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: 09.03.04 – «Программная инженерия»

**Лабораторная работа.**

**«Графы. Алгоритм Дейкстры. Алгоритм Флойда и задача коммивояжёра»**

Выполнил студент гр. РИС-24-1б

Чижов Денис Николаевич

Проверил:

Доц. Каф. ИТАС 

Ольга Андреевна Полякова

(оценка) (подпись)

(дата)

г. Пермь, 2025

**Задача:**

Реализовать алгоритмы для собственного варианта графа, имеющего не менее 6 вершин. Если вершин меньше 6, то нужно самому добавить нужное кол-во вершин.

Алгоритмы:

1. Обход в ширину.

2. Обход в глубину.

3. Алгоритм Дейкстры.

4. Алгоритм Флойда

Требования:

1. Пользовательский интерфейс на усмотрение разработчика с условием кроссплатформенности (поощряется использование Qt или иных фреймворков)

2. Визуализация графа с использованием любой доступной графической библиотеки (SFML, SDL, OpenGL и подобных)

3. Реализованные алгоритмы должны справляться как с графом, представленным в задании варианта, так и с другими на усмотрение проверяющего.

4. Необходимо реализовать функции для редактирования графа:

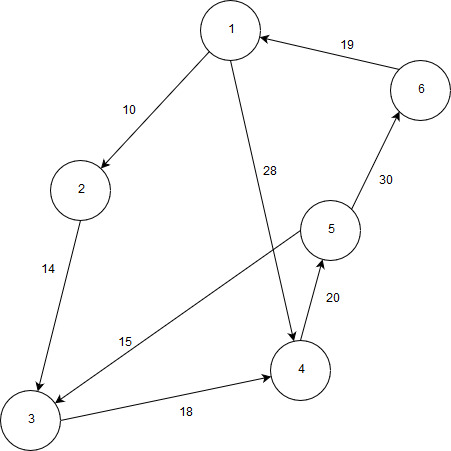
- Создание новой вершины.

- Удаление вершины.

- Добавление и удаление ребра.

- Редактирование весов ребер.

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



**Анализ:**

**1. Основные задачи проекта**

Необходимо создать кроссплатформенное приложение с графическим интерфейсом, которое позволит:

* Отображать графы (узлы, рёбра, весовые коэффициенты).
* Редактировать графы (добавлять/удалять вершины и рёбра).
* Запускать 4 алгоритма обработки графов:
  + **BFS** – обход в ширину.
  + **DFS** – обход в глубину.
  + **Алгоритм Дейкстры** – поиск кратчайшего пути от заданной вершины.
  + **Алгоритм Флойда-Уоршелла** – расчёт кратчайших путей между всеми вершинами.

**2. Техническая реализация**

**Кроссплатформенность:**

* Используется фреймворк **Qt** (QGraphicsView и QML) для совместимости с Windows, Linux и macOS.

**Визуализация графа:**

* Узлы отображаются с помощью **QGraphicsEllipseItem**.
* Рёбра рисуются как **QGraphicsLineItem** с подписями весов (**QGraphicsTextItem**).

**Хранение данных:**

* Основная структура – **матрица смежности** (QVector<QVector<int>>), удобная для алгоритмов Дейкстры и Флойда-Уоршелла.

**Алгоритмы:**

* **BFS** – реализован через очередь (QQueue).
* **DFS** – на основе стека (QStack) или рекурсии.
* **Дейкстра** – с приоритетной очередью (priority\_queue).
* **Флойд-Уоршелл** – метод динамического программирования.

**Редактирование графа:**

* Добавлены методы:
  + addVertex(), removeVertex() – управление вершинами.
  + addEdge(), setWeight() – изменение рёбер.
* Для реактивности интерфейса задействована система **сигналов и слотов Qt**.

**3. Потенциальные сложности и решения**

**1. Отображение весов рёбер:**

* Решение: добавление текстовых элементов (**QGraphicsTextItem**) рядом с линиями.

**2. Удаление вершин и пересчёт структуры:**

* Решение: автоматическое обновление матрицы смежности при изменении графа.

**3. Оптимизация алгоритмов:**

* Для Флойда-Уоршелла используется значение INT\_MAX / 2 вместо "бесконечности", чтобы избежать переполнения.

**4. Визуализация работы алгоритмов:**

* Подсвечиваются текущие вершины (setBrush(Qt::yellow)) и рёбра (setPen(Qt::red)) для наглядности.

**4. Итоговые возможности и выбор технологий**

**Функционал приложения:**

* Интерактивное создание и изменение графов.
* Анимация пошагового выполнения BFS/DFS.
* Расчёт кратчайших путей (Дейкстра, Флойд-Уоршелл).
* Настройка весов рёбер.

**Почему Qt + C++?**

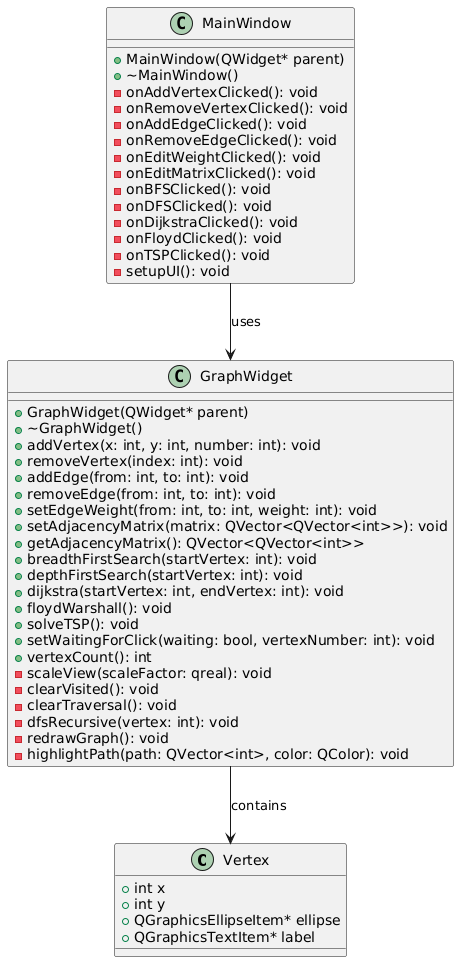
* **Производительность:** C++ обеспечивает высокую скорость работы с графами.
* **Гибкость:** Qt предоставляет готовые инструменты для графики и кроссплатформенного GUI.

**Альтернативы:**

* **Python (PyQt + NetworkX)** – проще в разработке, но менее производителен.
* **JavaScript (D3.js)** – мощная визуализация, но нет нативного кроссплатформенного GUI.

**Вывод:**  
Приложение сочетает интуитивно понятный интерфейс, широкую функциональность и поддержку нескольких ОС благодаря Qt и эффективным алгоритмам на C++.

**UML диаграмма:**

****

**Код:**

graphwidged.h:

#ifndef GRAPHWIDGET\_H

#define GRAPHWIDGET\_H

#include <QGraphicsView>

#include <QVector>

#include <QMouseEvent>

#include <QGraphicsTextItem>

#include <QGraphicsEllipseItem>

#include <QGraphicsLineItem>

#include <QQueue>

#include <QStack>

struct Vertex {

    int x, y;

    QGraphicsEllipseItem\* ellipse = nullptr;

    QGraphicsTextItem\* label = nullptr;

};

class GraphWidget : public QGraphicsView

{

    Q\_OBJECT

public:

    explicit GraphWidget(QWidget \*parent = nullptr);

    ~GraphWidget();

    // Основные операции с графом

    void addVertex(int x, int y, int number);

    void removeVertex(int index);

    void addEdge(int from, int to);

    void removeEdge(int from, int to);

    void setEdgeWeight(int from, int to, int weight);

    void setAdjacencyMatrix(const QVector<QVector<int>> &matrix);

    QVector<QVector<int>> getAdjacencyMatrix() const;

    // Алгоритмы

    void breadthFirstSearch(int startVertex);

    void depthFirstSearch(int startVertex);

    void dijkstra(int startVertex, int endVertex);

    void floydWarshall();

    void solveTSP();

    // Вспомогательные методы

    void setWaitingForClick(bool waiting, int vertexNumber);

    int vertexCount() const { return vertices.size(); }

signals:

    void vertexAdded(int index);

    void vertexRemoved(int index);

    void edgeAdded(int from, int to);

    void edgeRemoved(int from, int to);

    void edgeWeightChanged(int from, int to, int weight);

    void adjacencyMatrixChanged(const QVector<QVector<int>> &matrix);

protected:

    void drawBackground(QPainter \*painter, const QRectF &rect) override;

    void wheelEvent(QWheelEvent \*event) override;

    void mousePressEvent(QMouseEvent \*event) override;

private:

    void scaleView(qreal scaleFactor);

    void clearVisited();

    void clearTraversal();

    void dfsRecursive(int vertex);

    void redrawGraph();

    void highlightPath(const QVector<int>& path, const QColor& color);

    QVector<Vertex> vertices;

    QVector<QVector<int>> adjacencyMatrix;

    QVector<QVector<int>> edgeWeights;

    QVector<bool> visited;

    QVector<int> bfsTraversal;

    QVector<int> dfsTraversal;

    bool m\_waitingForClick = false;

    int m\_nextVertexNumber = -1;

    QVector<QGraphicsLineItem\*> edgeLines;

    QVector<QGraphicsTextItem\*> edgeLabels;

};

#endif // GRAPHWIDGET\_H

graphwidged.cpp:

#include "graphwidget.h"

#include <QGraphicsScene>

#include <QPainter>

#include <QWheelEvent>

#include <QMessageBox>

#include <QInputDialog>

#include <QColorDialog>

#include <climits>

#include <algorithm>

#include <QDebug>

GraphWidget::GraphWidget(QWidget \*parent)

    : QGraphicsView(parent)

{

    QGraphicsScene \*scene = new QGraphicsScene(this);

    scene->setSceneRect(0, 0, 800, 600);

    setScene(scene);

    setRenderHint(QPainter::Antialiasing);

    setDragMode(QGraphicsView::ScrollHandDrag);

    setCacheMode(QGraphicsView::CacheBackground);

    setViewportUpdateMode(QGraphicsView::BoundingRectViewportUpdate);

    setTransformationAnchor(QGraphicsView::AnchorUnderMouse);

}

GraphWidget::~GraphWidget()

{

    clearTraversal();

}

void GraphWidget::addVertex(int x, int y, int number) {

    if (number < 0) {

        QMessageBox::warning(nullptr, "Ошибка", "Номер вершины не может быть отрицательным");

        return;

    }

    // Проверяем, существует ли уже вершина с таким номером

    if (number < vertices.size() && vertices[number].ellipse != nullptr) {

        QMessageBox::warning(nullptr, "Ошибка", "Вершина с таким номером уже существует");

        return;

    }

    // Увеличиваем размер вектора вершин, если нужно

    if (number >= vertices.size()) {

        vertices.resize(number + 1);

        // Запоминаем старый размер матрицы

        int oldSize = adjacencyMatrix.size();

        // Расширяем матрицы смежности и весов

        adjacencyMatrix.resize(number + 1);

        edgeWeights.resize(number + 1);

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

            // Если строка новая, создаём её

            if (i >= oldSize) {

                adjacencyMatrix[i] = QVector<int>(number + 1, 0);

                edgeWeights[i] = QVector<int>(number + 1, 0);

            }

            // Если строка старая, расширяем её

            else {

                adjacencyMatrix[i].resize(number + 1);

                edgeWeights[i].resize(number + 1);

                // Инициализируем новые элементы в старых строках

                for (int j = oldSize; j <= number; ++j) {

                    adjacencyMatrix[i][j] = 0;

                    edgeWeights[i][j] = 0;

                }

            }

        }

    }

    // Создаём вершину

    Vertex &v = vertices[number];

    v.x = x;

    v.y = y;

    // Создаём графические элементы

    v.ellipse = scene()->addEllipse(v.x - 15, v.y - 15, 30, 30,

                                    QPen(Qt::black), QBrush(Qt::white));

    v.label = scene()->addText(QString::number(number));

    QFont font = v.label->font();

    font.setPointSize(12);

    v.label->setFont(font);

    v.label->setDefaultTextColor(Qt::black); // Устанавливаем черный цвет текста

    QRectF textRect = v.label->boundingRect();

    v.label->setPos(v.x - textRect.width() / 2, v.y - textRect.height() / 2);

    emit vertexAdded(number);

}

void GraphWidget::removeVertex(int index)

{

    if (index < 0 || index >= vertices.size() || vertices[index].ellipse == nullptr) {

        QMessageBox::warning(nullptr, "Ошибка", "Неверный индекс вершины");

        return;

    }

    // Удаляем графические элементы вершины

    scene()->removeItem(vertices[index].ellipse);

    scene()->removeItem(vertices[index].label);

    delete vertices[index].ellipse;

    delete vertices[index].label;

    // Удаляем все рёбра, связанные с этой вершиной

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

        if (adjacencyMatrix[index][i] != 0) {

            adjacencyMatrix[index][i] = 0;

            adjacencyMatrix[i][index] = 0;

            edgeWeights[index][i] = 0;

            edgeWeights[i][index] = 0;

        }

    }

    // Удаляем вершину из вектора

    vertices.remove(index);

    // Удаляем строку и столбец из матриц смежности и весов

    adjacencyMatrix.remove(index);

    edgeWeights.remove(index);

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

        adjacencyMatrix[i].remove(index);

        edgeWeights[i].remove(index);

    }

    // Обновляем индексы в оставшихся вершинах

    for (int i = index; i < vertices.size(); ++i) {

        vertices[i].label->setPlainText(QString::number(i));

    }

    // Перерисовываем граф

    redrawGraph();

    emit vertexRemoved(index);

}

void GraphWidget::addEdge(int from, int to)

{

    if (from < 0 || to < 0 || from >= vertices.size() || to >= vertices.size() ||

        vertices[from].ellipse == nullptr || vertices[to].ellipse == nullptr) {

        QMessageBox::warning(nullptr, "Ошибка", "Неверные индексы вершин");

        return;

    }

    if (from == to) {

        QMessageBox::warning(nullptr, "Ошибка", "Нельзя создать петлю");

        return;

    }

    adjacencyMatrix[from][to] = 1;

    adjacencyMatrix[to][from] = 1;

    // Устанавливаем вес по умолчанию

    edgeWeights[from][to] = 1;

    edgeWeights[to][from] = 1;

    redrawGraph();

    emit edgeAdded(from, to);

}

void GraphWidget::removeEdge(int from, int to)

{

    if (from < 0 || to < 0 || from >= vertices.size() || to >= vertices.size()) {

        QMessageBox::warning(nullptr, "Ошибка", "Неверные индексы вершин");

        return;

    }

    if (adjacencyMatrix[from][to] == 0) {

        QMessageBox::warning(nullptr, "Ошибка", "Ребро не существует");

        return;

    }

    adjacencyMatrix[from][to] = 0;

    adjacencyMatrix[to][from] = 0;

    edgeWeights[from][to] = 0;

    edgeWeights[to][from] = 0;

    redrawGraph();

    emit edgeRemoved(from, to);

}

void GraphWidget::setEdgeWeight(int from, int to, int weight)

{

    if (from < 0 || to < 0 || from >= vertices.size() || to >= vertices.size()) {

        QMessageBox::warning(nullptr, "Ошибка", "Неверные индексы вершин");

        return;

    }

    if (adjacencyMatrix[from][to] == 0) {

        QMessageBox::warning(nullptr, "Ошибка", "Ребро не существует");

        return;

    }

    if (weight <= 0) {

        QMessageBox::warning(nullptr, "Ошибка", "Вес должен быть положительным");

        return;

    }

    edgeWeights[from][to] = weight;

    edgeWeights[to][from] = weight;

    redrawGraph();

    emit edgeWeightChanged(from, to, weight);

}

void GraphWidget::setAdjacencyMatrix(const QVector<QVector<int>> &matrix)

{

    if (matrix.size() != vertices.size()) {

        QMessageBox::warning(nullptr, "Ошибка", "Размер матрицы не соответствует количеству вершин");

        return;

    }

    adjacencyMatrix = matrix;

    edgeWeights.resize(vertices.size());

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

        edgeWeights[i].resize(vertices.size());

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

            edgeWeights[i][j] = (adjacencyMatrix[i][j] != 0) ? 1 : 0;

        }

    }

    redrawGraph();

    emit adjacencyMatrixChanged(adjacencyMatrix);

}

QVector<QVector<int>> GraphWidget::getAdjacencyMatrix() const

{

    return adjacencyMatrix;

}

void GraphWidget::redrawGraph() {

    // Очищаем только рёбра и их подписи

    for (auto line : edgeLines) {

        scene()->removeItem(line);

        delete line;

    }

    edgeLines.clear();

    for (auto label : edgeLabels) {

        scene()->removeItem(label);

        delete label;

    }

    edgeLabels.clear();

    // Радиус вершин (15 по умолчанию, так как размер 30x30)

    const qreal vertexRadius = 15.0;

    // Рисуем рёбра

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

        if (vertices[i].ellipse == nullptr) continue;

        for (int j = i + 1; j < vertices.size(); ++j) {

            if (vertices[j].ellipse == nullptr) continue;

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

                QPen pen(Qt::black);

                if (edgeWeights[i][j] > 1) {

                    pen.setWidth(2);

                }

                // Координаты центров вершин

                QPointF center1(vertices[i].x, vertices[i].y);

                QPointF center2(vertices[j].x, vertices[j].y);

                // Вектор направления линии

                QPointF direction = center2 - center1;

                qreal length = sqrt(direction.x() \* direction.x() + direction.y() \* direction.y());

                // Нормализуем вектор

                if (length > 0) {

                    direction /= length;

                }

                // Вычисляем точки начала и конца линии на границах кругов

                QPointF startPoint = center1 + direction \* vertexRadius;

                QPointF endPoint = center2 - direction \* vertexRadius;

                QGraphicsLineItem\* edgeLine = scene()->addLine(

                    QLineF(startPoint, endPoint), pen);

                edgeLines.append(edgeLine);

                // Добавляем подпись веса

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

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

                    weightText->setDefaultTextColor(Qt::black);

                    // Позиция надписи - середина линии между границами кругов

                    QPointF midPoint = (startPoint + endPoint) / 2;

                    weightText->setPos(midPoint);

                    edgeLabels.append(weightText);

                }

            }

        }

    }

}

void GraphWidget::breadthFirstSearch(int startVertex)

{

    if (startVertex < 0 || startVertex >= vertices.size() || vertices[startVertex].ellipse == nullptr) {

        QMessageBox::warning(nullptr, "Ошибка", "Неверная начальная вершина");

        return;

    }

    clearVisited();

    clearTraversal();

    QQueue<int> queue;

    queue.enqueue(startVertex);

    visited[startVertex] = true;

    QString order = "BFS: ";

    while (!queue.isEmpty()) {

        int v = queue.dequeue();

        bfsTraversal.append(v);

        order += QString::number(v) + " ";

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

            if (adjacencyMatrix[v][i] != 0 && !visited[i]) {

                visited[i] = true;

                queue.enqueue(i);

            }

        }

    }

    QMessageBox::information(nullptr, "Обход в ширину", order);

}

void GraphWidget::depthFirstSearch(int startVertex)

{

    if (startVertex < 0 || startVertex >= vertices.size() || vertices[startVertex].ellipse == nullptr) {

        QMessageBox::warning(nullptr, "Ошибка", "Неверная начальная вершина");

        return;

    }

    clearVisited();

    clearTraversal();

    dfsRecursive(startVertex);

    QString order = "DFS: ";

    for (int v : dfsTraversal) {

        order += QString::number(v) + " ";

    }

    QMessageBox::information(nullptr, "Обход в глубину", order);

}

void GraphWidget::dfsRecursive(int vertex)

{

    visited[vertex] = true;

    dfsTraversal.append(vertex);

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

        if (adjacencyMatrix[vertex][i] != 0 && !visited[i]) {

            dfsRecursive(i);

        }

    }

}

void GraphWidget::dijkstra(int startVertex, int endVertex)

{

    if (startVertex < 0 || startVertex >= vertices.size() || vertices[startVertex].ellipse == nullptr ||

        endVertex < 0 || endVertex >= vertices.size() || vertices[endVertex].ellipse == nullptr) {

        QMessageBox::warning(nullptr, "Ошибка", "Неверные вершины");

        return;

    }

    int n = vertices.size();

    QVector<int> dist(n, INT\_MAX);

    QVector<int> prev(n, -1);

    QVector<bool> visited(n, false);

    dist[startVertex] = 0;

    for (int i = 0; i < n; ++i) {

        int v = -1;

        for (int j = 0; j < n; ++j) {

            if (!visited[j] && (v == -1 || dist[j] < dist[v])) {

                v = j;

            }

        }

        if (v == -1 || dist[v] == INT\_MAX) {

            break;

        }

        visited[v] = true;

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

            if (adjacencyMatrix[v][to] != 0) {

                int alt = dist[v] + edgeWeights[v][to];

                if (alt < dist[to]) {

                    dist[to] = alt;

                    prev[to] = v;

                }

            }

        }

    }

    if (dist[endVertex] == INT\_MAX) {

        QMessageBox::information(nullptr, "Алгоритм Дейкстры", "Путь не существует");

        return;

    }

    QVector<int> path;

    for (int v = endVertex; v != -1; v = prev[v]) {

        path.prepend(v);

    }

    QString pathStr;

    for (int v : path) {

        pathStr += QString::number(v) + " ";

    }

    QMessageBox::information(nullptr, "Алгоритм Дейкстры",

                             QString("Кратчайшее расстояние: %1\nПуть: %2").arg(dist[endVertex]).arg(pathStr));

    highlightPath(path, Qt::red);

}

void GraphWidget::floydWarshall()

{

    int n = vertices.size();

    if (n == 0) return;

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

    // Инициализация матрицы расстояний

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            if (i == j) {

                dist[i][j] = 0;

            } else if (adjacencyMatrix[i][j] != 0) {

                dist[i][j] = edgeWeights[i][j];

            } else {

                dist[i][j] = INT\_MAX / 2; // Чтобы избежать переполнения

            }

        }

    }

    // Алгоритм Флойда-Уоршелла

    for (int k = 0; k < n; ++k) {

        for (int i = 0; i < n; ++i) {

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

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

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

                }

            }

        }

    }

    // Формирование результата

    QString result = "Матрица кратчайших расстояний:\n";

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            if (dist[i][j] == INT\_MAX / 2) {

                result += "∞ ";

            } else {

                result += QString::number(dist[i][j]) + " ";

            }

        }

        result += "\n";

    }

    QMessageBox::information(nullptr, "Алгоритм Флойда-Уоршелла", result);

}

void GraphWidget::solveTSP()

{

    int n = vertices.size();

    if (n < 2) {

        QMessageBox::warning(nullptr, "Ошибка", "Недостаточно вершин для задачи коммивояжера");

        return;

    }

    // Проверяем, является ли граф полным

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

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

                QMessageBox::warning(nullptr, "Ошибка", "Граф должен быть полным для задачи коммивояжера");

                return;

            }

        }

    }

    QVector<int> vertexIndices;

    for (int i = 0; i < n; ++i) {

        vertexIndices.append(i);

    }

    QVector<int> bestPath;

    int minCost = INT\_MAX;

    // Генерация всех перестановок вершин

    do {

        int currentCost = 0;

        bool valid = true;

        // Вычисляем стоимость текущего пути

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

            int from = vertexIndices[i];

            int to = vertexIndices[i + 1];

            currentCost += edgeWeights[from][to];

        }

        // Добавляем возвращение в начальную вершину

        currentCost += edgeWeights[vertexIndices[n - 1]][vertexIndices[0]];

        if (currentCost < minCost) {

            minCost = currentCost;

            bestPath = vertexIndices;

        }

    } while (std::next\_permutation(vertexIndices.begin() + 1, vertexIndices.end()));

    // Формируем результат

    QString result = "Оптимальный маршрут: ";

    for (int v : bestPath) {

        result += QString::number(v) + " -> ";

    }

    result += QString::number(bestPath[0]); // Возвращаемся в начало

    result += QString("\nОбщая стоимость: %1").arg(minCost);

    QMessageBox::information(nullptr, "Задача коммивояжера", result);

    // Подсвечиваем путь

    bestPath.append(bestPath[0]); // Для замыкания цикла

    highlightPath(bestPath, Qt::blue);

}

void GraphWidget::highlightPath(const QVector<int>& path, const QColor& color) {

    if (path.size() < 2) return;

    const qreal vertexRadius = 15.0;

    for (int i = 0; i < path.size() - 1; ++i) {

        int from = path[i];

        int to = path[i + 1];

        if (from < 0 || to < 0 || from >= vertices.size() || to >= vertices.size() ||

            vertices[from].ellipse == nullptr || vertices[to].ellipse == nullptr) {

            continue;

        }

        // Координаты центров вершин

        QPointF center1(vertices[from].x, vertices[from].y);

        QPointF center2(vertices[to].x, vertices[to].y);

        // Вектор направления линии

        QPointF direction = center2 - center1;

        qreal length = sqrt(direction.x() \* direction.x() + direction.y() \* direction.y());

        if (length > 0) {

            direction /= length;

        }

        // Вычисляем точки начала и конца линии на границах кругов

        QPointF startPoint = center1 + direction \* vertexRadius;

        QPointF endPoint = center2 - direction \* vertexRadius;

        // Подсвечиваем ребро

        QPen pen(color);

        pen.setWidth(3);

        scene()->addLine(QLineF(startPoint, endPoint), pen);

        // Подсвечиваем вершины

        vertices[from].ellipse->setBrush(QBrush(color.lighter(150)));

        vertices[to].ellipse->setBrush(QBrush(color.lighter(150)));

    }

}

void GraphWidget::setWaitingForClick(bool waiting, int vertexNumber)

{

    m\_waitingForClick = waiting;

    m\_nextVertexNumber = vertexNumber;

}

void GraphWidget::mousePressEvent(QMouseEvent \*event)

{

    if (m\_waitingForClick) {

        QPointF pt = mapToScene(event->pos());

        addVertex(pt.x(), pt.y(), m\_nextVertexNumber);

        m\_waitingForClick = false;

        m\_nextVertexNumber = -1;

        return;

    }

    QGraphicsView::mousePressEvent(event);

}

void GraphWidget::wheelEvent(QWheelEvent \*event)

{

    scaleView(pow((double)2, -event->angleDelta().y() / 240.0));

}

void GraphWidget::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 GraphWidget::clearVisited()

{

    visited.clear();

    visited.resize(vertices.size());

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

        visited[i] = false;

    }

}

void GraphWidget::clearTraversal()

{

    bfsTraversal.clear();

    dfsTraversal.clear();

}

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

{

    Q\_UNUSED(rect);

    painter->fillRect(sceneRect(), Qt::white);

}

mainwindow.h:

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include <QVector>

QT\_BEGIN\_NAMESPACE

namespace Ui { class MainWindow; }

QT\_END\_NAMESPACE

class GraphWidget;

class MainWindow : public QMainWindow

{

    Q\_OBJECT

public:

    explicit MainWindow(QWidget \*parent = nullptr);

    ~MainWindow();

private slots:

    void onAddVertexClicked();

    void onRemoveVertexClicked();

    void onAddEdgeClicked();

    void onRemoveEdgeClicked();

    void onEditWeightClicked();

    void onEditMatrixClicked();

    void onBFSClicked();

    void onDFSClicked();

    void onDijkstraClicked();

    void onFloydClicked();

    void onTSPClicked();

private:

    Ui::MainWindow \*ui;

    GraphWidget \*graphWidget;

    void setupUI();

};

#endif // MAINWINDOW\_H

mainwindow.cpp:

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include "graphwidget.h"

#include <QPushButton>

#include <QInputDialog>

#include <QVBoxLayout>

#include <QHBoxLayout>

#include <QMessageBox>

#include <QDebug>

MainWindow::MainWindow(QWidget \*parent)

    : QMainWindow(parent)

    , ui(new Ui::MainWindow)

{

    ui->setupUi(this);

    setupUI();

}

MainWindow::~MainWindow()

{

    delete ui;

}

void MainWindow::setupUI()

{

    graphWidget = new GraphWidget(this);

    // Создаем кнопки

    QPushButton \*addVertexButton = new QPushButton("Добавить вершину", this);

    QPushButton \*removeVertexButton = new QPushButton("Удалить вершину", this);

    QPushButton \*addEdgeButton = new QPushButton("Добавить ребро", this);

    QPushButton \*removeEdgeButton = new QPushButton("Удалить ребро", this);

    QPushButton \*editWeightButton = new QPushButton("Изменить вес ребра", this);

    QPushButton \*editMatrixButton = new QPushButton("Редактировать матрицу смежности", this);

    QPushButton \*bfsButton = new QPushButton("Обход в ширину (BFS)", this);

    QPushButton \*dfsButton = new QPushButton("Обход в глубину (DFS)", this);

    QPushButton \*dijkstraButton = new QPushButton("Алгоритм Дейкстры", this);

    QPushButton \*floydButton = new QPushButton("Алгоритм Флойда", this);

    QPushButton \*tspButton = new QPushButton("Решить TSP", this);

    // Подключаем кнопки к слотам

    connect(addVertexButton, &QPushButton::clicked, this, &MainWindow::onAddVertexClicked);

    connect(removeVertexButton, &QPushButton::clicked, this, &MainWindow::onRemoveVertexClicked);

    connect(addEdgeButton, &QPushButton::clicked, this, &MainWindow::onAddEdgeClicked);

    connect(removeEdgeButton, &QPushButton::clicked, this, &MainWindow::onRemoveEdgeClicked);

    connect(editWeightButton, &QPushButton::clicked, this, &MainWindow::onEditWeightClicked);

    connect(editMatrixButton, &QPushButton::clicked, this, &MainWindow::onEditMatrixClicked);

    connect(bfsButton, &QPushButton::clicked, this, &MainWindow::onBFSClicked);

    connect(dfsButton, &QPushButton::clicked, this, &MainWindow::onDFSClicked);

    connect(dijkstraButton, &QPushButton::clicked, this, &MainWindow::onDijkstraClicked);

    connect(floydButton, &QPushButton::clicked, this, &MainWindow::onFloydClicked);

    connect(tspButton, &QPushButton::clicked, this, &MainWindow::onTSPClicked);

    // Создаем layout для кнопок

    QHBoxLayout \*buttonLayout = new QHBoxLayout();

    buttonLayout->addWidget(addVertexButton);

    buttonLayout->addWidget(removeVertexButton);

    buttonLayout->addWidget(addEdgeButton);

    buttonLayout->addWidget(removeEdgeButton);

    buttonLayout->addWidget(editWeightButton);

    buttonLayout->addWidget(editMatrixButton);

    QHBoxLayout \*algorithmLayout = new QHBoxLayout();

    algorithmLayout->addWidget(bfsButton);

    algorithmLayout->addWidget(dfsButton);

    algorithmLayout->addWidget(dijkstraButton);

    algorithmLayout->addWidget(floydButton);

    algorithmLayout->addWidget(tspButton);

    // Основной layout

    QVBoxLayout \*mainLayout = new QVBoxLayout();

    mainLayout->addWidget(graphWidget);

    mainLayout->addLayout(buttonLayout);

    mainLayout->addLayout(algorithmLayout);

    // Центральный виджет

    QWidget \*centralWidget = new QWidget(this);

    centralWidget->setLayout(mainLayout);

    setCentralWidget(centralWidget);

    // Настройки окна

    setWindowTitle("Графы: алгоритмы и визуализация");

    resize(1000, 800);

}

void MainWindow::onAddVertexClicked()

{

    bool ok;

    int vertexNumber = QInputDialog::getInt(this, "Добавить вершину",

                                            "Введите номер вершины:",

                                            graphWidget->vertexCount(), 0, 1000, 1, &ok);

    if (ok) {

        graphWidget->setWaitingForClick(true, vertexNumber);

    }

}

void MainWindow::onRemoveVertexClicked()

{

    bool ok;

    int vertexNumber = QInputDialog::getInt(this, "Удалить вершину",

                                            "Введите номер вершины для удаления:",

                                            0, 0, graphWidget->vertexCount() - 1, 1, &ok);

    if (ok) {

        graphWidget->removeVertex(vertexNumber);

    }

}

void MainWindow::onAddEdgeClicked()

{

    bool ok1, ok2;

    int from = QInputDialog::getInt(this, "Добавить ребро", "Первая вершина:",

                                    0, 0, graphWidget->vertexCount() - 1, 1, &ok1);

    int to = QInputDialog::getInt(this, "Добавить ребро", "Вторая вершина:",

                                  0, 0, graphWidget->vertexCount() - 1, 1, &ok2);

    if (ok1 && ok2) {

        graphWidget->addEdge(from, to);

    }

}

void MainWindow::onRemoveEdgeClicked()

{

    bool ok1, ok2;

    int from = QInputDialog::getInt(this, "Удалить ребро", "Первая вершина:",

                                    0, 0, graphWidget->vertexCount() - 1, 1, &ok1);

    int to = QInputDialog::getInt(this, "Удалить ребро", "Вторая вершина:",

                                  0, 0, graphWidget->vertexCount() - 1, 1, &ok2);

    if (ok1 && ok2) {

        graphWidget->removeEdge(from, to);

    }

}

void MainWindow::onEditWeightClicked()

{

    bool ok1, ok2, ok3;

    int from = QInputDialog::getInt(this, "Изменить вес ребра", "Первая вершина:",

                                    0, 0, graphWidget->vertexCount() - 1, 1, &ok1);

    int to = QInputDialog::getInt(this, "Изменить вес ребра", "Вторая вершина:",

                                  0, 0, graphWidget->vertexCount() - 1, 1, &ok2);

    int weight = QInputDialog::getInt(this, "Изменить вес ребра", "Новый вес:",

                                      1, 1, 10000, 1, &ok3);

    if (ok1 && ok2 && ok3) {

        graphWidget->setEdgeWeight(from, to, weight);

    }

}

void MainWindow::onEditMatrixClicked()

{

    int n = graphWidget->vertexCount();

    if (n == 0) {

        QMessageBox::warning(this, "Ошибка", "Нет вершин для редактирования матрицы");

        return;

    }

    QVector<QVector<int>> matrix = graphWidget->getAdjacencyMatrix();

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            bool ok;

            int val = QInputDialog::getInt(this, "Редактирование матрицы",

                                           QString("Матрица[%1][%2]:").arg(i).arg(j),

                                           matrix[i][j], 0, 1, 1, &ok);

            if (!ok) return;

            matrix[i][j] = val;

        }

    }

    graphWidget->setAdjacencyMatrix(matrix);

}

void MainWindow::onBFSClicked()

{

    bool ok;

    int start = QInputDialog::getInt(this, "BFS", "Начальная вершина:",

                                     0, 0, graphWidget->vertexCount() - 1, 1, &ok);

    if (ok) {

        graphWidget->breadthFirstSearch(start);

    }

}

void MainWindow::onDFSClicked()

{

    bool ok;

    int start = QInputDialog::getInt(this, "DFS", "Начальная вершина:",

                                     0, 0, graphWidget->vertexCount() - 1, 1, &ok);

    if (ok) {

        graphWidget->depthFirstSearch(start);

    }

}

void MainWindow::onDijkstraClicked()

{

    bool ok1, ok2;

    int from = QInputDialog::getInt(this, "Дейкстра", "Откуда:",

                                    0, 0, graphWidget->vertexCount() - 1, 1, &ok1);

    int to = QInputDialog::getInt(this, "Дейкстра", "Куда:",

                                  0, 0, graphWidget->vertexCount() - 1, 1, &ok2);

    if (ok1 && ok2) {

        graphWidget->dijkstra(from, to);

    }

}

void MainWindow::onFloydClicked()

{

    graphWidget->floydWarshall();

}

void MainWindow::onTSPClicked()

{

    graphWidget->solveTSP();

}

main.cpp:

#include "mainwindow.h"

#include <QApplication>

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

{

    QApplication a(argc, argv);

    MainWindow w;

    w.show();

    return a.exec();

}

Выводы:

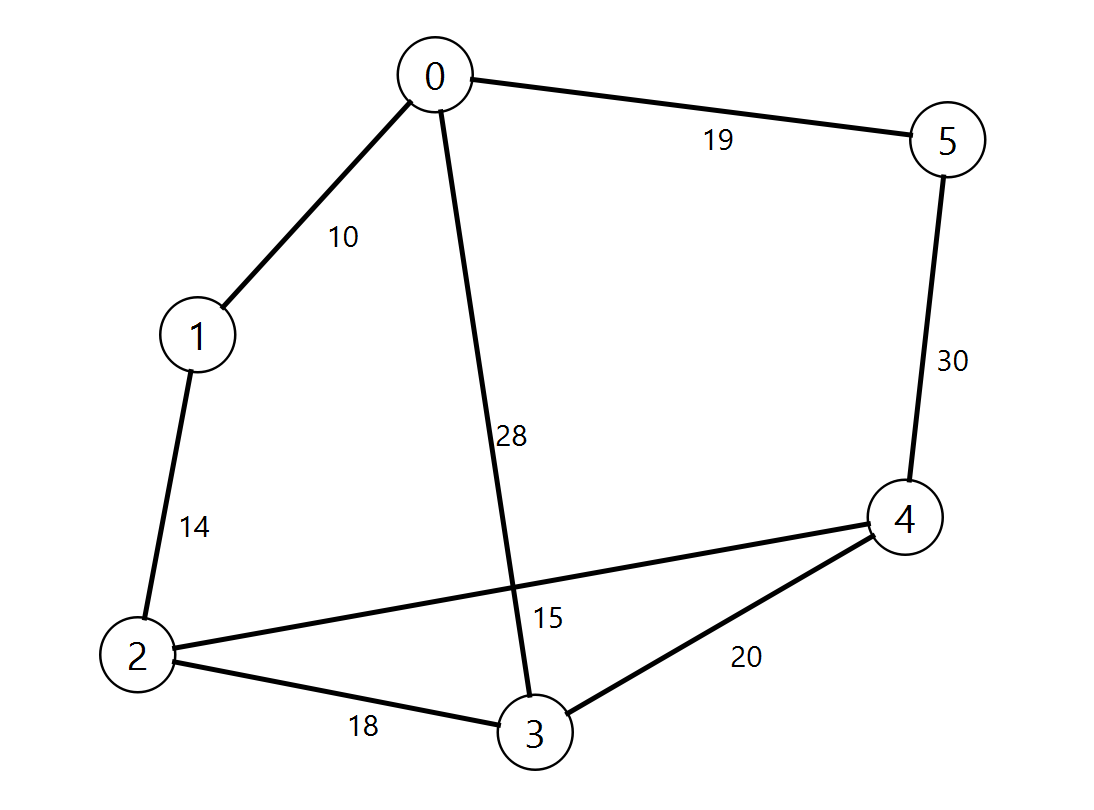
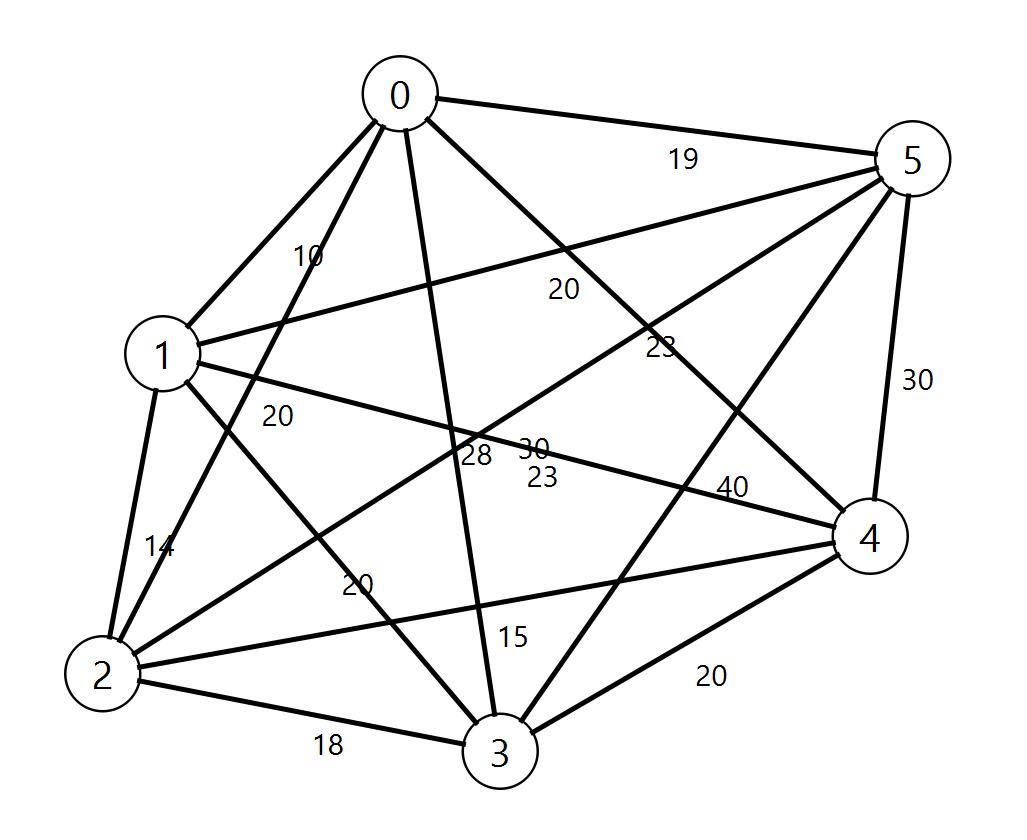
Рисунок 1 Граф варианта

Рисунок 2. Переделанный граф

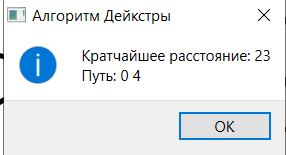


Рисунок 3. Вывод алгоритма Дейкстры

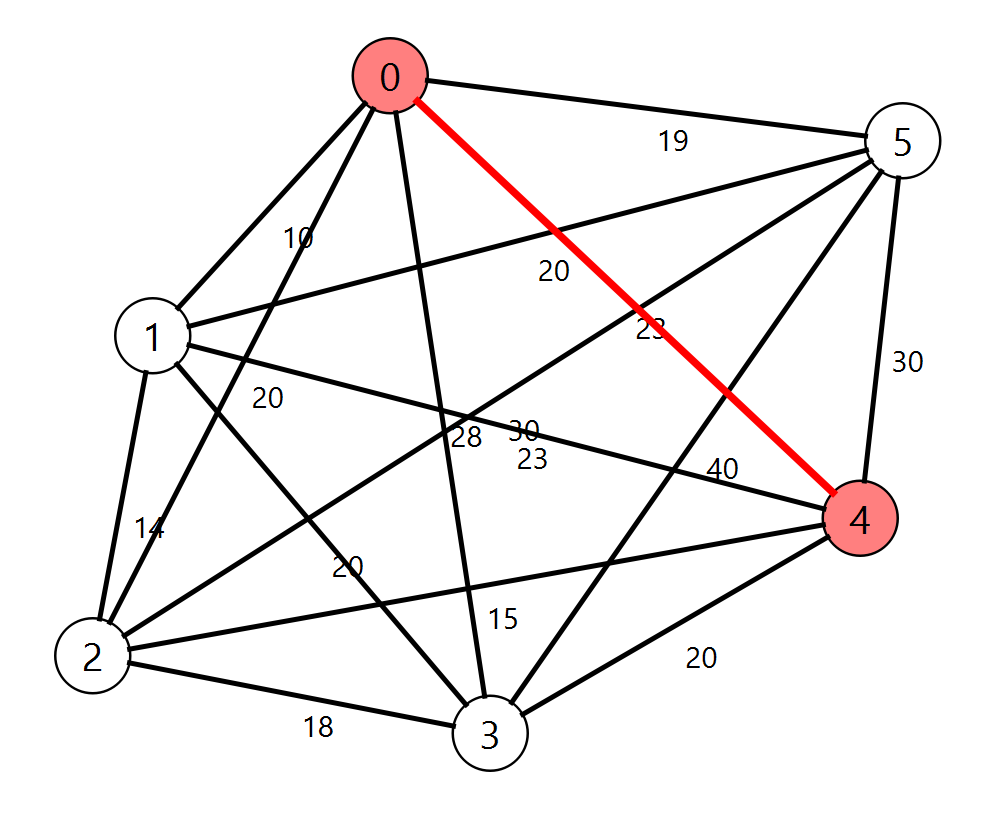


Рисунок 4. Вывод алгоритма Дейкстры

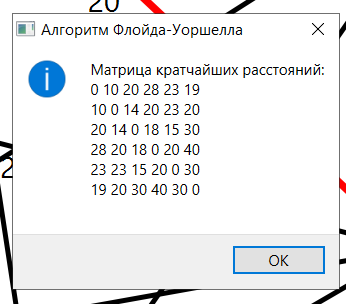


Рисунок 5. Вывод алгоритма Флойда

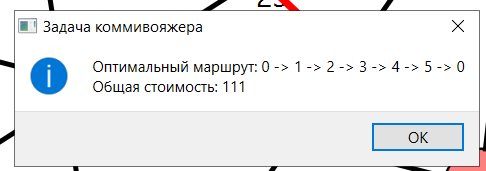


Рисунок 6. Вывод решения задачи коммивояжера

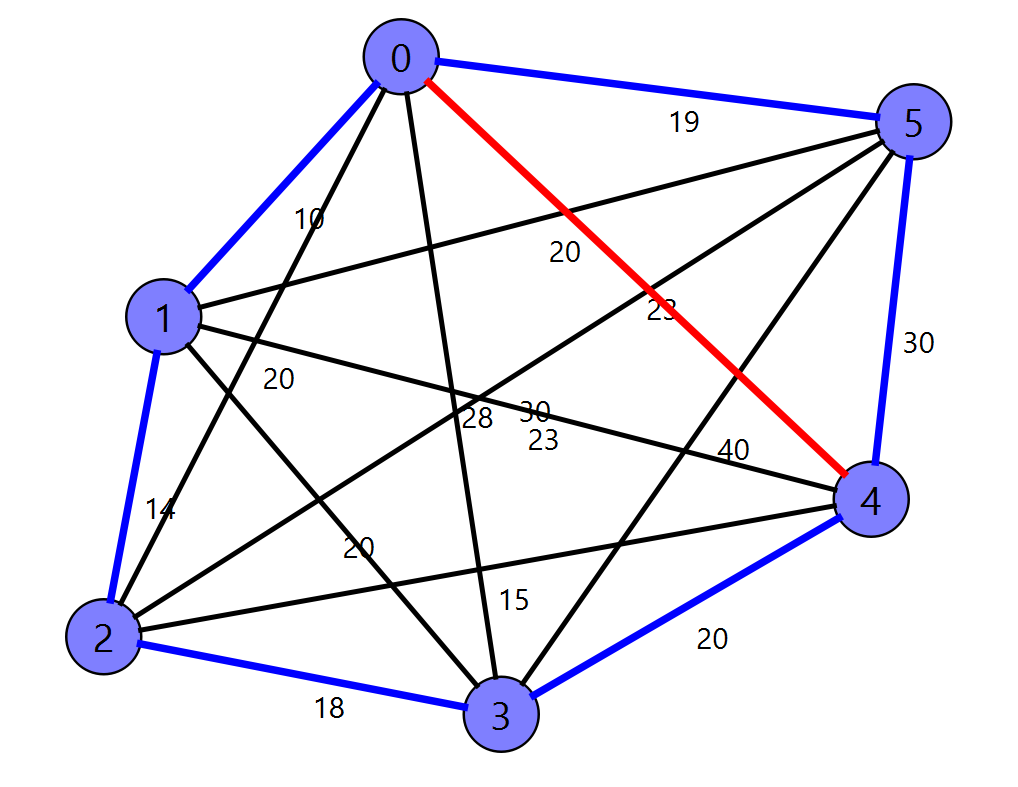


Рисунок 7. Вывод коммивояжера