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**

**«Задача Коммивояжёра»**

Выполнил студент гр. РИС-24-2б

Молочко Артём Анатольевич

Проверил:

Доц. Каф. ИТАС

Ольга Андреевна Полякова

(оценка) (подпись)

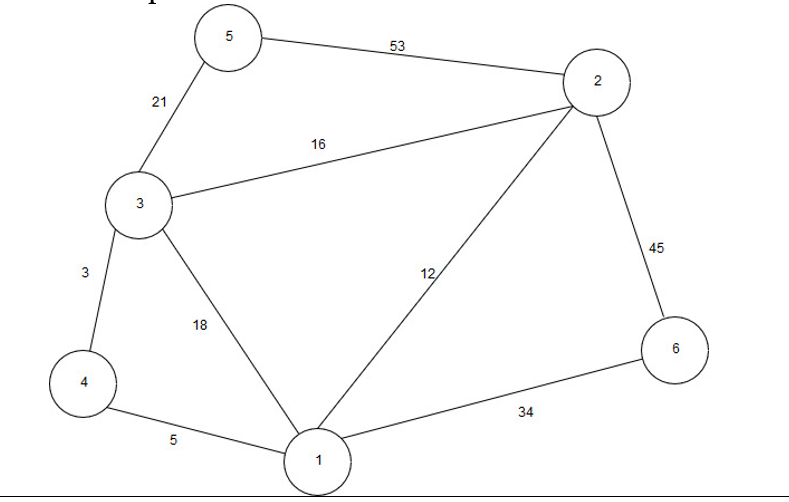
(дата)

г. Пермь, 2025

**Постановка задачи**

Разработать программу, которая:

1. **Генерирует** или позволяет вручную задавать города (вершины графа) на плоскости.
2. **Строит** полный взвешенный граф, где веса рёбер — расстояния между городами.
3. **Решает** задачу коммивояжёра методом ветвей и границ
4. **Визуализирует** процесс решения и итоговый маршрут на графической сцене.

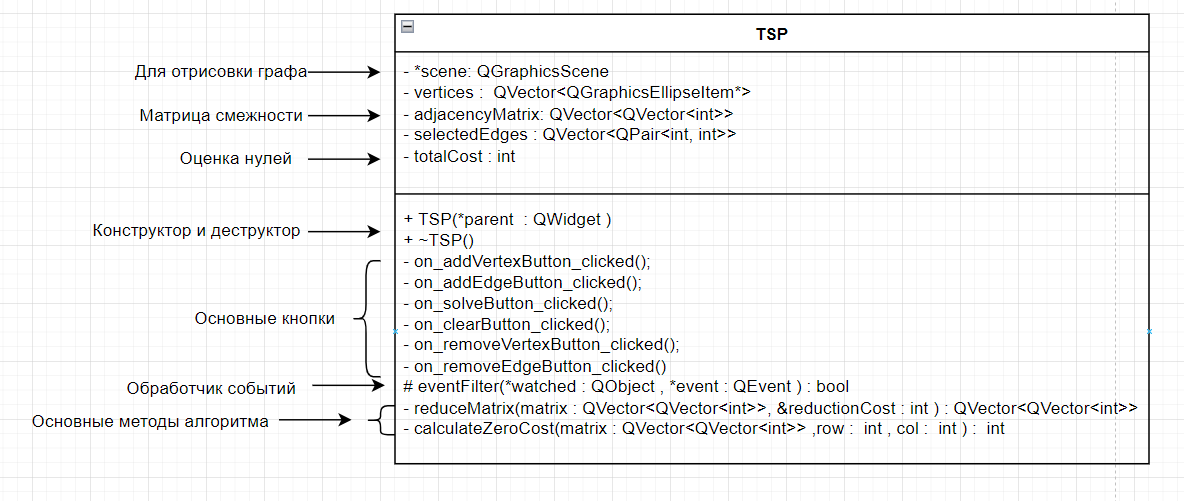
****

**Анализ задачи**

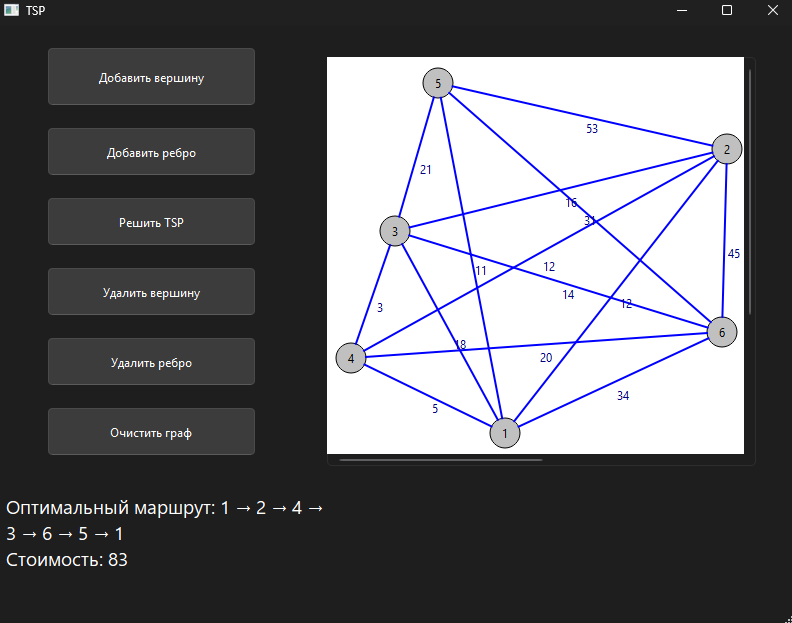
Задача коммивояжёра заключается в нахождении **кратчайшего замкнутого маршрута**, проходящего через все города ровно по одному разу и возвращающегося в исходную точку.

Алгоритм ветвей и границ для задачи коммивояжёра работает следующим образом:

1. **Построение матрицы смежности**
   * Если города заданы координатами, вычисляются расстояния между ними.
   * Если матрица дана, она используется напрямую.
2. **Редукция матрицы (упрощение)**
   * В каждой строке находится минимальный элемент и вычитается из всех элементов строки.
   * То же самое делается для столбцов.
   * Сумма всех вычтенных минимумов даёт начальную нижнюю границу стоимости маршрута.
3. **Выбор нулевого элемента с максимальной оценкой**
   * Для всех нулей в матрице вычисляется оценка (сумма минимальных элементов в строке и столбце без этого нуля).
   * Выбирается ноль с наибольшей оценкой — это ребро, которое войдёт в маршрут.
4. **Разделение задачи на две подзадачи**
   * **Включение ребра:**
     + Ребро добавляется в маршрут.
     + Удаляются строка и столбец, соответствующие этому ребру.
   * **Исключение ребра:**
     + Этот элемент в матрице заменяется на бесконечность (чтобы больше не рассматривать).
5. **Повторение редукции и ветвления**
   * Для каждой новой матрицы повторяются шаги 2–4.
   * Процесс продолжается, пока не будет найден полный маршрут.
6. **Построение итогового маршрута**
   * Когда матрица становится 2×2, оставшиеся рёбра автоматически включаются в путь.
   * Вычисляется общая длина оптимального цикла.

**Диаграмма классов**

**Модернизированный граф и результат работы**

****

**Код программы**

Программа разработа на Qt 6 / C++

**tsp.h**

#ifndef TSP\_H

#define TSP\_H

#include <QMainWindow>

#include <QGraphicsScene>

#include <QGraphicsEllipseItem>

#include <QGraphicsLineItem>

#include <QVector>

#include <QPair>

#include <QMouseEvent>

#include <climits>

QT\_BEGIN\_NAMESPACE

namespace Ui { class TSP; }

QT\_END\_NAMESPACE

class TSP : public QMainWindow

{

Q\_OBJECT

public:

explicit TSP(QWidget \*parent = nullptr);

~TSP();

protected:

bool eventFilter(QObject \*watched, QEvent \*event) override;

private slots:

// Основные кнопки

void on\_addVertexButton\_clicked();

void on\_addEdgeButton\_clicked();

void on\_solveButton\_clicked();

void on\_clearButton\_clicked();

void on\_removeVertexButton\_clicked();

void on\_removeEdgeButton\_clicked();

private:

Ui::TSP \*ui;

QGraphicsScene \*scene;

QVector<QGraphicsEllipseItem\*> vertices;

QVector<QVector<int>> adjacencyMatrix;

QVector<QPair<int, int>> selectedEdges;

QVector<QGraphicsLineItem\*> selectedEdgeItems;

QVector<QGraphicsTextItem\*> selectedEdgeLabels;

int totalCost;

// Основные методы алгоритма

QVector<QVector<int>> reduceMatrix(QVector<QVector<int>> matrix, int &reductionCost);

int calculateZeroCost(QVector<QVector<int>> matrix, int row, int col);

// Вспомогательные методы

void addVertexAtPosition(const QPointF &pos);

bool isOverlapping(int x, int y, int minDistance);

};

#endif // TSP\_H

**tsp.cpp**

#include "tsp.h"

#include "ui\_tsp.h"

#include <QInputDialog>

#include <QMessageBox>

#include <algorithm>

#include <cmath>

TSP::TSP(QWidget \*parent)

: QMainWindow(parent)

, ui(new Ui::TSP)

, scene(new QGraphicsScene(this))

, totalCost(0)

{

ui->setupUi(this);

ui->graphicsView->setScene(scene);

scene->setBackgroundBrush(Qt::white);

ui->graphicsView->setRenderHint(QPainter::Antialiasing);

ui->graphicsView->viewport()->installEventFilter(this);

ui->graphicsView->setSceneRect(0, 0, 800, 600);

ui->resultLabel->setWordWrap(true);

}

TSP::~TSP()

{

delete ui;

}

bool TSP::eventFilter(QObject \*watched, QEvent \*event)

{

if (watched == ui->graphicsView->viewport() && event->type() == QEvent::MouseButtonPress) {

QMouseEvent \*mouseEvent = static\_cast<QMouseEvent\*>(event);

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

QPointF scenePos = ui->graphicsView->mapToScene(mouseEvent->pos());

addVertexAtPosition(scenePos);

return true;

}

}

return QMainWindow::eventFilter(watched, event);

}

void TSP::on\_addVertexButton\_clicked()

{

ui->resultLabel->setText("Кликните на сцену, чтобы разместить вершину");

ui->graphicsView->setMouseTracking(true);

}

void TSP::addVertexAtPosition(const QPointF &pos)

{

if (isOverlapping(pos.x(), pos.y(), 40)) {

QMessageBox::warning(this, "Ошибка", "Вершины слишком близко друг к другу!");

return;

}

QGraphicsEllipseItem \*vertex = scene->addEllipse(0, 0, 30, 30, QPen(Qt::black), QBrush(Qt::lightGray));

vertex->setPos(pos.x() - 15, pos.y() - 15);

QGraphicsTextItem \*label = scene->addText(QString::number(vertices.size() + 1));

label->setPos(vertex->pos() + QPointF(15 - label->boundingRect().width()/2,

15 - label->boundingRect().height()/2));

label->setDefaultTextColor(Qt::black);

vertices.append(vertex);

adjacencyMatrix.resize(vertices.size());

for (auto &row : adjacencyMatrix) {

row.resize(vertices.size(), INT\_MAX);

}

}

void TSP::on\_addEdgeButton\_clicked()

{

if (vertices.size() < 2) {

QMessageBox::warning(this, "Ошибка", "Нужно минимум 2 вершины!");

return;

}

QStringList vertexList;

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

vertexList << QString::number(i + 1);

}

bool ok;

QString from = QInputDialog::getItem(this, "Выберите вершину", "От:", vertexList, 0, false, &ok);

if (!ok) return;

QString to = QInputDialog::getItem(this, "Выберите вершину", "До:", vertexList, 1, false, &ok);

if (!ok || from == to) {

QMessageBox::warning(this, "Ошибка", "Выберите разные вершины!");

return;

}

int cost = QInputDialog::getInt(this, "Стоимость", "Введите стоимость:", 1, 1, 1000, 1, &ok);

if (!ok) return;

int fromIdx = from.toInt() - 1;

int toIdx = to.toInt() - 1;

adjacencyMatrix[fromIdx][toIdx] = cost;

adjacencyMatrix[toIdx][fromIdx] = cost;

// Получаем позиции центров вершин

QPointF fromCenter = vertices[fromIdx]->pos() + QPointF(15, 15);

QPointF toCenter = vertices[toIdx]->pos() + QPointF(15, 15);

// Вычисляем вектор направления

QPointF direction = toCenter - fromCenter;

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

if (length > 0) {

// Нормализуем вектор и укорачиваем на радиус вершины (15px)

QPointF normalized = direction / length;

QPointF startPoint = fromCenter + normalized \* 15;

QPointF endPoint = toCenter - normalized \* 15;

// Создаем линию

QLineF line(startPoint, endPoint);

QGraphicsLineItem \*edge = scene->addLine(line, QPen(Qt::blue, 2));

edge->setZValue(1); // Помещаем ребро под текст

// Добавляем текст со стоимостью

QGraphicsTextItem \*costLabel = scene->addText(QString::number(cost));

costLabel->setPos(line.center());

costLabel->setDefaultTextColor(Qt::darkBlue);

costLabel->setZValue(2); // Текст поверх ребра

// Сохраняем информацию о рёбрах и тексте стоимости

selectedEdges.append(qMakePair(fromIdx, toIdx));

selectedEdgeItems.append(edge); // Сохраняем ребро

selectedEdgeLabels.append(costLabel); // Сохраняем текст стоимости

}

}

void TSP::on\_solveButton\_clicked()

{

if (vertices.size() < 2) {

QMessageBox::warning(this, "Ошибка", "Добавьте минимум 2 вершины!");

return;

}

// Удаляем предыдущее выделение пути

for (auto item : scene->items()) {

if (item->data(0).toString() == "path") {

scene->removeItem(item);

delete item;

}

}

// Базовый случай для 2 вершин

if (vertices.size() == 2) {

int cost = adjacencyMatrix[0][1];

if (cost == INT\_MAX) {

ui->resultLabel->setText("Нет пути между вершинами!");

return;

}

ui->resultLabel->setText(QString("Оптимальный маршрут: 1 → 2 → 1\nСтоимость: %1").arg(2 \* cost));

return;

}

QVector<int> vertexIndices;

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

vertexIndices.append(i);

}

int minCost = INT\_MAX;

QVector<int> bestPath;

do {

int currentCost = 0;

bool validPath = true;

// Считаем стоимость

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

int from = vertexIndices[i];

int to = vertexIndices[(i + 1) % vertexIndices.size()];

if (adjacencyMatrix[from][to] == INT\_MAX) {

validPath = false;

break;

}

currentCost += adjacencyMatrix[from][to];

}

if (validPath && currentCost < minCost) {

minCost = currentCost;

bestPath = vertexIndices;

}

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

if (bestPath.isEmpty()) {

ui->resultLabel->setText("Нет возможного маршрута!");

return;

}

// Формируем строку результата

QString result = "Оптимальный маршрут: ";

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

if (i != 0) result += " → ";

result += QString::number(bestPath[i] + 1);

}

// Добавляем возврат в начальную вершину

result += QString(" → %1").arg(bestPath[0] + 1);

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

ui->resultLabel->setText(result);

}

QVector<QVector<int>> TSP::reduceMatrix(QVector<QVector<int>> matrix, int &reductionCost)

{

// Редукция строк

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

int minVal = INT\_MAX;

// Находим минимальный элемент в строке

for (int j = 0; j < matrix[i].size(); ++j) {

if (matrix[i][j] < minVal) {

minVal = matrix[i][j];

}

}

if (minVal != INT\_MAX && minVal > 0) {

// Вычитаем минимальный элемент из всех элементов строки

for (int j = 0; j < matrix[i].size(); ++j) {

if (matrix[i][j] != INT\_MAX) {

matrix[i][j] -= minVal;

}

}

reductionCost += minVal;

}

}

// Редукция столбцов

for (int j = 0; j < matrix[0].size(); ++j) {

int minVal = INT\_MAX;

// Находим минимальный элемент в столбце

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

if (matrix[i][j] < minVal) {

minVal = matrix[i][j];

}

}

if (minVal != INT\_MAX && minVal > 0) {

// Вычитаем минимальный элемент из всех элементов столбца

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

if (matrix[i][j] != INT\_MAX) {

matrix[i][j] -= minVal;

}

}

reductionCost += minVal;

}

}

return matrix;

}

int TSP::calculateZeroCost(QVector<QVector<int>> matrix, int row, int col)

{

int minRow = INT\_MAX;

int minCol = INT\_MAX;

for (int j = 0; j < matrix[row].size(); ++j) {

if (j != col && matrix[row][j] < minRow) {

minRow = matrix[row][j];

}

}

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

if (i != row && matrix[i][col] < minCol) {

minCol = matrix[i][col];

}

}

return (minRow == INT\_MAX ? 0 : minRow) + (minCol == INT\_MAX ? 0 : minCol);

}

void TSP::on\_removeVertexButton\_clicked()

{

if (vertices.isEmpty()) {

QMessageBox::warning(this, "Ошибка", "Нет вершин для удаления!");

return;

}

// Создаем список вершин для выбора

QStringList vertexList;

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

vertexList << QString::number(i + 1);

}

bool ok;

QString vertexToRemove = QInputDialog::getItem(this, "Удаление вершины",

"Выберите вершину для удаления:",

vertexList, 0, false, &ok);

if (!ok) return;

int vertexIndex = vertexToRemove.toInt() - 1;

// Удаляем все связанные с вершиной элементы из сцены

QGraphicsEllipseItem\* vertexItem = vertices[vertexIndex];

// Удаляем все ребра, связанные с этой вершиной

QList<QGraphicsItem\*> itemsToRemove;

for (auto item : scene->items()) {

if (auto line = dynamic\_cast<QGraphicsLineItem\*>(item)) {

QLineF lineF = line->line();

QPointF vertexCenter = vertexItem->pos() + QPointF(15, 15);

// Проверяем, связано ли ребро с удаляемой вершиной

if (lineF.p1() == vertexCenter || lineF.p2() == vertexCenter) {

itemsToRemove.append(item);

// Удаляем все текстовые метки стоимости этого ребра

for (auto label : scene->items(lineF.center())) {

if (dynamic\_cast<QGraphicsTextItem\*>(label)) {

itemsToRemove.append(label);

}

}

}

}

// Удаляем текстовую метку вершины

else if (auto text = dynamic\_cast<QGraphicsTextItem\*>(item)) {

if (text->pos() == vertexItem->pos() + QPointF(15 - text->boundingRect().width()/2,

15 - text->boundingRect().height()/2)) {

itemsToRemove.append(text);

}

}

}

// Удаляем все найденные элементы

for (auto item : itemsToRemove) {

scene->removeItem(item);

delete item;

}

// Удаляем саму вершину

scene->removeItem(vertexItem);

delete vertexItem;

// Удаляем вершину из вектора

vertices.remove(vertexIndex);

// Обновляем матрицу смежности

adjacencyMatrix.remove(vertexIndex);

for (auto &row : adjacencyMatrix) {

row.remove(vertexIndex);

}

// Обновляем нумерацию оставшихся вершин

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

for (auto item : scene->items(vertices[i]->pos())) {

if (auto text = dynamic\_cast<QGraphicsTextItem\*>(item)) {

text->setPlainText(QString::number(i + 1));

}

}

}

ui->resultLabel->setText(QString("Вершина %1 удалена").arg(vertexIndex + 1));

}

void TSP::on\_removeEdgeButton\_clicked()

{

if (selectedEdgeItems.isEmpty()) {

QMessageBox::warning(this, "Ошибка", "Нет рёбер для удаления!");

return;

}

// Создаем список рёбер для выбора

QStringList edgeList;

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

int from = selectedEdges[i].first + 1;

int to = selectedEdges[i].second + 1;

edgeList << QString("%1 ↔ %2").arg(from).arg(to);

}

bool ok;

QString edgeToRemove = QInputDialog::getItem(this, "Удаление рёбер", "Выберите ребро для удаления:", edgeList, 0, false, &ok);

if (!ok) return;

// Получаем индексы вершин, между которыми находится выбранное ребро

int edgeIndex = edgeList.indexOf(edgeToRemove);

if (edgeIndex == -1) return; // Если ничего не выбрано, выходим

int fromIdx = selectedEdges[edgeIndex].first;

int toIdx = selectedEdges[edgeIndex].second;

// Удаляем ребро из сцены

QGraphicsLineItem \*edge = selectedEdgeItems[edgeIndex];

scene->removeItem(edge);

delete edge;

// Удаляем текстовую метку стоимости

QGraphicsTextItem \*costLabel = selectedEdgeLabels[edgeIndex];

scene->removeItem(costLabel);

delete costLabel;

// Удаляем рёбра из матрицы смежности

adjacencyMatrix[fromIdx][toIdx] = INT\_MAX;

adjacencyMatrix[toIdx][fromIdx] = INT\_MAX;

// Удаляем ребро и текст из соответствующих списков

selectedEdges.removeAt(edgeIndex);

selectedEdgeItems.removeAt(edgeIndex);

selectedEdgeLabels.removeAt(edgeIndex);

ui->resultLabel->setText(QString("Ребро между %1 и %2 удалено").arg(fromIdx + 1).arg(toIdx + 1));

}

void TSP::on\_clearButton\_clicked()

{

scene->clear();

vertices.clear();

adjacencyMatrix.clear();

selectedEdges.clear();

totalCost = 0;

ui->resultLabel->setText("Граф очищен");

}

bool TSP::isOverlapping(int x, int y, int minDistance)

{

for (auto vertex : vertices) {

QPointF pos = vertex->pos();

if (sqrt(pow(x - pos.x(), 2) + pow(y - pos.y(), 2) < minDistance)) {

return true;

}

}

return false;

}