Реализация на C++

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include <QVBoxLayout>

#include <QLabel>

#include <QGraphicsEllipseItem>

#include <QGraphicsLineItem>

#include <cmath>

#include <limits>

#include <iostream>

#include <QHeaderView>

using namespace std;

// Конструктор главного окна приложения

MainWindow::MainWindow(QWidget \*parent)

: QMainWindow(parent)

, ui(new Ui::MainWindow)

{

ui->setupUi(this); // Настройка UI из файла .ui

QWidget \*central = new QWidget(this); // Создаем центральный виджет

QVBoxLayout \*layout = new QVBoxLayout(central); // Вертикальный слой для размещения элементов

// Создаем кнопку "Решить"

solveButton = new QPushButton("Решить", this);

output = new QTextEdit(this); // Текстовое поле для вывода результатов

output->setReadOnly(true); // Делает поле только для чтения

// Таблица для отображения матрицы расстояний

matrixTable = new QTableWidget(this);

matrixTable->setEditTriggers(QAbstractItemView::NoEditTriggers); // Запрет редактирования

// Таблица для отображения маршрута

pathTable = new QTableWidget(this);

pathTable->setEditTriggers(QAbstractItemView::NoEditTriggers); // Запрет редактирования

// Добавляем в layout метки и таблицы

layout->addWidget(new QLabel("Маршрут:"));

layout->addWidget(pathTable);

layout->addWidget(new QLabel("Матрица расстояний:"));

layout->addWidget(matrixTable);

// Виджет для визуализации графа

view = new QGraphicsView(this);

scene = new QGraphicsScene(this);

view->setScene(scene);

layout->addWidget(view);

layout->addWidget(solveButton);

layout->addWidget(new QLabel("Результат:"));

layout->addWidget(output);

setCentralWidget(central); // Устанавливаем центральный виджет

setWindowTitle("Коммивояжер — ветви и границы"); // Заголовок окна

resize(700, 800); // Размер окна

// Инициализация матрицы расстояний между вершинами

matrix = {

{0, 8, 0, 0, 0, 11},

{8, 0, 12, 0, 10, 0},

{0, 12, 0, 16, 0, 0},

{0, 0, 16, 0, 5, 9},

{0, 10, 0, 5, 0, 6},

{11, 0, 0, 9, 6, 0}

};

// Подключаем сигнал нажатия кнопки "Решить" к слоту solveTSP()

connect(solveButton, &QPushButton::clicked, this, &MainWindow::solveTSP);

}

MainWindow::~MainWindow()

{

delete ui; // Удаляем UI при уничтожении окна

}

// Слот, вызываемый при нажатии кнопки "Решить"

void MainWindow::solveTSP() {

vector<int> path; // Вектор для хранения найденного маршрута

int cost = tspBranchBound(matrix, path); // Запуск алгоритма ветвей и границ для задачи коммивояжера

output->clear(); // Очистка текстового поля результата

output->append("Минимальная стоимость: " + QString::number(cost)); // Вывод минимальной стоимости маршрута

output->append("Маршрут:");

for (int node : path) // Вывод вершин маршрута по порядку (с учётом сдвига на 1 для удобства пользователя)

output->append("Вершина " + QString::number(node + 1));

int n = matrix.size();

// Настройка таблицы с матрицей расстояний

matrixTable->setRowCount(n);

matrixTable->setColumnCount(n);

matrixTable->setHorizontalHeaderLabels(QStringList());

matrixTable->setVerticalHeaderLabels(QStringList());

matrixTable->horizontalHeader()->setStretchLastSection(true);

matrixTable->verticalHeader()->setStretchLastSection(true);

matrixTable->resizeColumnsToContents();

matrixTable->resizeRowsToContents();

// Настройка таблицы маршрута

pathTable->clear();

pathTable->setRowCount(path.size() - 1);

pathTable->setColumnCount(3);

pathTable->setHorizontalHeaderLabels(QStringList() << "От" << "До" << "Расстояние");

int total = 0;

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

int from = path[i];

int to = path[i + 1];

int dist = matrix[from][to];

// Заполняем таблицу маршрута: из какой вершины, в какую, и расстояние

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

pathTable->setItem(i, 1, new QTableWidgetItem(QString::number(to + 1)));

pathTable->setItem(i, 2, new QTableWidgetItem(QString::number(dist)));

total += dist;

}

pathTable->resizeColumnsToContents();

pathTable->resizeRowsToContents();

// Заполняем таблицу матрицы расстояний, заменяя нули без ребер на "∞"

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

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

QString value = (matrix[i][j] > 0 || i == j)

? QString::number(matrix[i][j])

: "∞";

matrixTable->setItem(i, j, new QTableWidgetItem(value));

}

}

// Отрисовка графа с маршрутом

drawGraph(path);

}

// Алгоритм решения задачи коммивояжера методом ветвей и границ

int MainWindow::tspBranchBound(const vector<vector<int>>& graph, vector<int>& bestPath) {

int n = graph.size();

vector<bool> visited(n, false); // Вектор посещенных вершин

vector<int> path = {5}; // Начинаем с вершины 5 (индексация с 0)

visited[5] = true;

int bestCost = numeric\_limits<int>::max(); // Инициализация минимальной стоимости очень большим числом

branch(1, 0, path, visited, bestCost, bestPath); // Запускаем рекурсивный обход с ветками

return bestCost;

}

// Рекурсивная функция обхода с ветвями и отсечками

void MainWindow::branch(int level, int currentCost, vector<int>& path, vector<bool>& visited,

int& bestCost, vector<int>& bestPath) {

int n = matrix.size();

if (level == n) { // Если достигли всех вершин

int totalCost = currentCost + matrix[path.back()][path[0]]; // Добавляем стоимость возврата в начальную вершину

if (totalCost < bestCost) { // Если нашли лучшее решение

bestCost = totalCost;

bestPath = path; // Запоминаем маршрут

bestPath.push\_back(0); // Возврат в стартовую вершину

}

return;

}

// Перебор всех непосещенных вершин, доступных из текущей

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

if (!visited[i] && matrix[path.back()][i] > 0) { // Если вершина не посещена и ребро существует

int nextCost = currentCost + matrix[path.back()][i]; // Предварительная стоимость маршрута при выборе этой вершины

if (nextCost < bestCost) { // Отсечение: идем дальше, только если стоимость меньше текущего лучшего

visited[i] = true; // Отмечаем вершину как посещенную

path.push\_back(i); // Добавляем вершину в маршрут

branch(level + 1, nextCost, path, visited, bestCost, bestPath); // Рекурсивный вызов

visited[i] = false; // Откат - снимаем отметку о посещении для других вариантов

path.pop\_back(); // Удаляем вершину из текущего пути для перебора других вариантов

}

}

}

}

// Функция отрисовки графа с маршрутом в QGraphicsScene

void MainWindow::drawGraph(const vector<int>& path) {

scene->clear(); // Очищаем предыдущую отрисовку

int n = matrix.size();

const int radius = 20; // Радиус рисуемых кружков-вершин

const int centerX = 300, centerY = 300, graphRadius = 200; // Центр и радиус круга для размещения вершин

QVector<QPointF> positions; // Позиции для каждой вершины

// Располагаем вершины по окружности

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

double angle = 2 \* M\_PI \* i / n; // Угол для размещения вершины

double x = centerX + graphRadius \* cos(angle);

double y = centerY + graphRadius \* sin(angle);

positions.append(QPointF(x, y));

// Добавляем вершину в сцену (желтый круг с черной границей)

scene->addEllipse(x - radius/2, y - radius/2, radius, radius,

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

// Добавляем подпись с номером вершины

QGraphicsTextItem \*text = scene->addText(QString::number(i + 1));

text->setPos(x + 10, y + 10);

}

// Рисуем серые ребра между всеми связанными вершинами

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

for (int j = i + 1; j < n; ++j)

if (matrix[i][j] > 0)

scene->addLine(positions[i].x(), positions[i].y(),

positions[j].x(), positions[j].y(),

QPen(Qt::gray));

// Рисуем красным цветом ребра, входящие в найденный маршрут

for (size\_t i = 1; i < path.size(); ++i) {

int from = path[i - 1];

int to = path[i];

scene->addLine(QLineF(positions[from], positions[to]), QPen(Qt::red, 3));

}

}

Реализация на Java

import java.io.\*;

import java.util.\*;

import java.util.concurrent.TimeUnit;

import java.util.stream.Collectors;

import java.util.stream.IntStream;

public class Lb2 {

// Функции для генерации, сохранения и загрузки матрицы весов

public static int[][] generateWeightMatrix(int n, boolean symmetric) {

// Создаем квадратную матрицу размером n x n

int[][] matrix = new int[n][n];

Random random = new Random();

// Двойной цикл для заполнения матрицы

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

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

// Главная диагональ всегда заполняется нулями

if (x == y) {

matrix[y][x] = 0;

// Если нужна симметричная матрица (для неориентированного графа)

} else if (symmetric) {

// Заполняем только верхний треугольник матрицы (x > y)

// и зеркально копируем значения в нижний треугольник

if (x > y) {

int weight = random.nextInt(10) + 1; // Случайный вес от 1 до 10

matrix[y][x] = weight;

matrix[x][y] = weight;

}

// Если нужна несимметричная матрица (для ориентированного графа)

} else {

// Заполняем все элементы, кроме диагонали

if (x != y) {

matrix[y][x] = random.nextInt(10) + 1;

}

}

}

}

return matrix;

}

//Сохраняет матрицу весов в текстовый файл

//throws IOException - если возникли проблемы с записью в файл

public static void saveWeightMatrix(String fileName, int[][] matrix) throws IOException {

// Используем try-with-resources для автоматического закрытия файла

try (PrintWriter writer = new PrintWriter(new FileWriter(fileName))) {

// Проходим по каждой строке матрицы

for (int[] row : matrix) {

// Преобразуем строку матрицы в текстовый формат:

// 1. Преобразуем int[] в IntStream

// 2. Конвертируем числа в строки

// 3. Объединяем через пробел

writer.println(Arrays.stream(row).mapToObj(String::valueOf).collect(Collectors.joining(" ")));

}

}

}

// Загружает матрицу весов из текстового файла

public static int[][] loadWeightMatrix(String fileName) throws IOException {

// Список для хранения строк матрицы

List<int[]> matrix = new ArrayList<>();

try (BufferedReader reader = new BufferedReader(new FileReader(fileName))) {

String line;

// Читаем файл построчно

while ((line = reader.readLine()) != null) {

// Преобразуем строку в массив чисел:

// 1. Разбиваем строку по пробелам

// 2. Преобразуем каждую подстроку в int

// 3. Собираем в массив int[]

int[] row = Arrays.stream(line.split(" ")).mapToInt(Integer::parseInt).toArray();

matrix.add(row);

}

}

// Преобразуем список в двумерный массив

return matrix.toArray(new int[0][]);

}

// Функции алгоритма нахождения пути

/\*

\* Вычисляет нижнюю границу стоимости для текущего частичного пути

\* Используется в алгоритме ветвей и границ для оценки минимально возможной стоимости завершения маршрута

\*/

public static double lowerBound(int[][] matrix, List<Integer> path, Set<Integer> visited) {

int n = matrix.length;

double bound = 0; // Итоговая нижняя граница

// 1. Разбиение на "куски" (подпути)

List<List<Integer>> parts = new ArrayList<>();

parts.add(path);// Добавляем текущий путь как первый кусок

// Добавляем непосещенные вершины как отдельные куски

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

if (!visited.contains(i)) {

parts.add(Collections.singletonList(i));

}

}

System.out.println("\nВычисляем нижнюю границу");

System.out.println("Куски: " + parts);

// 2. Расчет вклада каждого куска в нижнюю границу

for (List<Integer> part : parts) {

int start = part.get(0); // Начальная вершина куска

int end = part.get(part.size() - 1); // Конечная вершина куска

// Списки для хранения весов допустимых дуг

List<Integer> incoming = new ArrayList<>(); // Входящие в начало куска

List<Integer> outgoing = new ArrayList<>(); // Исходящие из конца куска

// 3. Поиск допустимых дуг между кусками

for (List<Integer> otherPart : parts) {

if (otherPart != part) { // Не рассматриваем текущий кусок

int otherPartStart = otherPart.get(0);

int otherPartEnd = otherPart.get(otherPart.size() - 1);

// Добавляем вес дуги из конца другого куска в начало текущего

if (matrix[otherPartEnd][start] != 0) {

incoming.add(matrix[otherPartEnd][start]);

}

// Добавляем вес дуги из конца текущего куска в начало другого

if (matrix[end][otherPartStart] != 0) {

outgoing.add(matrix[end][otherPartStart]);

}

}

}

// 4. Находим минимальные дуги

int incomingMin = incoming.isEmpty() ? 0 : Collections.min(incoming);

int outgoingMin = outgoing.isEmpty() ? 0 : Collections.min(outgoing);

// 5. Добавляем среднее значение к общей границе

bound += (incomingMin + outgoingMin) / 2.0;

System.out.printf(

"Для куска %s: расстояние минимальной входящей дуги = %d, расстояние минимальной исходящей дуги = %d, добавлено к границе: %.1f%n",

part, incomingMin, outgoingMin, (incomingMin + outgoingMin) / 2.0

);

}

System.out.printf("Итоговая нижняя граница для пути %s: %.1f%n%n", path, bound);

return bound;

}

//Реализация алгоритма ветвей и границ для задачи коммивояжера

public static Object[] branchAndBound(int[][] matrix, int start) {

int n = matrix.length;

List<Integer> bestPath = null;// Лучший найденный путь

double bestCost = Double.POSITIVE\_INFINITY; // Стоимость лучшего пути

Set<Integer> visited = new HashSet<>(Collections.singleton(start)); // Посещенные вершины

// Хэш-таблица для мемоизации аналогичных путей

Map<String, Double> memory = new HashMap<>();

// Стек для хранения ветвей

Deque<Object[]> stack = new ArrayDeque<>();

// Начальный путь [start], Начальная стоимость, Множество посещенных

stack.push(new Object[]{new ArrayList<>(Collections.singletonList(start)), 0.0, new HashSet<>(visited)});

System.out.printf("%n(1) ----- Начало работы метода ветвей и границ. Стартовая вершина: %d -----%n", start);

//Основной цикл (пока есть ветви для исследования)

while (!stack.isEmpty()) {

// Извлекаем текущую ветвь из стека

Object[] node = stack.pop();

//Приведение типов (компилятор требует аннотацию @SuppressWarnings)

@SuppressWarnings("unchecked") //Необходимо, так как мы храним разные типы в Object[]

List<Integer> path = (List<Integer>) node[0]; // Текущий путь

double cost = (double) node[1]; // Текущая стоимость

@SuppressWarnings("unchecked")

Set<Integer> pathVisited = (Set<Integer>) node[2]; // Посещенные вершины

System.out.printf("%n# Берем ветвь: Путь %s, Стоимость %.1f%n", path, cost);

// Проверка на завершение пути (все вершины посещены)

if (path.size() == n) {

// Вычисляем полную стоимость (с возвратом в начало)

double totalCost = cost + matrix[path.get(path.size() - 1)][start];

// Проверяем, лучше ли текущий путь

if (totalCost < bestCost && matrix[path.get(path.size() - 1)][start] != 0) {

bestCost = totalCost;

bestPath = new ArrayList<>(path);

bestPath.add(start); // Замыкаем цикл

System.out.printf("Найден новый лучший путь: %s, Стоимость: %.1f%n", bestPath, bestCost);

} else {

System.out.printf("Отсечение ветви: %.1f >= %.1f%n", totalCost, bestCost);

}

} else {

// Оптимизация: проверка аналогичных путей (для путей ≥4 вершин)

if (path.size() >= 4) {

// Генерируем ключ: "начало,отсортированные\_промежуточные,конец"

int startVertex = path.get(0);

int endVertex = path.get(path.size() - 1);

List<Integer> middle = new ArrayList<>(path.subList(1, path.size() - 1));

Collections.sort(middle);

String key = startVertex + "," + middle + "," + endVertex;

// Если уже есть путь с таким же шаблоном и меньшей стоимостью

if (memory.containsKey(key)) {

if (cost >= memory.get(key)) {

System.out.printf("Отсечение ветви: %s, Стоимость: %.1f (уже есть лучший аналог)%n", path, cost);

continue; // Пропускаем эту ветвь

}

} else {

memory.put(key, cost); // Сохраняем новый шаблон

System.out.printf("Сохранение ветви в хэш-таблицу: %s, Стоимость: %.1f%n", path, cost);

}

}

//Вычисляем нижнюю границу стоимости завершения

double lb = lowerBound(matrix, path, pathVisited);

//Проверка перспективности ветви

if (cost + lb < bestCost) {

int last = path.get(path.size() - 1); // Последняя вершина в пути

List<int[]> candidates = new ArrayList<>();

//Формируем список возможных продолжений

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

// Если вершина не посещена и есть ребро last→i

if (!pathVisited.contains(i) && matrix[last][i] > 0) {

candidates.add(new int[]{matrix[last][i], i});// [вес, вершина]

}

}

//Сортируем кандидатов по убыванию веса

candidates.sort((a, b) -> Integer.compare(b[0], a[0]));

System.out.printf("Текущий путь: %s, Стоимость: %.1f, Нижняя оценка: %.1f%n", path, cost, lb);

//Добавляем новые ветви в стек

for (int[] candidate : candidates) {

int weight = candidate[0];

int i = candidate[1];

double newCost = cost + weight;

// Создаем новый путь

List<Integer> newPath = new ArrayList<>(path);

newPath.add(i);

// Обновляем множество посещенных

Set<Integer> newVisited = new HashSet<>(pathVisited);

newVisited.add(i);

// Добавляем новое состояние в стек

stack.push(new Object[]{newPath, newCost, newVisited}); //DFS

System.out.printf("Добавлен в стек путь: %s, Стоимость: %.1f%n", newPath, newCost);

}

} else {

//Отсечение неперспективных ветвей

System.out.printf("Отсечение ветви: %.1f >= %.1f%n", cost + lb, bestCost);

}

}

}

System.out.printf("Завершение работы метода ветвей и границ. Лучший путь: %s, Стоимость: %.1f%n", bestPath, bestCost);

return new Object[]{bestPath, bestCost};

}

//Алгоритм "Ближайшего включения"

public static Object[] nearestInsertion(int[][] matrix, int start) {

int n = matrix.length;

List<Integer> path = new ArrayList<>(Collections.singletonList(start));

double cost = 0;

Set<Integer> visited = new HashSet<>(Collections.singleton(start));

System.out.printf("%n(2) ----- Начало работы алгоритма ближайшего включения. Стартовая вершина: %d -----%n", start);

System.out.println("Начальный путь: " + path + ", Стоимость: " + cost);

while (path.size() < n) {

double minDistance = Double.POSITIVE\_INFINITY;

int nearestVertex = -1;

int insertPosition = -1;

System.out.println("\n--- Поиск ближайшей вершины для вставки ---");

System.out.println("Текущий путь: " + path);

System.out.println("Непосещённые вершины: " +

IntStream.range(0, n).filter(i -> !visited.contains(i)).boxed().collect(Collectors.toList()));

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

int currentVertex = path.get(i);

int nextVertex = (i + 1 < path.size()) ? path.get(i + 1) : path.get(0);

System.out.printf("\nПроверяем рёбра из вершины %d (позиция %d в пути):%n", currentVertex, i);

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

if (!visited.contains(j) && matrix[currentVertex][j] > 0) {

System.out.printf(" Вершина %d -> %d (вес %d)", currentVertex, j, matrix[currentVertex][j]);

if (path.size() > 1) {

System.out.printf(", проверка обратного ребра %d -> %d (вес %d)",

j, nextVertex, matrix[j][nextVertex]);

}

if (matrix[currentVertex][j] < minDistance && (path.size() == 1 || matrix[j][nextVertex] > 0)) {

System.out.print(" — НОВЫЙ КАНДИДАТ (минимальное расстояние)");

minDistance = matrix[currentVertex][j];

nearestVertex = j;

insertPosition = i;

}

System.out.println();

}

}

}

if (nearestVertex != -1) {

System.out.printf("\n>>> Наилучший выбор: вершина %d (расстояние %.1f), вставляем после позиции %d%n",

nearestVertex, minDistance, insertPosition);

if (insertPosition == path.size() - 1) {

cost += matrix[path.get(insertPosition)][nearestVertex];

path.add(nearestVertex);

System.out.printf(

"Добавляем в конец: %d -> %d (вес %d). Новый путь: %s, Общая стоимость: %.1f%n",

path.get(insertPosition), nearestVertex, matrix[path.get(insertPosition)][nearestVertex],

path, cost

);

} else {

int prevVertex = path.get(insertPosition);

int nextVertex = path.get(insertPosition + 1);

cost += matrix[prevVertex][nearestVertex] + matrix[nearestVertex][nextVertex] - matrix[prevVertex][nextVertex];

path.add(insertPosition + 1, nearestVertex);

System.out.printf(

"Вставляем между %d и %d: %d -> %d (вес %d) + %d -> %d (вес %d) - %d -> %d (вес %d).%n",

prevVertex, nextVertex,

prevVertex, nearestVertex, matrix[prevVertex][nearestVertex],

nearestVertex, nextVertex, matrix[nearestVertex][nextVertex],

prevVertex, nextVertex, matrix[prevVertex][nextVertex]

);

System.out.printf("Новый путь: %s, Общая стоимость: %.1f%n", path, cost);

}

visited.add(nearestVertex);

} else {

System.out.println("\n!!! Не найдено допустимой вершины для вставки !!!");

return new Object[]{null, Double.POSITIVE\_INFINITY};

}

}

if (matrix[path.get(path.size() - 1)][start] != 0) {

cost += matrix[path.get(path.size() - 1)][start];

path.add(start);

System.out.printf(

"\nЗамыкаем цикл: %d -> %d (вес %d). Итоговый путь: %s, Итоговая стоимость: %.1f%n",

path.get(path.size() - 2), start, matrix[path.get(path.size() - 2)][start], path, cost

);

return new Object[]{path, cost};

}

System.out.println("\n!!! Невозможно вернуться в начальную вершину !!!");

return new Object[]{null, Double.POSITIVE\_INFINITY};

}

public static void visualizeGraph(int[][] matrix, List<Integer> path, String filename) {

try {

StringBuilder dot = new StringBuilder();

dot.append("digraph G {\n"); // Начало графа (ориентированный)

dot.append(" rankdir=LR;\n"); // Ориентация: слева направо

dot.append(" node [shape=circle];\n"); // Форма вершин - круги

// Добавляем все вершины

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

dot.append(String.format(" %d;\n", i));

}

// Запоминаем рёбра из оптимального пути

Set<String> pathEdges = new HashSet<>();

if (path != null && path.size() > 1) {

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

int from = path.get(i);

int to = path.get(i + 1);

pathEdges.add(from + "->" + to);

}

}

//Добавляем рёбра с весами

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

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

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

String edge = String.format(" %d -> %d [label=\"%d\"", i, j, matrix[i][j]);

if (pathEdges.contains(i + "->" + j)) {

edge += " color=red penwidth=2.0"; // Красные и толстые

}

edge += "];\n";

dot.append(edge);

}

}

}

dot.append("}\n");

// Сохраняем DOT-файл

String dotFile = filename + ".dot";

try (PrintWriter out = new PrintWriter(dotFile)) {

out.println(dot.toString());

}

// Генерируем PNG с помощью Graphviz

String[] cmd = {"dot", "-Tpng", dotFile, "-o", filename + ".png"};

Process p = Runtime.getRuntime().exec(cmd);

// Ожидаем завершения процесса с таймаутом (30 секунд)

if (!p.waitFor(30, TimeUnit.SECONDS)) {

p.destroy();

throw new RuntimeException("Превышено время ожидания генерации графа");

}

// Проверяем код завершения

if (p.exitValue() != 0) {

throw new RuntimeException("Ошибка генерации графа. Убедитесь, что Graphviz установлен");

}

System.out.println("Граф сохранен в файл: " + filename + ".png");

} catch (IOException | InterruptedException e) {

System.err.println("Ошибка при визуализации графа: " + e.getMessage());

if (e instanceof InterruptedException) {

Thread.currentThread().interrupt(); // Восстанавливаем флаг прерывания

}

}

}

// Основная программа

public static void main(String[] args) {

String fileName = "matrix.txt";

int startVertex = 0;

int[][] matrix = null;

Scanner scanner = new Scanner(System.in);

List<Integer> bestPath = null;

while (true) {

System.out.println("\n1 - Сгенерировать несимметричную матрицу и сохранить матрицу в файл");

System.out.println("2 - Сгенерировать симметричную матрицу и сохранить матрицу в файл");

System.out.println("3 - Выбрать первую вершину (по умолчанию 0)");

System.out.println("4 - Загрузить матрицу из файла и запустить алгоритмы");

System.out.println("5 - Вывести текущую матрицу на экран");

System.out.println("6 - Визуализировать граф с оптимальным путем");

System.out.println("7 - Закончить программу");

System.out.print("Выберите опцию: ");

int option = scanner.nextInt();

try {

if (option == 1 || option == 2) {

System.out.print("Введите количество вершин графа: ");

int numberOfVertices = scanner.nextInt();

matrix = generateWeightMatrix(numberOfVertices, option == 2);

saveWeightMatrix(fileName, matrix);

} else if (option == 3) {

System.out.print("Введите начальную вершину: ");

startVertex = scanner.nextInt();

} else if (option == 4) {

matrix = loadWeightMatrix(fileName);

// Запуск алгоритмов

try (PrintStream fileOut = new PrintStream(new FileOutputStream("output.txt"))) {

System.setOut(fileOut);

// Решение методом ветвей и границ

Object[] bbResult = branchAndBound(matrix, startVertex);

bestPath = (List<Integer>) bbResult[0];

double bbCost = (double) bbResult[1];

// Решение методом АВБГ

Object[] niResult = nearestInsertion(matrix, startVertex);

@SuppressWarnings("unchecked")

List<Integer> niPath = (List<Integer>) niResult[0];

double niCost = (double) niResult[1];

System.setOut(new PrintStream(new FileOutputStream(FileDescriptor.out)));

// Вывод результата МВиГ

if (bestPath != null && bbCost != Double.POSITIVE\_INFINITY) {

System.out.printf("%nМетод ветвей и границ: Путь = %s, Стоимость = %.1f%n", bestPath, bbCost);

} else {

System.out.println("\nМетод ветвей и границ: Путь не найден");

}

// Вывод результата АВБГ

if (niPath != null && niCost != Double.POSITIVE\_INFINITY) {

System.out.printf("Метод АВБГ: Путь = %s, Стоимость = %.1f%n", niPath, niCost);

} else {

System.out.println("Метод АВБГ: Путь не найден");

}

} catch (IOException e) {

System.err.println("Ошибка при записи в файл: " + e.getMessage());

}

} else if (option == 5) {

if (matrix != null) {

System.out.println("\nТекущая матрица весов:");

for (int[] row : matrix) {

for (int value : row) {

System.out.printf("%3d", value);

}

System.out.println();

}

} else {

System.out.println("Матрица еще не загружена или не сгенерирована!");

}

} else if (option == 6) {

if (matrix != null) {

if (bestPath != null) {

visualizeGraph(matrix, bestPath, "graph");

// Попробуем открыть изображение

try {

if (System.getProperty("os.name").toLowerCase().contains("win")) {

Runtime.getRuntime().exec("cmd /c start graph.png");

} else if (System.getProperty("os.name").toLowerCase().contains("mac")) {

Runtime.getRuntime().exec("open graph.png");

} else {

Runtime.getRuntime().exec("xdg-open graph.png");

}

} catch (IOException e) {

System.out.println("Не удалось открыть изображение автоматически. Файл: graph.png");

}

} else {

System.out.println("Сначала запустите алгоритмы (опция 4) для нахождения пути!");

}

} else {

System.out.println("Матрица еще не загружена или не сгенерирована!");

}

}else if (option == 7){

break;

}

} catch (Exception e) {

System.err.println("Ошибка: " + e.getMessage());

}

}

}

}