход в глубину  
 void bfs(); // обход в ширину  
 void floyd(); // алгоритм Флойда  
 void dijkstra(); // алгоритм Дейкстры  
  
private:  
 Graph\* m\_graph;  
 Ui::output \*ui;  
};  
  
#endif // SEARCHMENU\_H

// Диалоговое окно решения задачи коммивояжера  
  
#ifndef TSP\_H  
#define TSP\_H  
  
#include <QDialog>  
#include <QDebug>  
  
#include "graph.h"  
  
namespace Ui {  
 class TSP;  
}  
  
class TSP : public QDialog  
{  
 Q\_OBJECT  
  
public:  
 explicit TSP(Graph\* graph, QWidget \*parent = nullptr);  
 ~TSP();  
  
public slots:  
 void tsp(); // задача коммивояжера  
  
private:  
 Graph\* m\_graph;  
 Ui::TSP \*ui;  
};  
  
#endif // TSP\_H

### Исходные файлы

#include "addedge.h"  
#include "ui\_addedge.h"  
  
AddNewEdgeWindow::AddNewEdgeWindow(Graph& graph, QWidget \*parent)  
 : QDialog(parent), m\_graph(graph), ui(new Ui::AddNewEdgeWindow)  
{  
 ui->setupUi(this);  
  
 connect(ui->addNewEdgeBtn, &QPushButton::clicked, this, &AddNewEdgeWindow::addNewEdge);  
}  
  
AddNewEdgeWindow::~AddNewEdgeWindow()  
{  
 delete ui;  
}  
  
void AddNewEdgeWindow::addNewEdge()  
{  
 // Получаем значения с lineEdit пользовательского интерфейса  
 QString fromLine = ui->fromLine->text();  
 QString toLine = ui->toLine->text();  
 QString weightLine = ui->weightLine->text();  
 // Переводим полученнные данные в int  
 int from = fromLine.toInt();  
 int to = toLine.toInt();  
 int weight = weightLine.toInt();  
  
 m\_graph.addEdge(from, to, weight); // добавляем ребро в матрицу смежности  
  
 close(); // после добавления ребра, закрываем окно  
}

#include "deleteedge.h"  
#include "ui\_deleteedge.h"  
  
DeleteEdgeWindow::DeleteEdgeWindow(Graph& graph, QWidget \*parent)  
 : QDialog(parent), m\_graph(graph), ui(new Ui::DeleteEdgeWindow)  
{  
 ui->setupUi(this);  
  
 connect(ui->delEdgeBtn, &QPushButton::clicked, this, &DeleteEdgeWindow::delEdge);  
}  
  
DeleteEdgeWindow::~DeleteEdgeWindow()  
{  
 delete ui;  
}  
  
void DeleteEdgeWindow::delEdge()  
{  
 // Получаем значения с lineEdit пользовательского интерфейса  
 QString fromLine = ui->fromLine->text();  
 QString toLine = ui->toLine->text();  
 // Переводим полученнные данные в int  
 int from = fromLine.toInt();  
 int to = toLine.toInt();  
 m\_graph.delEdge(from, to); // удаляем ребро из матрицы смежности  
  
 close(); // после удаления ребра, закрываем окно  
}

#include "deletenode.h"  
#include "ui\_deletenode.h"  
  
DeleteNodeWindow::DeleteNodeWindow(Graph& graph, QGraphicsScene\* scene, QGraphicsView\* printGraph, QGraphicsItemGroup\*\* matrixOfGroups, QWidget \*parent)  
 : QDialog(parent), m\_graph(graph), m\_scene(scene), m\_printGraph(printGraph), m\_matrixOfGroups(matrixOfGroups), ui(new Ui::DeleteNodeWindow)  
{  
 ui->setupUi(this);  
  
 connect(ui->delNodeBtn, &QPushButton::clicked, this, &DeleteNodeWindow::delNode);  
}  
  
DeleteNodeWindow::~DeleteNodeWindow()  
{  
 delete ui;  
}  
  
void DeleteNodeWindow::delNode()  
{  
 // Получаем значения с lineEdit пользовательского интерфейса  
 QString nodeNumber = ui->numberLine->text();  
 // Переводим полученнные данные в int  
 int number = nodeNumber.toInt();  
  
 // Удаляем со сцены объект узла (эллипс и текст)  
 m\_scene->removeItem(m\_matrixOfGroups[number]);  
 delete m\_matrixOfGroups[number];  
 m\_matrixOfGroups[number] = nullptr;  
  
 m\_graph.delNode(number); // удаляем узел из матрицы смежности  
 close(); // после удаления ребра, закрываем окно  
}

#include "graph.h"  
  
Graph::Graph()  
{  
 sizeOfMatrix = 1;  
 curSerialNumber = 1;  
 matrix = new int\* [sizeOfMatrix];  
 for (int i = 0; i < sizeOfMatrix; i++)  
 {  
 matrix[i] = new int [sizeOfMatrix];  
 }  
 matrix[0][0] = 0;  
  
 matrixOfEllipses = new QGraphicsEllipseItem\* [sizeOfMatrix];  
 matrixOfEllipses[0] = nullptr;  
  
 matrixOfGroups = new QGraphicsItemGroup\* [sizeOfMatrix];  
 matrixOfEllipses[0] = nullptr;  
}  
  
Graph::~Graph()  
{  
 // Освобождаем память, выделенную для матрицы смежности  
 if (matrix) {  
 for (int i = 0; i < sizeOfMatrix; ++i) {  
 delete[] matrix[i];  
 }  
 delete[] matrix;  
 }  
  
 // Освобождаем память, выделенную для массива ellipses  
 if (matrixOfEllipses) {  
 delete[] matrixOfEllipses;  
 }  
  
 // Освобождаем память, выделенную для массива groups  
 if (matrixOfGroups) {  
 delete[] matrixOfGroups;  
 }  
  
 // Освобождаем память, выделенную для элементов вектора arrows  
 for (auto arrow : vectorOfArrows) {  
 delete arrow;  
 }  
}  
  
int Graph::getSize()  
{  
 return sizeOfMatrix;  
}  
  
int Graph::getSerialNumber()  
{  
 return curSerialNumber;  
}  
  
int\*\* Graph::getMatrix()  
{  
 return matrix;  
}  
  
QGraphicsEllipseItem\*\* Graph::getMatrixOfEllipses()  
{  
 return matrixOfEllipses;  
}  
  
QGraphicsItemGroup\*\* Graph::getMatrixOfGroups()  
{  
 return matrixOfGroups;  
}  
  
std::vector<QGraphicsItemGroup\*>& Graph::getVectorOfArrows()  
{  
 return vectorOfArrows;  
}  
  
void Graph::addEdge(int from, int to, int weight)  
{  
 matrix[to][from] = weight;  
}  
  
void Graph::delEdge(int from, int to)  
{  
 matrix[to][from] = 0;  
}  
  
void Graph::delNode(int number)  
{  
 int curSize = getSize();  
  
 // Удаление строки и столбца  
 for (int i = number; i < curSize - 1; ++i)  
 {  
 for (int j = 0; j < curSize - 1; ++j)  
 {  
 // Сдвигаем элементы влево и вверх  
 if (i < number || j < number)  
 matrix[i][j] = matrix[i][j];  
 else  
 matrix[i][j] = matrix[i + 1][j + 1];  
 }  
 }  
  
 // Создаем новую матрицу  
 int\*\* newMatrix = new int\*[curSize - 1];  
 for (int i = 0; i < curSize - 1; ++i)  
 {  
 newMatrix[i] = new int[curSize - 1];  
 }  
  
 for (int i = 0; i < curSize - 1; ++i)  
 {  
 for (int j = 0; j < curSize - 1; ++j)  
 {  
 newMatrix[i][j] = matrix[i][j];  
 }  
 }  
  
 delete[] matrix;  
 matrix = newMatrix;  
  
 // Обновляем массивы групп и эллипсов  
 QGraphicsEllipseItem\*\* newMatrixOfEllipses = new QGraphicsEllipseItem\*[curSize - 1];  
 QGraphicsItemGroup\*\* newMatrixOfGroups = new QGraphicsItemGroup\*[curSize - 1];  
  
 for (int i = 0; i < number; ++i)  
 {  
 newMatrixOfEllipses[i] = matrixOfEllipses[i];  
 newMatrixOfGroups[i] = matrixOfGroups[i];  
 }  
  
 for (int i = number + 1; i < curSize; ++i)  
 {  
 newMatrixOfEllipses[i - 1] = matrixOfEllipses[i];  
 newMatrixOfGroups[i - 1] = matrixOfGroups[i];  
 }  
  
 delete[] matrixOfEllipses;  
 delete[] matrixOfGroups;  
  
 matrixOfEllipses = newMatrixOfEllipses;  
 matrixOfGroups = newMatrixOfGroups;  
  
 sizeOfMatrix--;  
 curSerialNumber--;  
  
 // Обновляем в матрице номера узлов  
 for (int i = 0; i < sizeOfMatrix; ++i)  
 {  
 matrix[0][i] = i;  
 matrix[i][0] = i;  
 }  
}  
  
void Graph::resizeMatrix(int difference) // только в большую сторону  
{  
 // matrix:  
 int\*\* newMatrix;  
 newMatrix = new int\* [sizeOfMatrix + difference];  
 for (int i = 0; i < sizeOfMatrix + difference; i++)  
 {  
 newMatrix[i] = new int [sizeOfMatrix + difference];  
 }  
  
 for (int i = 0; i < sizeOfMatrix; i++)  
 {  
 for (int j = 0; j < sizeOfMatrix; j++)  
 {  
 newMatrix[i][j] = matrix[i][j];  
 }  
 }  
  
 for (int i = sizeOfMatrix; i < sizeOfMatrix + difference; i++)  
 {  
 for (int j = 0; j < sizeOfMatrix + difference; j++)  
 {  
 newMatrix[i][j] = 0;  
 newMatrix[j][i] = 0;  
 }  
 }  
  
 for (int i = 0; i < sizeOfMatrix; i++)  
 {  
 delete[] matrix[i];  
 }  
 delete[] matrix;  
 matrix = newMatrix;  
  
 // matrixOfEllipses:  
 QGraphicsEllipseItem\*\* newMatrixOfEllipses = new QGraphicsEllipseItem\* [sizeOfMatrix + 1];  
 for (int i = 0; i < sizeOfMatrix; i++)  
 {  
 newMatrixOfEllipses[i] = matrixOfEllipses[i];  
 }  
 delete matrixOfEllipses;  
 matrixOfEllipses = newMatrixOfEllipses;  
  
 //matrixOfGroups:  
 QGraphicsItemGroup\*\* newMatrixOfGroups = new QGraphicsItemGroup\* [sizeOfMatrix + 1];  
 for (int i = 0; i < sizeOfMatrix; i++)  
 {  
 newMatrixOfGroups[i] = matrixOfGroups[i];  
 }  
 delete matrixOfGroups;  
 matrixOfGroups = newMatrixOfGroups;  
  
 sizeOfMatrix += difference;  
}  
  
void Graph::increaseSerialNumber()  
{  
 curSerialNumber += 1;  
}  
  
void Graph::addNodeOnLastPos(QGraphicsEllipseItem\* node, QGraphicsItemGroup\* group)  
{  
 // Добавляем последний добавленный узел  
 matrix[0][sizeOfMatrix - 1] = curSerialNumber;  
 matrix[sizeOfMatrix - 1][0] = curSerialNumber;  
  
 matrixOfEllipses[sizeOfMatrix - 1] = node;  
 matrixOfGroups[sizeOfMatrix - 1] = group;  
}  
  
void Graph::updateMatrixOfEllipses(int index, QGraphicsEllipseItem\* node)  
{  
 matrixOfEllipses[index] = node;  
}  
  
void Graph::updateMatrixOfGroups(int index, QGraphicsItemGroup\* group)  
{  
 matrixOfGroups[index] = group;  
}  
  
void Graph::updateVector(std::vector<QGraphicsItemGroup\*>& vect)  
{  
 vectorOfArrows = vect;  
}  
  
void Graph::printMatrix()  
{  
 for (int i = 0; i < sizeOfMatrix; i++)  
 {  
 QString rowString;  
 for (int j = 0; j < sizeOfMatrix; j++)  
 {  
 rowString += QString::number(matrix[j][i]) + " ";  
 }  
 qDebug() << rowString;  
 }  
 qDebug() << "\n";  
}  
  
std::vector<int> Graph::dfs(int start)  
{  
 // Создание вектора для отслеживания посещенных вершин, инициализированного нулями  
 std::vector<int> visited(sizeOfMatrix, 0);  
 // Вызов рекурсивной функции dfs для запуска обхода в глубину  
 dfs(start, visited);  
 return visited;  
}  
  
void Graph::dfs(int cur, std::vector<int>& visited)  
{  
 // Помечаем текущую вершину как посещенную  
 visited[cur] = 1;  
 for (int i = 1; i < sizeOfMatrix; i++)  
 {  
 // Если есть ребро между текущей вершиной и вершиной i и вершина i ещё не посещалась  
 if (matrix[i][cur] != 0 && visited[i] == 0)  
 {  
 // Рекурсивно вызываем dfs для вершины i  
 dfs(i, visited);  
 }  
 }  
}  
  
std::vector<int> Graph::bfs(int start)  
{  
 std::vector<int> distance(sizeOfMatrix, 1e9);  
 std::queue<int> q;  
  
 // Начальная вершина имеет расстояние 0  
 distance[start] = 0;  
 q.push(start);  
  
 // Начало обхода в ширину  
 while(!q.empty())  
 {  
 // Получаем текущую вершину из очереди  
 int cur = q.front();  
 q.pop();  
  
 // Проходим по всем смежным вершинам текущей  
 for (int i = 1; i < sizeOfMatrix; i++)  
 {  
 // Если вершина ещё не посещена и есть ребро между текущей и i  
 if (distance[i] == 1e9 && matrix[i][cur] != 0)  
 {  
 // Расстояние до вершины i равно расстоянию до текущей + 1  
 distance[i] = distance[cur] + 1;  
 // Добавляем вершину i в очередь для дальнейшего обхода  
 q.push(i);  
 }  
 }  
 }  
  
 return distance;  
}  
  
std::vector<std::vector<int>> Graph::floyd()  
{  
 std::vector<std::vector<int>> dist(sizeOfMatrix - 1, std::vector<int>(sizeOfMatrix - 1, 1e9));  
  
 // Задание нулевых расстояний для диагональных элементов (вершин до самих себя)  
 for (int i = 0; i < sizeOfMatrix - 1; i++)  
 {  
 dist[i][i] = 0;  
 }  
  
 // Заполнение матрицы расстояний из матрицы смежности графа  
 for (int i = 1; i < sizeOfMatrix; i++)  
 {  
 for (int j = 1; j < sizeOfMatrix; j++)  
 {  
 if (matrix[i][j] != 0)  
 {  
 // Если между вершинами есть ребро, записываем его в матрицу расстояний  
 dist[i - 1][j - 1] = matrix[i][j];  
 }  
 }  
 }  
  
 for (int v = 0; v < sizeOfMatrix - 1; v++)  
 {  
 for (int i = 0; i < sizeOfMatrix - 1; i++)  
 {  
 for (int j = 0; j < sizeOfMatrix - 1; j++)  
 {  
 // Если существует путь через вершину v, короче, чем текущий путь от i до j  
 if (dist[i][v] != 1e9 && dist[v][j] != 1e9 && dist[i][j] > dist[i][v] + dist[v][j])  
 {  
 // Обновляем значение кратчайшего пути от i до j  
 dist[i][j] = dist[i][v] + dist[v][j];  
 }  
 }  
 }  
 }  
  
 return dist;  
}  
  
std::vector<int> Graph::dijkstra(int start)  
{  
 std::vector<int> dist(sizeOfMatrix, 1e9);  
 dist[start] = 0; // Расстояние до стартовой вершины равно 0  
 std::vector<bool> visited(sizeOfMatrix, false);  
  
 // Проходим по всем вершинам графа  
 for (int k = 0; k < sizeOfMatrix - 1; k++) // Внешний цикл повторяется sizeOfMatrix - 1 раз, так как для каждой вершины будет найдено кратчайшее расстояние до всех остальных вершин за sizeOfMatrix - 1 итераций  
 {  
 // Находим вершину с наименьшим расстоянием  
 int minDist = 1e9;  
 int nearest = -1;  
 for (int v = 0; v < sizeOfMatrix; v++)  
 {  
 if (!visited[v] && dist[v] < minDist)  
 {  
 minDist = dist[v];  
 nearest = v;  
 }  
 }  
  
 // Помечаем вершину как посещенную  
 visited[nearest] = true;  
  
 // Обновляем расстояния до смежных вершин  
 for (int v = 0; v < sizeOfMatrix; v++)  
 {  
 if (matrix[nearest][v] != 0 && !visited[v] && dist[nearest] + matrix[nearest][v] < dist[v])  
 {  
 dist[v] = dist[nearest] + matrix[nearest][v];  
 }  
 if (matrix[v][nearest] != 0 && !visited[v] && dist[nearest] + matrix[v][nearest] < dist[v])  
 {  
 dist[v] = dist[nearest] + matrix[v][nearest];  
 }  
 }  
 }  
  
 return dist;  
}  
  
std::vector<int> Graph::tsp(int start, int end)  
{  
 std::vector<int> path;  
 std::vector<bool> visited(sizeOfMatrix, false);  
  
 int curNode = start;  
 // Добавляем начальную точку в путь и отмечаем как посещенную  
 path.push\_back(curNode);  
 visited[curNode] = true;  
  
 while (path.size() < sizeOfMatrix)  
 {  
 int nextNode = -1;  
 int minDistance = INT\_MAX;  
  
 // Найдем ближайшего непосещенного соседа  
 for (int i = 1; i < sizeOfMatrix; ++i)  
 {  
 // Если вершина не посещена, есть ребро до нее и расстояние меньше минимального  
 if (!visited[i] && matrix[curNode][i] != 0 && matrix[curNode][i] < minDistance)  
 {  
 // Обновляем ближайшего соседа и минимальное расстояние  
 minDistance = matrix[curNode][i];  
 nextNode = i;  
 }  
 }  
  
 // Если не найден непосещенный сосед, вернемся к начальной точке  
 if (nextNode == -1)  
 nextNode = start;  
  
 // Переходим к следующему узлу  
 path.push\_back(nextNode);  
 visited[nextNode] = true;  
 curNode = nextNode;  
 }  
  
 // Добавляем конечную точку  
 path.push\_back(end);  
  
 return path;  
}

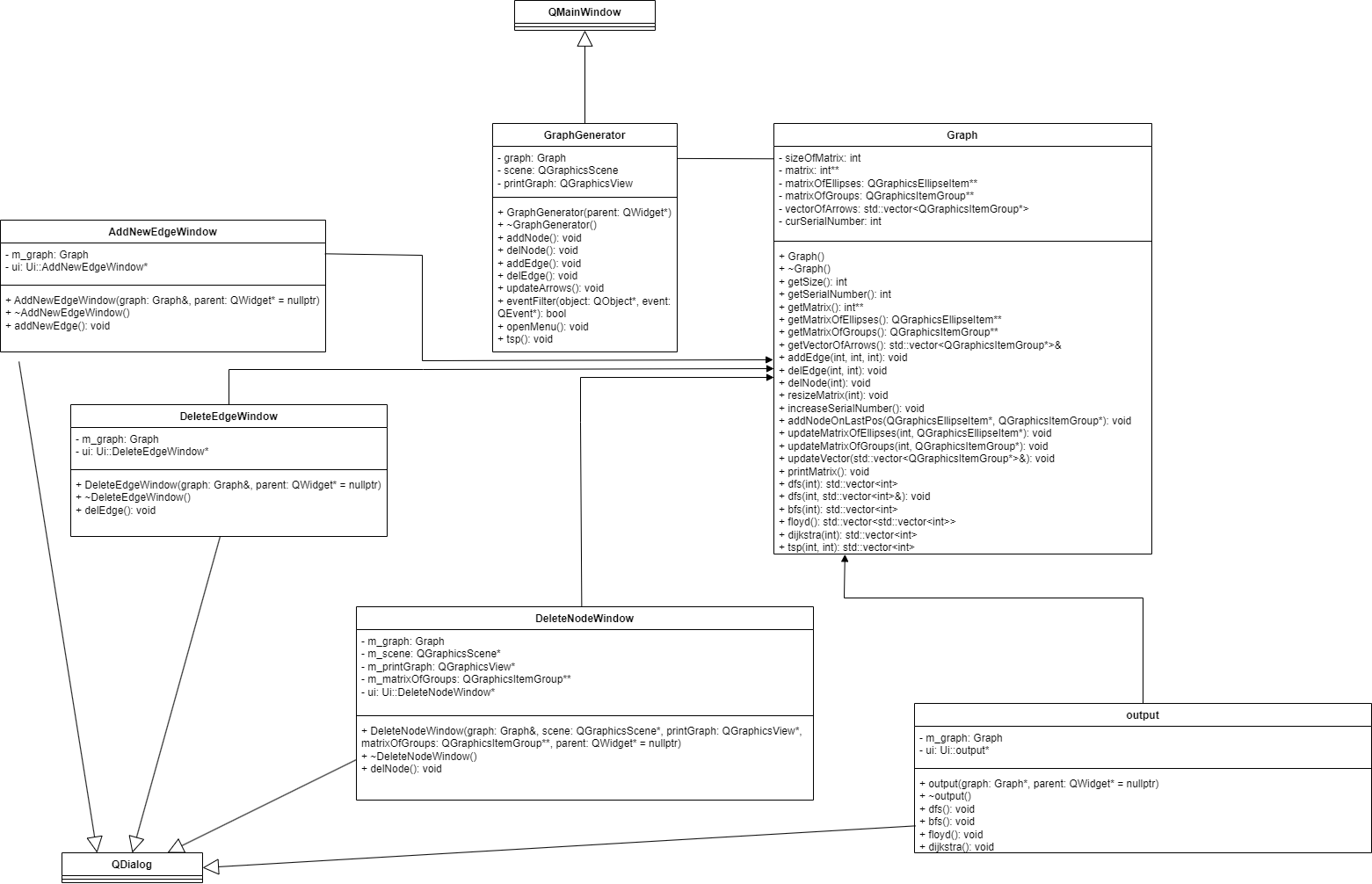
#include "graphgenerator.h"  
#include "ui\_graphgenerator.h"  
  
GraphGenerator::GraphGenerator(QWidget \*parent)  
 : QMainWindow(parent)  
 , ui(new Ui::GraphGenerator)  
{  
 ui->setupUi(this);  
  
 printGraph = ui -> printGraph;  
 scene = new QGraphicsScene;  
 printGraph -> setScene(scene);  
  
 connect(ui->addNodeBtn, &QPushButton::clicked, this, &GraphGenerator::addNode);  
 connect(ui->addEdgeBtn, &QPushButton::clicked, this, &GraphGenerator::addEdge);  
 connect(ui->updateArrowsBtn, &QPushButton::clicked, this, &GraphGenerator::updateArrows);  
 connect(ui->deleteEdgeBtn, &QPushButton::clicked, this, &GraphGenerator::delEdge);  
 connect(ui->deleteNodeBtn, &QPushButton::clicked, this, &GraphGenerator::delNode);  
 connect(ui->searchBtn, &QPushButton::clicked, this, &GraphGenerator::openMenu);  
 connect(ui->tspBtn, &QPushButton::clicked, this, &GraphGenerator::tsp);  
}  
  
GraphGenerator::~GraphGenerator()  
{  
 delete scene;  
 delete ui;  
}  
  
void GraphGenerator::addNode()  
{  
 QGraphicsEllipseItem \*node = scene->addEllipse(0, 0, 64, 64, QPen(Qt::black), QBrush(Qt::lightGray)); // добавляем эллипс на координату (0, 0) размер 64, черный контур, яркосерая заливка  
  
 // Добавляем порядковый номер:  
 QString nodeSerialNumber = QString::number(graph.getSerialNumber());  
 QGraphicsTextItem \*textItem = scene->addText(nodeSerialNumber);  
 textItem->setPos(node->boundingRect().center().x() - textItem->boundingRect().width() / 2,  
 node->boundingRect().center().y() - textItem->boundingRect().height() / 2);  
  
 // Объединяем в группу  
 QList<QGraphicsItem\*> items;  
 items << node << textItem;  
 QGraphicsItemGroup \*group = scene->createItemGroup(items); // объеденяем в группу эллипс и текст  
 group->setFlag(QGraphicsItem::ItemIsMovable, true); // даем возможность двигать группу мышкой  
 /\*  
 \* не имеет смысла, т. к. node и TextItem объеденины в группу, а дочерние элементы  
 \* группы не изменяют pos при перемещении группы  
 node->setFlag(QGraphicsItem::ItemIsMovable, true);  
 textItem->setFlag(QGraphicsItem::ItemIsMovable, true);  
 \*/  
 scene->installEventFilter(this); // обработка события, если узел перетаскивают мышкой, чтобы обновить pos  
 // нужно, чтобы рисовать стрелки между узлами  
  
 // Обновляем матрицу смежности и массивы объектов после добавления узла  
 graph.resizeMatrix(1);  
 graph.addNodeOnLastPos(node, group);  
 graph.increaseSerialNumber();  
  
 graph.printMatrix(); // вывод матрицы в консоль  
}  
  
void GraphGenerator::delNode()  
{  
 // Вызываем диалговое окно поверх основного окна  
 DeleteNodeWindow DeleteNodeWindow(graph, scene, printGraph, graph.getMatrixOfGroups());  
 DeleteNodeWindow.setModal(true);  
 DeleteNodeWindow.exec();  
  
 updateArrows(); // обновляем стрелки после удаления узла  
 updateSerialNumbers(); // обновляем порядковые номера после удаления узла  
  
 graph.printMatrix(); // вывод матрицы в консоль  
}  
  
void GraphGenerator::addEdge()  
{  
 AddNewEdgeWindow addEdgeWindow(graph);  
 addEdgeWindow.setModal(true);  
 addEdgeWindow.exec();  
  
 updateArrows();  
  
 graph.printMatrix();  
}  
  
void GraphGenerator::delEdge()  
{  
 // Вызываем диалговое окно поверх основного окна  
 DeleteEdgeWindow delEdgeWindow(graph);  
 delEdgeWindow.setModal(true);  
 delEdgeWindow.exec();  
  
 updateArrows(); // обновляем стрелки после удаления ребра  
  
 graph.printMatrix(); // выводим матрицу в консоль  
}  
  
void GraphGenerator::updateArrows()  
{  
 // Получаем объекты графа, чтобы рисовать/обновлять стрелки  
 int\*\* matrix = graph.getMatrix();  
 QGraphicsItemGroup\*\* matrixOfGroups = graph.getMatrixOfGroups();  
 QGraphicsEllipseItem\*\* matrixOfEllipses = graph.getMatrixOfEllipses();  
 std::vector<QGraphicsItemGroup\*> vectorOfArrows = graph.getVectorOfArrows();  
  
 for (int i = 0; i < vectorOfArrows.size(); i++)  
 {  
 scene->removeItem(vectorOfArrows[i]); // Удаляем группы со сцены  
 delete vectorOfArrows[i]; // Освобождаем память  
 }  
 vectorOfArrows.clear();  
  
 // Перебираем элементы матрицы смежности для рисования стрелок  
 for (int i = 1; i < graph.getSize(); i++)  
 {  
 for (int j = i; j < graph.getSize(); j++)  
 {  
 if (matrix[i][j] != 0 && matrix[i][j] == matrix[j][i] && i != j) // если ребро в обе стороны и вес равен, то рисуем линию  
 {  
 // Получаем центры эллипсов, с помощью mapToScene  
 QPointF center1 = matrixOfGroups[i]->mapToScene(matrixOfGroups[i]->boundingRect().center());  
 QPointF center2 = matrixOfGroups[j]->mapToScene(matrixOfGroups[j]->boundingRect().center());  
  
 qreal angle = qAtan2(center2.y() - center1.y(), center2.x() - center1.x()); // Находим угол между двумя центрами эллипсов  
  
 // Вычисляем новые координаты начальной и конечной точек линии с учетом укорочения на 32 пикселя  
 QPointF newStart(center1.x() + 32 \* qCos(angle), center1.y() + 32 \* qSin(angle));  
 QPointF newEnd(center2.x() - 32 \* qCos(angle), center2.y() - 32 \* qSin(angle));  
  
 QGraphicsLineItem \*line = new QGraphicsLineItem(); // линия между эллипсами  
 line->setLine(QLineF(newStart, newEnd));  
  
 QPointF textPos((center1.x() + center2.x()) / 2, (center1.y() + center2.y()) / 2); // позиция текста веса на центре линии  
 QGraphicsTextItem\* textItem = scene->addText(QString::number(matrix[i][j])); // текст веса на центре линии  
 textItem->setPos(textPos);  
 QList<QGraphicsItem\*> items;  
 items << line << textItem;  
 QGraphicsItemGroup \*group = scene->createItemGroup(items); // добавляем в группу линию и вес  
  
 vectorOfArrows.push\_back(group);  
 }  
 else if (matrix[i][j] != 0 && matrix[j][i] == 0 && i != j) // если стрелка только в одну сторону  
 {  
 QPointF center1 = matrixOfGroups[i]->mapToScene(matrixOfGroups[i]->boundingRect().center());  
 QPointF center2 = matrixOfGroups[j]->mapToScene(matrixOfGroups[j]->boundingRect().center());  
  
 qreal angle = qAtan2(center2.y() - center1.y(), center2.x() - center1.x()); // Находим угол между двумя центрами эллипсов  
  
 // Вычисляем новые координаты начальной и конечной точек линии с учетом укорочения на 32 пикселя  
 QPointF newStart(center1.x() + 32 \* qCos(angle), center1.y() + 32 \* qSin(angle));  
 QPointF newEnd(center2.x() - 32 \* qCos(angle), center2.y() - 32 \* qSin(angle));  
  
 QGraphicsLineItem \*line = new QGraphicsLineItem(); // линия между эллипсами  
 line->setLine(QLineF(newStart, newEnd));  
  
 // Создаем стрелковые концы линии  
 QPolygonF arrowHead = createArrowHead(newEnd, newStart);  
 QGraphicsPolygonItem \*arrow = new QGraphicsPolygonItem(arrowHead);  
 arrow->setBrush(Qt::black);  
 arrow->setPen(Qt::NoPen);  
  
 QPointF textPos((center1.x() + center2.x()) / 2, (center1.y() + center2.y()) / 2); // позиция текста веса на центре линии  
 QGraphicsTextItem\* textItem = scene->addText(QString::number(matrix[i][j])); // текст веса на центре линии  
 textItem->setPos(textPos);  
  
 QList<QGraphicsItem\*> items;  
 items << line << textItem << arrow;  
 QGraphicsItemGroup \*group = scene->createItemGroup(items); // добавляем в группу линию и вес  
  
 vectorOfArrows.push\_back(group);  
 }  
 else if (matrix[j][i] != 0 && matrix[i][j] == 0 && i != j) // если стрелка в другую сторону  
 {  
 QPointF center1 = matrixOfGroups[i]->mapToScene(matrixOfGroups[i]->boundingRect().center());  
 QPointF center2 = matrixOfGroups[j]->mapToScene(matrixOfGroups[j]->boundingRect().center());  
  
 qreal angle = qAtan2(center2.y() - center1.y(), center2.x() - center1.x()); // Находим угол между двумя центрами эллипсов  
  
 // Вычисляем новые координаты начальной и конечной точек линии с учетом укорочения на 32 пикселя  
 QPointF newStart(center1.x() + 32 \* qCos(angle), center1.y() + 32 \* qSin(angle));  
 QPointF newEnd(center2.x() - 32 \* qCos(angle), center2.y() - 32 \* qSin(angle));  
  
 QGraphicsLineItem \*line = new QGraphicsLineItem(); // линия между эллипсами  
 line->setLine(QLineF(newStart, newEnd));  
  
 // Создаем стрелковые концы линии  
 QPolygonF arrowHead = createArrowHead(newStart, newEnd);  
 QGraphicsPolygonItem \*arrow = new QGraphicsPolygonItem(arrowHead);  
 arrow->setBrush(Qt::black);  
 arrow->setPen(Qt::NoPen);  
  
 QPointF textPos((center1.x() + center2.x()) / 2, (center1.y() + center2.y()) / 2); // позиция текста веса на центре линии  
 QGraphicsTextItem\* textItem = scene->addText(QString::number(matrix[j][i])); // текст веса на центре линии  
 textItem->setPos(textPos);  
  
 QList<QGraphicsItem\*> items;  
 items << line << textItem << arrow;  
 QGraphicsItemGroup \*group = scene->createItemGroup(items); // добавляем в группу линию и вес  
  
 vectorOfArrows.push\_back(group);  
 }  
 else if (matrix[i][j] != 0 && matrix[j][i] != 0 && matrix[i][j] != matrix[j][i] && i != j) // если направление в обе стороны, но разные веса  
 {  
 QPointF center1 = matrixOfGroups[i]->mapToScene(matrixOfGroups[i]->boundingRect().center());  
 QPointF center2 = matrixOfGroups[j]->mapToScene(matrixOfGroups[j]->boundingRect().center());  
  
 qreal angle = qAtan2(center2.y() - center1.y(), center2.x() - center1.x()); // Находим угол между двумя центрами эллипсов  
  
 // Вычисляем новые координаты начальной и конечной точек линии с учетом укорочения на 32 пикселя  
 QPointF newStart(center1.x() + 32 \* qCos(angle), center1.y() + 32 \* qSin(angle));  
 QPointF newEnd(center2.x() + 32 \* qCos(angle + M\_PI), center2.y() + 32 \* qSin(angle + M\_PI)); // Изменил угол на angle + M\_PI  
  
 QGraphicsLineItem \*line1 = new QGraphicsLineItem(); // линия между эллипсами  
 line1->setLine(QLineF(newStart, newEnd));  
  
 // Создаем стрелковые концы линии  
 QPolygonF arrowHead1 = createArrowHead(newStart, newEnd);  
 QGraphicsPolygonItem \*arrow1 = new QGraphicsPolygonItem(arrowHead1);  
 arrow1->setBrush(Qt::black);  
 arrow1->setPen(Qt::NoPen);  
  
 // Вычисляем координаты текста на конце первой стрелки  
 QPointF textPos1(newEnd.x(), newEnd.y());  
 QGraphicsTextItem\* textItem1 = scene->addText(QString::number(matrix[j][i])); // текст веса на конце первой стрелки  
 textItem1->setPos(textPos1);  
  
 QGraphicsLineItem \*line2 = new QGraphicsLineItem(); // линия между эллипсами  
 line2->setLine(QLineF(newEnd, newStart)); // Поменял начало и конец  
  
 // Создаем стрелковые концы линии  
 QPolygonF arrowHead2 = createArrowHead(newEnd, newStart);  
 QGraphicsPolygonItem \*arrow2 = new QGraphicsPolygonItem(arrowHead2);  
 arrow2->setBrush(Qt::black);  
 arrow2->setPen(Qt::NoPen);  
  
 // Вычисляем координаты текста на конце второй стрелки  
 QPointF textPos2(newStart.x(), newStart.y());  
 QGraphicsTextItem\* textItem2 = scene->addText(QString::number(matrix[i][j])); // текст веса на конце второй стрелки  
 textItem2->setPos(textPos2);  
  
 QList<QGraphicsItem\*> items;  
 items << line1 << textItem1 << arrow1 << line2 << textItem2 << arrow2;  
 QGraphicsItemGroup \*group = scene->createItemGroup(items); // добавляем в группу линию и вес  
  
 vectorOfArrows.push\_back(group);  
 }  
 else if (i == j && matrix[i][j] != 0) // петля  
 {  
 QGraphicsEllipseItem \*ellipse = matrixOfEllipses[i]; // Получаем узел (эллипс)  
  
 // Вычисляем центр узла  
 QPointF center = ellipse->mapToScene(ellipse->boundingRect().center());  
  
 qreal radius = ellipse->boundingRect().width() / 2.0; // Радиус узла  
  
 qreal angle = 45 \* M\_PI / 180; // Угол поворота петли  
  
 // Вычисляем координаты точек для рисования петли на контуре узла  
 QPointF start(center.x() + radius \* qCos(angle), center.y() + radius \* qSin(angle));  
 QPointF end(center.x() + radius \* qCos(angle + M\_PI), center.y() + radius \* qSin(angle + M\_PI));  
  
 // Контрольные точки для создания кривой Безье  
 QPointF controlPoint1(center.x() + radius \* qCos(angle - M\_PI / 4), center.y() + radius \* qSin(angle - M\_PI / 4));  
 QPointF controlPoint2(center.x() + radius \* qCos(angle + M\_PI + M\_PI / 4), center.y() + radius \* qSin(angle + M\_PI + M\_PI / 4));  
  
 QPainterPath loopPath;  
 loopPath.moveTo(start);  
 loopPath.cubicTo(controlPoint1, controlPoint2, end);  
  
 QGraphicsPathItem \*loopItem = new QGraphicsPathItem(loopPath);  
  
 // Создаем стрелковой конец петли  
 QPolygonF loopArrowHead = createArrowHead(end, start);  
 QGraphicsPolygonItem \*arrow = new QGraphicsPolygonItem(loopArrowHead);  
 arrow->setBrush(Qt::black);  
 arrow->setPen(Qt::NoPen);  
  
 // Вычисляем координаты текста на петле  
 QPointF textPos(ellipse->mapToScene(ellipse->boundingRect().topRight())); // текст веса на петле  
  
 QGraphicsTextItem\* textItem = scene->addText(QString::number(matrix[i][j])); // текст веса на петле  
 textItem->setPos(textPos);  
  
 QList<QGraphicsItem\*> items;  
 items << loopItem << textItem << arrow;  
 QGraphicsItemGroup \*loopGroup = scene->createItemGroup(items); // добавляем в группу линию и вес  
  
 vectorOfArrows.push\_back(loopGroup);  
 }  
 }  
 }  
 graph.updateVector(vectorOfArrows);  
}  
  
bool GraphGenerator::eventFilter(QObject \*watched, QEvent \*event)  
{  
 if (watched == scene && event->type() == QEvent::GraphicsSceneMouseMove)  
 {  
 // Приводим событие к QGraphicsSceneMouseEvent для получения информации о движении мыши  
 QGraphicsSceneMouseEvent \*mouseEvent = static\_cast<QGraphicsSceneMouseEvent \*>(event);  
 // Проверяем, что произошло перемещение группы  
 if (mouseEvent->buttons() & Qt::LeftButton)  
 {  
 // Проверяем, что элемент перемещаемый  
 if (mouseEvent->lastScenePos() != mouseEvent->scenePos())  
 {  
 // Получаем элемент, на который указывает мышь  
 QGraphicsItem \*item = scene->itemAt(mouseEvent->scenePos(), QTransform());  
 // Проверяем, является ли этот элемент группой объектов  
 if (item && item->type() == QGraphicsItemGroup::Type)  
 {  
 updateArrows(); // меняем стрелки  
 return true; // Помечаем событие как обработанное  
 }  
 }  
 }  
 }  
 // Если событие не обработано, передаем его дальше для обработки другими фильтрами  
 return QObject::eventFilter(watched, event);  
}  
  
void GraphGenerator::openMenu()  
{  
 // // Вызываем диалговое окно поверх основного окна для поисков/обходов  
 output searchMenu(&graph);  
 searchMenu.setModal(true);  
 searchMenu.exec();  
}  
  
void GraphGenerator::tsp()  
{  
 // Вызываем диалговое окно поверх основного окна для решения задачи коммивояжера  
 TSP tspMenu(&graph);  
 tspMenu.setModal(true);  
 tspMenu.exec();  
}  
  
QPolygonF GraphGenerator::createArrowHead(const QPointF& startPoint, const QPointF& endPoint)  
{  
 // Добавление наконечника стрелки:  
  
 QPolygonF arrowHead;  
  
 // Рассчитываем угол между горизонтальной линией и линией между startPoint и endPoint  
 qreal angle = qAtan2(endPoint.y() - startPoint.y(), endPoint.x() - startPoint.x());  
 // Длина стрелки  
 qreal arrowLength = 10.0; // Измените это значение по вашему усмотрению  
  
 // Угол стрелки с горизонтальной линией (половина угла)  
 qreal arrowAngle = M\_PI / 6.0; // 30 градусов  
  
 // Координаты точек стрелки  
 QPointF arrowP1 = endPoint - QPointF(arrowLength \* std::cos(angle + arrowAngle), arrowLength \* std::sin(angle + arrowAngle));  
 QPointF arrowP2 = endPoint - QPointF(arrowLength \* std::cos(angle - arrowAngle), arrowLength \* std::sin(angle - arrowAngle));  
  
 // Добавляем точки в полигон стрелки  
 arrowHead << endPoint << arrowP1 << arrowP2;  
  
 return arrowHead;  
}  
  
void GraphGenerator::updateSerialNumbers()  
{  
 int size = graph.getSize();  
 QGraphicsItemGroup\*\* groups = graph.getMatrixOfGroups();  
 for (int i = 1; i < size; i++)  
 {  
 foreach (QGraphicsItem \*item, groups[i]->childItems()) // перебор контейнера  
 {  
 QGraphicsTextItem \*textItem = qgraphicsitem\_cast<QGraphicsTextItem\*>(item);  
 if (textItem)  
 {  
 textItem->setPlainText(QString::number(i)); // обновляем порядковый номер на сцене  
 }  
 }  
 }  
 scene->update();  
}

#include "graphgenerator.h"  
  
#include <QApplication>  
  
int main(int argc, char \*argv[])  
{  
 QApplication a(argc, argv);  
 GraphGenerator w;  
 w.resize(1420, 960); // устанавливаем размер окна по умолчанию 1420х960  
 w.show();  
 return a.exec();  
}

#include "searchmenu.h"  
#include "ui\_searchmenu.h"  
  
output::output(Graph\* graph, QWidget \*parent) :  
 QDialog(parent), m\_graph(graph),  
 ui(new Ui::output)  
{  
 ui->setupUi(this);  
  
 connect(ui->dfsBtn, &QPushButton::clicked, this, &output::dfs);  
 connect(ui->bfsBtn, &QPushButton::clicked, this, &output::bfs);  
 connect(ui->floydBtn, &QPushButton::clicked, this, &output::floyd);  
 connect(ui->dijkstraBtn, &QPushButton::clicked, this, &output::dijkstra);  
}  
  
output::~output()  
{  
 delete ui;  
}  
  
void output::dfs()  
{  
 // Получаем значения с lineEdit пользовательского интерфейса и записываем их в переменные типа int (изменяя тип данных)  
 QString startNode = ui->startLine->text();  
 int start = startNode.toInt();  
 std::vector<int> visited = m\_graph->dfs(start); // вызываем метод графа для обхода в глубину  
 // Вывод:  
 ui->outputDist->clear();  
 ui->outputDist->setText("Path availability:\n");  
 for (int i = 1; i < visited.size(); i++)  
 {  
 QString node = QString::number(i), isVisited = QString::number(visited[i]);  
 QString line;  
 line.append(node).append(" : ").append(isVisited).append("\n");  
 QString currentText = ui->outputDist->text();  
 QString newText = currentText + line;  
 ui->outputDist->setText(newText);  
 }  
}  
  
void output::bfs()  
{  
 // Получаем значения с lineEdit пользовательского интерфейса и записываем их в переменные типа int (изменяя тип данных)  
 QString startNode = ui->startLine->text();  
 int start = startNode.toInt();  
 std::vector<int> distance = m\_graph->bfs(start); // вызываем метод графа для обхода в ширину  
 // Вывод:  
 ui->outputDist->clear();  
 ui->outputDist->setText("Minimum number of edges from the start node:\n");  
 for (int i = 1; i < distance.size(); i++)  
 {  
 QString from = QString::number(start), to = QString::number(i), dist = QString::number(distance[i]);  
 QString line;  
 line.append(from).append(" -> ").append(to).append(" : ").append(dist).append("\n");  
 QString currentText = ui->outputDist->text();  
 QString newText = currentText + line;  
 ui->outputDist->setText(newText);  
 }  
}  
  
void output::floyd()  
{  
 std::vector<std::vector<int>> distance = m\_graph->floyd(); // вызываем метод графа для алгоритма Флойда  
 // Вывод:  
 ui->outputDist->clear();  
 ui->outputDist->setText("Shortest path from each node to each node");  
 QString text;  
 text.append(" ");  
 for (int i = 0; i < m\_graph->getSize() - 1; i++)  
 {  
 text.append(QString::number(i + 1)).append(QString(" ").repeated(4 - QString::number(i).length()));  
 }  
 text.append("\n");  
  
 for (int i = 0; i < m\_graph->getSize() - 1; i++)  
 {  
 text.append(QString::number(i + 1)).append(QString(" ").repeated(4 - QString::number(i).length()));  
 for (int j = 0; j < m\_graph->getSize() - 1; j++)  
 {  
 QString dist;  
 dist = QString::number(distance[j][i]);  
 if (distance[j][i] == 1e9)  
 {  
 dist = "\_";  
 }  
 text.append(dist).append(QString(" ").repeated(4 - dist.length()));  
 }  
 text.append("\n");  
 }  
 ui->outputDist->setText(text);  
}  
  
void output::dijkstra()  
{  
 // Получаем значения с lineEdit пользовательского интерфейса и записываем их в переменные типа int (изменяя тип данных)  
 QString startNode = ui->startLine->text();  
 int start = startNode.toInt();  
 std::vector<int> distance = m\_graph->dijkstra(start); // вызываем метод графа для алгоритма Дейкстры  
 // Вывод:  
 ui->outputDist->clear();  
 ui->outputDist->setText("Minimum number of edges (considering the minimumpath length)\nfrom the start node:\n");  
 for (int i = 1; i < distance.size(); i++)  
 {  
 QString from = QString::number(start), to = QString::number(i), dist = QString::number(distance[i]);  
 QString line;  
 line.append(from).append(" -> ").append(to).append(" : ").append(dist).append("\n");  
 QString currentText = ui->outputDist->text();  
 QString newText = currentText + line;  
 ui->outputDist->setText(newText);  
 }  
}

#include "tsp.h"  
#include "ui\_tsp.h"  
  
TSP::TSP(Graph\* graph, QWidget \*parent) :  
 QDialog(parent), m\_graph(graph),  
 ui(new Ui::TSP)  
{  
 ui->setupUi(this);  
  
 connect(ui->solveBtn, &QPushButton::clicked, this, &TSP::tsp);  
}  
  
void TSP::tsp()  
{  
 // Получаем значения с lineEdit пользовательского интерфейса и записываем их в переменные типа int (изменяя тип данных)  
 QString getFrom = ui->fromLine->text();  
 int from = getFrom.toInt();  
 QString getTo = ui->toLine->text();  
 int to = getTo.toInt();  
  
 std::vector<int> path = m\_graph->tsp(from, to); // вызов метода для решения задачи коммивояжера  
 std::reverse(path.begin(), path.end()); // "переворачиваем" вектор  
  
 QString outputText = QString::number(path[1]); // в path[0] промежуточный город (to)  
 // если для узлов from -> to невозможно решить задачу коммивояжера, то вывод будет не совсем корректный  
 for (int i = 2; i < path.size(); i++)  
 {  
 outputText += "->" + QString::number(path[i]);  
 }  
 ui->outputLabel->setText(outputText); // выводим маршрут решения  
}  
  
TSP::~TSP()  
{  
 delete ui;  
}

# UML-диаграмма



# Пример работы

# **https://www.youtube.com/watch?v=sIjwo-KzAus**