**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №2**

**по дисциплине «Построение и анализ алгоритмов»**

**Тема: Кратчайшие пути в графе: коммивояжёр. Вариант 4.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3343 |  | Старков С. А. |
| Преподаватель |  | Жангиров Т. Р. |

Санкт-Петербург

2025

## Цель работы

Разработать и реализовать два алгоритма для решения задачи коммивояжёра: точный метод ветвления с отсечением (МВиГ) и приближённый метод модификации решения (АВБГ), с использованием эвристик для ускорения поиска.

## Задание

4 МВиГ: последовательный рост пути + использование для отсечения двух нижних оценок веса оставшегося пути: 1) полусуммы весов двух легчайших рёбер по всем кускам; 2) веса МОД. Эвристика выбора дуги — не в глубину, а по антиприоритету (S/k+L/N)(4N/(3N+k)). Приближённый алгоритм: АВБГ "улучшенный". Замечание к варианту 4 И МВиГ, и АВБГ "улучшенный" начинать со стартовой вершины.

### Выполнение работы

Для решения задачи коммивояжёра в данной лабораторной работе были реализованы два подхода: метод ветвей и границ, а также приближённый жадный алгоритм с улучшением по методу АВБГ (улучшенный). Каждый из подходов инкапсулирован в собственный класс.

#### **1. Класс TSP (Приближённый алгоритм АВБГ)**

#### Реализует жадную эвристику с пошаговой вставкой городов в оптимальную позицию.

#### Особенности:

#### Начальный путь: Начинается с города 0.

#### Итеративная вставка: На каждом шаге выбирается город и позиция, минимизирующие приращение стоимости: java

#### double increase = I + O - R; // I (входящее ребро), O (исходящее), R (заменяемое)

#### Критерий выбора: Минимальное увеличение стоимости (minIncrease).

#### Завершение: Построение полного цикла с возвратом в начальный город.

#### Сложность:

#### Временная: O(n^3) (для каждой из nn вершин проверяется O(n) позиций).

#### Пространственная: O(n).

#### 2. Метод ветвей и границ

#### Класс TSPBranchAndBound реализует метод с приоритетной очередью частичных решений.

#### Особенности:

#### Приоритетная очередь: Хранит состояния (State) с учётом антиприоритета.

#### Оценка нижней границы:

#### Полусумма минимальных рёбер: Для каждого города учитываются два минимальных входящих и исходящих ребра.

#### Минимальное остовное дерево (MST): Используется для оценки оставшегося пути.

#### java

#### lowerBound = Math.max(calculateHalfSumMinEdges(), calculateMST());

#### **Антиприоритет**: Для выбора состояния из очереди используется формула:

#### java

#### priority = (S + L) / (0.5 \* N + k); // S — текущая стоимость, L — оценка остатка, k — длина пути

#### **Отсечение ветвей**: Ветви с оценкой выше текущего лучшего решения игнорируются.

#### **Сложность**:

#### Временная: O(b⋅n^3), где b — количество рассматриваемых состояний (экспоненциально в худшем случае).

#### Пространственная: O(b⋅n).

#### **3. Класс State (Состояние для ветвей и границ)**

#### Инкапсулирует частичное решение и метаданные для оценки его перспективности.

#### Поля:

#### path: Текущий частичный путь.

#### visited: Посещённые города.

#### lowerBound: Нижняя оценка стоимости полного пути.

#### priority: Рассчитывается по формуле варианта 4 для управления порядком в очереди.

#### Методы:

#### calculateLowerBound(): Комбинирует оценку через полусумму рёбер и MST.

#### calculatePriority(): Вычисляет антиприоритет для оптимизации выбора пути**.**

***4. Главный класс Main***

Обеспечивает взаимодействие с пользователем и управление алгоритмами.

Функционал:

* 1. Генерация матрицы
  2. Симметричные/несимметричные матрицы с диагональю -1.
  3. Случайные веса рёбер в диапазоне 0–99.
  4. Загрузка из файла
  5. Выбор алгоритма: Пользователь может запустить либо метод ветвей и границ, либо улучшенный АВБГ.

## Тестирование

Результаты тестирования представлены в таблице 1.

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
| 1. |  | -1 3 4 1  1 -1 3 4  9 2 -1 4  8 9 2 -1  Проверяем что можно вставить город : 1  -Проверяем позицию 1 для города 1  Пытаемся вычислить prev 0  Пытаемся вычислить next -1  Пытаемся обновить лучший выбор - increase: 3.0 mincrease: 1.7976931348623157E308  Обновленные данные: mincrease 1.7976931348623157E308 bestCity: -1 bestPos: -1  Проверяем что можно вставить город : 2  -Проверяем позицию 1 для города 2  Пытаемся вычислить prev 0  Пытаемся вычислить next -1  Пытаемся обновить лучший выбор - increase: 4.0 mincrease: 3.0  Проверяем что можно вставить город : 3  -Проверяем позицию 1 для города 3  Пытаемся вычислить prev 0  Пытаемся вычислить next -1  Пытаемся обновить лучший выбор - increase: 1.0 mincrease: 3.0  Обновленные данные: mincrease 3.0 bestCity: 1 bestPos: 1  Проверяем что можно вставить город : 1  -Проверяем позицию 1 для города 1  Пытаемся вычислить prev 0  Пытаемся вычислить next 3  Пытаемся обновить лучший выбор - increase: 6.0 mincrease: 1.7976931348623157E308  Обновленные данные: mincrease 1.7976931348623157E308 bestCity: -1 bestPos: -1  -Проверяем позицию 2 для города 1  Пытаемся вычислить prev 3  Пытаемся вычислить next -1  Пытаемся обновить лучший выбор - increase: 9.0 mincrease: 6.0  Проверяем что можно вставить город : 2  -Проверяем позицию 1 для города 2  Пытаемся вычислить prev 0  Пытаемся вычислить next 3  Пытаемся обновить лучший выбор - increase: 7.0 mincrease: 6.0  -Проверяем позицию 2 для города 2  Пытаемся вычислить prev 3  Пытаемся вычислить next -1  Пытаемся обновить лучший выбор - increase: 2.0 mincrease: 6.0  Обновленные данные: mincrease 6.0 bestCity: 1 bestPos: 1  Проверяем что можно вставить город : 1  -Проверяем позицию 1 для города 1  Пытаемся вычислить prev 0  Пытаемся вычислить next 3  Пытаемся обновить лучший выбор - increase: 6.0 mincrease: 1.7976931348623157E308  Обновленные данные: mincrease 1.7976931348623157E308 bestCity: -1 bestPos: -1  -Проверяем позицию 2 для города 1  Пытаемся вычислить prev 3  Пытаемся вычислить next 2  Пытаемся обновить лучший выбор - increase: 10.0 mincrease: 6.0  -Проверяем позицию 3 для города 1  Пытаемся вычислить prev 2  Пытаемся вычислить next -1  Пытаемся обновить лучший выбор - increase: 2.0 mincrease: 6.0  Обновленные данные: mincrease 6.0 bestCity: 1 bestPos: 1  Оптимальный путь: [0, 3, 2, 1]  Лучшая цена: 6 | АБВГ |
| 2. |  | -1 3 4 1  1 -1 3 4  9 2 -1 4  8 9 2 -1  Достаем часть пути из очереди: [0] с нижней оценкой 6  Перебираем лучшие пути: [0] + новый город 1  Создаем маршрут [0, 1]  Проверяем нижнюю оценку - если плохая отсекаем путь и не добавляем его в очередь  Нижняя оценка хорошая, добавляем в очередь  Перебираем лучшие пути: [0] + новый город 2  Создаем маршрут [0, 2]  Проверяем нижнюю оценку - если плохая отсекаем путь и не добавляем его в очередь  Нижняя оценка хорошая, добавляем в очередь  Перебираем лучшие пути: [0] + новый город 3  Создаем маршрут [0, 3]  Проверяем нижнюю оценку - если плохая отсекаем путь и не добавляем его в очередь  Нижняя оценка хорошая, добавляем в очередь  Достаем часть пути из очереди: [0, 3] с нижней оценкой 6  Перебираем лучшие пути: [0, 3] + новый город 1  Создаем маршрут [0, 3, 1]  Проверяем нижнюю оценку - если плохая отсекаем путь и не добавляем его в очередь  Нижняя оценка хорошая, добавляем в очередь  Перебираем лучшие пути: [0, 3] + новый город 2  Создаем маршрут [0, 3, 2]  Проверяем нижнюю оценку - если плохая отсекаем путь и не добавляем его в очередь  Нижняя оценка хорошая, добавляем в очередь  Достаем часть пути из очереди: [0, 3, 2] с нижней оценкой 6  Перебираем лучшие пути: [0, 3, 2] + новый город 1  Создаем маршрут [0, 3, 2, 1]  Проверяем нижнюю оценку - если плохая отсекаем путь и не добавляем его в очередь  Нижняя оценка хорошая, добавляем в очередь  Достаем часть пути из очереди: [0, 3, 2, 1] с нижней оценкой 7  Проверяем завершенный путь и считаем его стоимость: [0, 3, 2, 1]  Лучшая стоимость пути: 6.0  Путь оказался лучшим обновляем стоимость  Достаем часть пути из очереди: [0, 1] с нижней оценкой 7  Перебираем лучшие пути: [0, 1] + новый город 2  Создаем маршрут [0, 1, 2]  Проверяем нижнюю оценку - если плохая отсекаем путь и не добавляем его в очередь  Нижняя оценка плохая - не добавляем  Перебираем лучшие пути: [0, 1] + новый город 3  Создаем маршрут [0, 1, 3]  Проверяем нижнюю оценку - если плохая отсекаем путь и не добавляем его в очередь  Нижняя оценка плохая - не добавляем  Достаем часть пути из очереди: [0, 2] с нижней оценкой 8  Перебираем лучшие пути: [0, 2] + новый город 1  Создаем маршрут [0, 2, 1]  Проверяем нижнюю оценку - если плохая отсекаем путь и не добавляем его в очередь  Нижняя оценка плохая - не добавляем  Перебираем лучшие пути: [0, 2] + новый город 3  Создаем маршрут [0, 2, 3]  Проверяем нижнюю оценку - если плохая отсекаем путь и не добавляем его в очередь  Нижняя оценка плохая - не добавляем  Достаем часть пути из очереди: [0, 3, 1] с нижней оценкой 14  Лучший путь: [0, 3, 2, 1]  Лучшая стоимость: 6.0 | Метод Ветвей И Границ |

Табл. 1. – Результаты тестирования

## Выводы В ходе выполнения работы были реализованы и исследованы два подхода к решению задачи коммивояжёра: **точный алгоритм ветвей и границ (МВиГ)** и **приближённый жадный алгоритм с улучшением (АВБГ)**. Оба метода продемонстрировали свои преимущества и ограничения в зависимости от характеристик задачи.

### ****1. Точный метод ветвей и границ****

* **Преимущества**:
  + Гарантирует нахождение **оптимального решения** за счёт полного перебора с отсечением неперспективных ветвей.
  + Использует **нижние оценки** (полусумма минимальных рёбер и вес минимального остовного дерева) для сокращения пространства поиска.
  + Эффективно работает для **небольших матриц** (до 15 вершин).
* **Недостатки**:
  + **Экспоненциальная сложность** O(n!), что делает метод неприменимым для задач с n>20.
  + Высокие требования к **памяти** из-за хранения множества частичных решений в приоритетной очереди.

### ****2. Приближённый алгоритм АВБГ****

* **Преимущества**:
  + **Полиномиальная сложность** O(n^3), что позволяет обрабатывать **крупные матрицы** (сотни вершин).
  + Быстро находит **близкие к оптимальным решения** за счёт жадной вставки городов в оптимальные позиции.
  + Низкие требования к памяти (O(n)).
* **Недостатки**:
  + **Не гарантирует оптимальность** решения, особенно для матриц с неочевидной структурой.
  + Риск застревания в **локальных минимумах**, если начальный путь далёк от оптимального.

### ****3. Сравнительный анализ****

| **Критерий** | **Метод ветвей и границ** | **АВБГ** |
| --- | --- | --- |
| **Точность** | Точное решение | Приближённое решение |
| **Сложность (время)** | O(n!) | O(n^3) |
| **Память** | O(b⋅n) | O(n) |

# Приложение А

package org.piaa;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import java.util.Random;

import java.util.Scanner;

public class Main {

private static final Logger log = LogManager.getLogger(Main.class);

private static TSPBranchAndBound tspBranchAndBound = new TSPBranchAndBound();

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.println("Выберите опцию:");

System.out.println("1. Сгенерировать матрицу:");

System.out.println("2. Выбрать матрицу из файла матрицу:");

System.out.printf("Ваш выбор: ");

int choice = scanner.nextInt();

switch (choice) {

case 1:

System.out.printf("Введите размер матрицы: ");

int size = scanner.nextInt();

System.out.printf("Выберите тип матрицы 1 ( - симметричная, 2 - несимметричная): ");

int isSymmetric = scanner.nextInt();

int[][] matrix = generateMatrix(size, isSymmetric);

System.out.println("Какой метод вы хотите использовать? ");

System.out.println("1. Метод ветвей и границ");

System.out.println("2. АВБГ");

int choice2 = scanner.nextInt();

switch (choice2){

case 1:

printMatrix(matrix);

tspBranchAndBound.solve(matrix);

break;

case 2:

printMatrix(matrix);

TSP tsp = new TSP(matrix);

tsp.improvedAVBG();

break;

}

break;

case 2:

System.out.printf("Введите путь до файла: ");

String pathToFile = scanner.next();

try {

int[][] matrixToSolve = readMatrixFromFile(pathToFile);

System.out.println("Какой метод вы хотите использовать? ");

System.out.println("1. Метод ветвей и границ");

System.out.println("2. АВБГ");

int choice3 = scanner.nextInt();

switch (choice3){

case 1:

printMatrix(matrixToSolve);

tspBranchAndBound.solve(matrixToSolve);

break;

case 2:

printMatrix(matrixToSolve);

TSP tsp = new TSP(matrixToSolve);

tsp.improvedAVBG();

break;

}

} catch (IOException e) {

System.out.println("Неверное имя файла");

}

break;

}

}

private static void printMatrix(int[][] matrix) {

System.out.println("Ваша матрица:");

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

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

System.out.print(matrix[i][j] + " ");

}

System.out.println();

}

}

private static int[][] generateMatrix(int size, int isSymmetric) {

int[][] matrix = new int[size][size];

Random random = new Random();

if (isSymmetric == 1) {

// Генерация симметричной матрицы с диагональю -1

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

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

if (i == j) {

matrix[i][j] = -1; // Диагональный элемент

} else {

int value = random.nextInt(100);

matrix[i][j] = value;

matrix[j][i] = value; // Зеркальное отражение

}

}

}

} else {

// Генерация обычной матрицы с диагональю -1

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

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

matrix[i][j] = (i == j) ? -1 : random.nextInt(100);

}

}

}

return matrix;

}

public static int[][] readMatrixFromFile(String filename) throws IOException {

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

// Читаем размер матрицы

String sizeLine = reader.readLine();

if (sizeLine == null) {

throw new IOException("Файл пустой");

}

int size;

try {

size = Integer.parseInt(sizeLine.trim());

} catch (NumberFormatException e) {

throw new IOException("Некорректный размер матрицы: " + sizeLine, e);

}

if (size <= 0) {

throw new IOException("Размер матрицы должен быть положительным числом: " + size);

}

// Инициализируем матрицу

int[][] matrix = new int[size][size];

// Читаем строки матрицы

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

String line = reader.readLine();

if (line == null) {

throw new IOException("Недостаточно строк в файле. Ожидалось: " + size + ", получено: " + i);

}

String[] elements = line.trim().split("\\s+");

if (elements.length != size) {

throw new IOException("Неправильное количество элементов в строке " + (i + 1) +

". Ожидалось: " + size + ", получено: " + elements.length);

}

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

try {

matrix[i][j] = Integer.parseInt(elements[j]);

} catch (NumberFormatException e) {

throw new IOException("Некорректное число в строке " + (i + 1) + ": " + elements[j], e);

}

}

}

return matrix;

}

}

}

package org.piaa;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

import java.util.\*;

public class State implements Comparable<State> {

private static final Logger log = LogManager.getLogger(State.class);

int[][] costMatrix;

public ArrayList<Integer> path; // Текущий частичный путь

boolean[] visited; // Посещенные города

int n; // количество дорог?

int cost; // Текущая стоимость пути

int lowerBound; // Нижняя оценка стоимости

double priority;

public State(int [][] costMatrix,List<Integer> path, boolean[] visited) {

this.costMatrix = costMatrix;

this.path = new ArrayList<>(path);

this.visited = Arrays.copyOf(visited, visited.length);

this.cost = calculateCost(path);

this.n = costMatrix.length - 1; // кол-во городов

this.lowerBound = cost + calculateLowerBound(); // Вычисляем нижнюю оценку

this.priority = calculatePriority();

}

private double calculatePriority() {

int k = path.size() - 1; // Количество дуг (рёбер)

int N = n; // Общее количество городов

// S - текущая стоимость пути

double S = cost;

// L - нижняя оценка остатка пути (lowerBound - S)

double L = lowerBound - S;

// Формула антиприоритета из варианта 4

double denominator = 0.5 \* N + k;

double priority = (S + L) / denominator;

return priority;

}

public int calculateCost(List<Integer> path) {

int total = 0;

int size = (path == null) ? 1: path.size();

if (path.size() > 1) {

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

total += costMatrix[path.get(i)][path.get(i + 1)];

}

}

return total;

}

@Override

public String toString() {

StringBuilder sb = new StringBuilder();

sb.append("\n--------- state ---------\n");

sb.append( "path : " + path.toString() + "\n");

sb.append("visited : ");

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

sb.append(visited[i] + " ");

}

sb.append("\n");

sb.append( "n: "+ n + "\n");

sb.append( "cost : " + cost + "\n");

sb.append( "lowerBound : " + lowerBound + "\n");

sb.append( "priority : " + priority + "\n");

sb.append("--------- state ---------\n");

return sb.toString();

}

private int calculateLowerBound() {

int sumMinEdges = calculateHalfSumMinEdges() ; // Полусумма ребер

log.info("--- sumMinEdges : " + sumMinEdges);

int mstWeight = calculateMST(); // Вес МОД

log.info("--- mstWeight : " + mstWeight);

return Math.max(sumMinEdges, mstWeight);

}

private int calculateMST() {

int startVertex = path.get(0);

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

Set<Integer> visited = new HashSet<>(path);

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

return 0;

}

PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[1]));

Set<Integer> inMST = new HashSet<>();

int weight = 0;

pq.add(new int[]{currentVertex, 0});

while (inMST.size() < n - visited.size() + 1 && !pq.isEmpty()) {

int[] entry = pq.poll();

int vertex = entry[0];

int edgeWeight = entry[1];

if (inMST.contains(vertex)) {

continue;

}

inMST.add(vertex);

weight += edgeWeight;

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

if(v != vertex && (!visited.contains(v) || v == startVertex)

&& (costMatrix[vertex][v] != -1 || costMatrix[vertex][v] != Integer.MAX\_VALUE)) {

pq.add(new int[]{v, costMatrix[vertex][v]});

}

}

}

return inMST.size() == n - visited.size() + 1 ? weight : Integer.MAX\_VALUE;

}

public int calculateHalfSumMinEdges() {

int sum = 0;

int startVertex = path.size() > 1 ? 0: path.get(0) ; // null

int endVertex = path.size() > 1 ? 0 : path.get(path.size() - 1) ; // null

TreeSet<Integer> chunksIn = new TreeSet<>();// [0 , 1] (0, 1)

TreeSet<Integer> chunksOut = new TreeSet<>();// [0 , 1] (0, 1)

int minIn = 0;

int minOut = 0;

if (path.size() > 1) {

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

if(visited[j] != true && costMatrix[endVertex][j] != -1) {

chunksOut.add(costMatrix[endVertex][j]);

}

}

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

if(costMatrix[startVertex][j] != -1){

chunksIn.add(costMatrix[startVertex][j]);

}

}

}

else{

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

if(costMatrix[0][j] != -1){

chunksOut.add(costMatrix[0][j]);

}

}

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

if (costMatrix[j][0] != -1){

chunksIn.add(costMatrix[j][0]);

}

}

}

Integer minOutInt1 = chunksOut.size() > 0 ? chunksOut.first(): 0;

Integer minOutInt2 = chunksOut.size() >= 2 ? chunksOut.higher(minOutInt1): 0;

Integer minInInt1 = chunksIn.size() > 0? chunksIn.first() : 0;

Integer minInInt2 = chunksIn.size() >= 2 ? chunksIn.higher(minInInt1): 0;

sum = ((minInInt1 + minInInt2) + (minOutInt2 + minOutInt1))/2;

return sum;

}

@Override

public int compareTo(State other) {

return Double.compare(this.priority, other.priority);

}

}

package org.piaa;

import java.util.\*;

public class TSP {

private int[][] cost; // Матрица стоимостей перемещения между городами

private int n; // Количество городов

private List<Integer> path; // Текущий построенный путь

private boolean[] visited; // Посещенные города

public TSP(int[][] costMatrix) {

this.cost = costMatrix;

this.n = costMatrix.length;

this.path = new ArrayList<>();

this.visited = new boolean[n];

}

public List<Integer> improvedAVBG() {

path.add(0); // Начинаем с города 0 (по условию варианта)

visited[0] = true;

// Пока не все города добавлены в путь

while (path.size() < n) {

double minIncrease = Double.MAX\_VALUE;

int bestCity = -1;

int bestPos = -1;

// Перебираем все города

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

if (visited[city]) continue; // Пропускаем уже посещенные

System.out.println("Проверяем что можно вставить город : " + city);

// Проверяем все возможные позиции для вставки

for (int pos = 1; pos <= path.size(); pos++) {

System.out.println("-Проверяем позицию " + pos + " для города " + city);

// Определяем предыдущий и следующий города в текущем пути

int prev = (pos == 0) ? -1 : path.get(pos - 1);

System.out.println("Пытаемся вычислить prev " + prev);

int next = (pos == path.size()) ? -1 : path.get(pos);

System.out.println("Пытаемся вычислить next " + next);

// Вычисляем стоимости

int I = (prev == -1) ? 0 : cost[prev][city]; // Входящее ребро

int O = (next == -1) ? 0 : cost[city][next]; // Исходящее ребро

int R = (prev != -1 && next != -1) ? cost[prev][next] : 0; // Заменяемое ребро

// Пропускаем недопустимые ребра (стоимость -1)

if (I == -1 || O == -1 || R == -1) continue;

// Вычисляем приращение стоимости

double increase = I + O - R;

// Обновляем лучший выбор

System.out.println("Пытаемся обновить лучший выбор - increase: "+ increase + " mincrease: " + minIncrease);

if (increase < minIncrease ) {

System.out.println("Обновленные данные: mincrease " + minIncrease + " bestCity: " + bestCity + " bestPos: " + bestPos);

minIncrease = increase;

bestCity = city;

bestPos = pos;

}

}

}

// Вставляем город в оптимальную позицию

if (bestCity != -1) {

path.add(bestPos, bestCity);

visited[bestCity] = true;

}

}

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

System.out.println("Лучшая цена: " + calculateCost(path));

return path;

}

public int calculateCost(List<Integer> path) {

int total = 0;

int size = (path == null) ? 1: path.size();

if (path.size() > 1) {

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

total += cost[path.get(i)][path.get(i + 1)];

}

total += cost[0][path.size() - 1];

}

return total;

}

}

package org.piaa;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

import org.apache.logging.log4j.core.util.JsonUtils;

import java.util.\*;

public class TSPBranchAndBound {

private static final Logger log = LogManager.getLogger(TSPBranchAndBound.class);

private int n; // Количество городов

private int[][] costMatrix;

private PriorityQueue<State> statePriorityQueue = new PriorityQueue<>();

private double bestCost = Double.MAX\_VALUE;

private List<Integer> bestPath;

public TSPBranchAndBound() {

}

public void solve(int[][] costMatrix) {

// Инициализация начального состояния (город 0)

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

initialPath.add(0);

boolean[] visited = new boolean[costMatrix.length];

visited[0] = true;

statePriorityQueue.add(new State(costMatrix,initialPath, visited));

while (!statePriorityQueue.isEmpty()) {

State current = statePriorityQueue.poll();

System.out.println("Достаем часть пути из очереди: " + current.path + " с нижней оценкой " + current.lowerBound);

log.info("state priority queue : " + statePriorityQueue.toString());

log.info("current state : " + current.toString());

if( current.cost >= bestCost ) {

log.info("- skipped current.lowerBound : " + current.lowerBound);

log.info("- skipped current.costMatrix : " + current.cost);

continue;

}

log.info("- NONskipped current.lowerBound : " + current.lowerBound);

log.info("- NONskipped current.costMatrix : " + current.cost);

// Если путь завершен, проверяем его стоимость

if (current.path.size() == costMatrix.length) {

int last = current.path.get(current.path.size() - 1);

if (costMatrix[last][0] != Integer.MAX\_VALUE || costMatrix[last][0] != -1) {

double total = current.cost + costMatrix[last][0];

System.out.println("Проверяем завершенный путь и считаем его стоимость: " + current.path);

log.info("-- total current.costMatrix : " + total);

log.info("-- current.costMatrix new : " + current.cost);

log.info("-- total current costMatrix : " + costMatrix[last][0]);

System.out.println("Лучшая стоимость пути: " +total);

if (total <= bestCost) {

System.out.println("Путь оказался лучшим обновляем стоимость");

log.info("--- total that lower : " + total);

bestCost = total;

bestPath = new ArrayList<>(current.path);

}

else {

System.out.println("Путь не самый лучший, идем дальше");

}

}

continue;

}

// Перебираем все возможные следующие города

for (int next = 0; next < costMatrix.length; next++) {

if (!current.visited[next]) {

System.out.println("Перебираем лучшие пути: " + current.path + " + новый город " + next);

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

newPath.add(next);

System.out.println("Создаем маршрут " + newPath);

boolean[] newVisited = Arrays.copyOf(current.visited, current.visited.length);

newVisited[next] = true;

State newState = new State(costMatrix, newPath, newVisited);

log.info("---- newState : " + newState.toString());

log.info("---- lowerBound : " + newState.lowerBound);

log.info("---- bestCost : " + bestCost);

// Отсекаем ветви, если оценка хуже текущего лучшего решения

System.out.println("Проверяем нижнюю оценку - если плохая отсекаем путь и не добавляем его в очередь");

if (newState.lowerBound <= bestCost ) {

System.out.println("Нижняя оценка хорошая, добавляем в очередь");

statePriorityQueue.add(newState);

}

else {

System.out.println("Нижняя оценка плохая - не добавляем");

}

}

}

}

System.out.println("Лучший путь: " + bestPath);

System.out.println("Лучшая стоимость: " + bestCost);

}

}