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 3.

Студент гр. 3343 _____ Поддубный В. 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 (АМР), с использованием эвристик для ускорения поиска.

Задание

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 — поиск в глубину с учётом веса добавляемой дуги и нижней оценки веса остатка пути. Приближённый алгоритм: АМР. Замечание к варианту 3 Начинать МВиГ 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 2-opt. Каждый из подходов инкапсулирован в собственную стратегию, реализующую общий интерфейс TspStrategy, что обеспечивает гибкость и расширяемость кода.

1. Интерфейс стратегии

Интерфейс TspStrategy содержит метод solve, который принимает матрицу расстояний и стартовую вершину, и возвращает пару: общую стоимость пути и последовательность посещения вершин. Это позволяет легко переключаться между различными алгоритмами решения, не изменяя остальной код.

2. Метод ветвей и границ

Класс BranchAndBoundTspStrategy реализует точный алгоритм ветвей и границ. Основные особенности реализации:

- Используется оценка нижней границы стоимости маршрута с помощью минимума рёбер и оценки веса минимального остовного дерева (MST).
- Осуществляется рекурсивный обход дерева решений с отсечением неэффективных путей.
- Ветви, у которых оценка превышает текущую наименьшую стоимость, отсекаются, что значительно сокращает перебор.

Оценка сложности: в худшем случае алгоритм имеет экспоненциальную временную сложность O(n!), где n — количество вершин, однако за счёт отсечения (pruning) реальное время выполнения часто значительно меньше.

3. Приближённый алгоритм (2-opt)

В классе ApproximateTspStrategy реализован эвристический алгоритм:

- Изначальный путь строится линейно.
- На каждой итерации осуществляется проверка всех возможных пар перестановок двух вершин.
- Если после перестановки стоимость маршрута улучшается, путь обновляется.

Оценка сложности: каждая итерация улучшения требует O(n^2) операций, а общее количество итераций ограничено O(n^2). Таким образом, общая временная сложность алгоритма составляет O(n^4) в худшем случае. Однако на практике алгоритм быстро сходится к локальному минимуму.

4. Стратегия выбора и запуск

Главный класс Context инкапсулирует выбранную стратегию и запускает решение. Выбор стратегии (BNB или APPROX) осуществляется через перечисление ChosenStrategy.

5. Генерация и загрузка матрицы

Пользователю предоставляется возможность:

- либо сгенерировать симметричную или несимметричную матрицу заданного размера со случайными весами;

- либо загрузить матрицу из файла. Поддерживается обозначение бесконечного расстояния (INF), означающее отсутствие ребра между вершинами.

Также предусмотрено сохранение сгенерированной матрицы в текстовый файл.

Тестирование

Результаты тестирования представлены в таблице 1.

№ п/п	Входные данные	Выходные данные	Комментарии
1.		INF 80 98 21 92 INF 5 19 19 56 INF 48 24 1 12 INF === BNB Strategy === DFS at [0] (cost: 0) Lower bound = 101, current minCost = 2147483647 Go to 3 (edge: 21, newCost: 21) DFS at [0, 3] (cost: 21) Lower bound = 195, current minCost = 2147483647 Go to 1 (edge: 1, newCost: 22) DFS at [0, 3, 1] (cost: 22) Lower bound = 139, current minCost = 2147483647 Go to 2 (edge: 5, newCost: 27) DFS at [0, 3, 1, 2] (cost: 27) Complete path found: [0, 3, 1, 2, 0] with cost 46 Go to 2 (edge: 12, newCost: 33) DFS at [0, 3, 2] (cost: 33) Lower bound = 205, current minCost = 46 Pruned branch (bound >= minCost) Go to 1 (edge: 80, newCost: 80) Go to 2 (edge: 98, newCost: 98) BNB Strategy: 0 3 1 2 0 46.0 === APPROX Strategy === Initial path: [0, 1, 2, 3, 0] Initial cost: 157 Iteration 0 Trying swap (1, 2): cost = 197 Trying swap (1, 3): cost = 181 Trying swap (2, 3): cost = 130	

		Improved! Previous cost: 157, New cost: 130 Iteration 1 Trying swap (1, 2): cost = 46 Improved! Previous cost: 130, New cost: 46 Trying swap (1, 3): cost = 197 Trying swap (2, 3): cost = 181 Iteration 2 Trying swap (1, 2): cost = 130 Trying swap (1, 3): cost = 197 Trying swap (2, 3): cost = 181 Final path: [0, 3, 1, 2, 0] Final cost: 46 APPROX Strategy: 0 3 1 2 0 46.0	
2.		INF 4 100 INF === BNB Strategy === DFS at [0] (cost: 0) Lower bound = 104, current minCost = 2147483647 Go to 1 (edge: 4, newCost: 4) DFS at [0, 1] (cost: 4) Complete path found: [0, 1, 0] with cost 104 BNB Strategy: 0 1 0 104.0 === APPROX Strategy === Initial path: [0, 1, 0] Initial cost: 104 Iteration 0 Final path: [0, 1, 0] Final cost: 104 APPROX Strategy: 0 1 0 104.0Матрица весов: [0, 56, 12, 62, 77] [70, 0, 77, 94, 26] [52, 36, 0, 67, 20]	

		[17, 59, 55, 0, 27] [79, 59, 72, 96, 0] Решение AMP(приближённый метод): Путь: [0, 2, 1, 4, 3, 0] Стоимость: 187	
3.		INF 44 83 44 3 58 76 44 INF 33 39 67 87 21 83 33 INF 28 93 6 100 44 39 28 INF 13 71 44 3 67 93 13 INF 8 41 58 87 6 71 8 INF 6 76 21 100 44 41 6 INF BNB Strategy: 0 4 5 2 3 6 1 0 154.0 APPROX Strategy: 0 3 2 1 6 5 4 0 143.0	

Табл. 1. – Результаты тестирования

Выводы

В результате проделанной работы были разработаны и реализованы два алгоритма для решения задачи коммивояжёра: точный метод ветвления с отсечением (МВиГ) и приближённый метод модификации решения (АМР). Точный метод, основанный на рекурсивном переборе с использованием нижних оценок остатка пути, гарантирует нахождение оптимального решения, однако его экспоненциальная сложность делает его непрактичным для больших экземпляров задачи. При этом приближённый метод, использующий эвристику локальных модификаций для улучшения начального решения, демонстрирует полиномиальную сложность и позволяет быстро получать хорошие приближения оптимального маршрута, хотя и не всегда гарантирует точное решение. Таким образом, выбор метода зависит от конкретных требований к точности и объёму обрабатываемых данных: для небольших задач можно использовать точный метод, а для более крупных – приближённый алгоритм, осознавая, что он может давать решения, отличающиеся от оптимальных.

ПРИЛОЖЕНИЕ А

```
package ru.rectid

import ApproximateTspStrategy
import BranchAndBoundTspStrategy
import TspStrategy
import java.io.File
import java.util.*

enum class ChosenStrategy {
    BNB,
    APPROX
}

fun main() {
    println("Выберите опцию:")
    println("1. Сгенерировать матрицу")
    println("2. Загрузить матрицу из файла")
    print("Ваш выбор: ")
    val choice = readln().toInt()

    val matrix = when (choice) {
        1 -> {
            print("Введите размер матрицы: ")
            val size = readln().toInt()
            print("Выберите тип матрицы (1 - симметричная, 2 -
несимметричная): ")
            val symmetricInput = readln().toInt()
            val symmetric = symmetricInput == 1
            val matrix = generateMatrix(size, symmetric = symmet-
ric)
            print("Введите имя файла для сохранения: ")
            val fileName = readln()
            saveMatrixToFile(matrix, fileName)
            println("Матрица сохранена в $fileName")
            matrix
        }

        2 -> {
            print("Введите имя файла: ")
            val fileName = readln()
            loadMatrixFromFile(fileName)
        }

        else -> throw IllegalArgumentException("Неверный выбор")
    }

    print("Введите стартовую вершину (0..${matrix.size - 1}): ")
    val startVertex = readln().toInt()

    val context = Context(matrix)

    println("\n=== BNB Strategy ===")
    context.strategy = ChosenStrategy.BNB
    context.run(startVertex)

    println("\n=== APPROX Strategy ===")
```

```
        context.strategy = ChosenStrategy.APPROX
        context.run(startVertex)
    }


    fun readMatrix(): Array<IntArray> {
        val input = mutableListOf<String>()
        while (true) {
            val line = readLine() ?: break
            if (line.trim().isEmpty()) break
            input.add(line)
        }

        return input.map { line ->
            line.split(" ").map { it.toInt() }.toIntArray()
        }.toTypedArray()
    }

    class Context(private val matrix: Array<IntArray>) {
        lateinit var strategy: ChosenStrategy

        fun run(startVertex: Int) {
            val solver: TspStrategy = when (strategy) {
                ChosenStrategy.BNB -> BranchAndBoundTspStrategy()
                ChosenStrategy.APPROX -> ApproximateTspStrategy()
            }

            printResult(solver.solve(matrix, startVertex))
        }

        private fun printResult(result: Pair<Int, List<Int>>) {
            when (strategy) {
                ChosenStrategy.BNB -> println("BNB Strategy:")
                ChosenStrategy.APPROX -> println("APPROX Strategy:")
            }
            if (result.first == -1) {
                println("no path")
                return
            }
            println(result.second.joinToString(" "))
            println(result.first.toFloat())
        }
    }

    fun generateMatrix(n: Int, maxWeight: Int = 100, symmetric: Boolean
= true): Array<IntArray> {
        val rand = Random()
        val matrix = Array(n) { IntArray(n) { Int.MAX_VALUE } }

        for (i in 0 until n) {
            for (j in 0 until n) {
                if (i == j) continue
                matrix[i][j] = rand.nextInt(maxWeight) + 1
                if (symmetric) {
                    matrix[j][i] = matrix[i][j]
                }
            }
        }
```

```
        return matrix
    }


    fun saveMatrixToFile(matrix: Array<IntArray>, fileName: String) {
        File(fileName).printWriter().use { out ->
            matrix.forEach { row ->
                out.println(row.joinToString("   ")   {   if   (it   ==
Int.MAX_VALUE) "INF" else it.toString() })
            }
        }
    }

    fun loadMatrixFromFile(fileName: String): Array<IntArray> {
        val lines = File(fileName).readLines()
        return lines.map { line ->
            line.trim().split(" ").map {
                if (it == "INF") Int.MAX_VALUE else it.toInt()
            }.toIntArray()
        }.toTypedArray()
    }

    import java.util.*

    interface TspStrategy {
        fun  solve(matrix:  Array<IntArray>,  startVertex:  Int  =  0):
Pair<Int, List<Int>>
    }

    object Logger {
        var depth = 0
        var enabled = true

        fun log(message: String) {
            if (enabled) println("${"  ".repeat(depth)}$message")
        }

        inline fun <T> withIndent(block: () -> T): T {
            depth++
            val result = block()
            depth--
            return result
        }
    }


    class BranchAndBoundTspStrategy : TspStrategy {
        override fun solve(matrix: Array<IntArray>, startVertex: Int):
Pair<Int, List<Int>> {
            val n = matrix.size
            if (n == 0) return -1 to emptyList()
            if (n == 1) return 0 to listOf(startVertex)

            var minCost = Int.MAX_VALUE
            var bestPath = listOf<Int>()

            fun calculateHalfSumMinEdges(visited: Set<Int>): Int {
```

```
                var sum = 0
                for (i in 0 until n) {
                    if (i in visited && i != startVertex) continue
                    val edges = matrix[i].filterIndexed { index, _ ->
                        index != i && (index !in visited || index ==
startVertex) && matrix[i][index] != Int.MAX_VALUE
                    }.sorted()
                    sum += when {
                        edges.isEmpty() -> return Int.MAX_VALUE
                        edges.size == 1 -> edges[0]
                        else -> (edges[0] + edges[1]) / 2
                    }
                }
                return sum
            }

            fun calculateMSTWeight(visited: Set<Int>, currentVertex:
Int): Int {
                if (visited.size == n) return 0

                val pq = PriorityQueue<Pair<Int, Int>>(compareBy
{ it.second })
                val inMST = mutableSetOf<Int>()
                var weight = 0

                pq.add(currentVertex to 0)

                while (inMST.size < n - visited.size + 1 && pq.is-
NotEmpty()) {
                    val (vertex, edgeWeight) = pq.poll()
                    if (vertex in inMST) continue

                    inMST.add(vertex)
                    weight += edgeWeight

                    for (v in 0 until n) {
                        if (v != vertex && (!visited.contains(v) || v
== startVertex) && matrix[vertex][v] != Int.MAX_VALUE) {
                            pq.add(v to matrix[vertex][v])
                        }
                    }
                }

                return if (inMST.size == n - visited.size + 1) weight
else Int.MAX_VALUE
            }

            fun dfs(path: List<Int>, visited: Set<Int>, currentCost:
Int) {
                val currentVertex = path.last()
                Logger.log("DFS at $path (cost: $currentCost)")

                if (path.size == n) {
                    val returnCost = matrix[currentVertex][startVertex]
                    if (returnCost != Int.MAX_VALUE) {
                        val totalCost = currentCost + returnCost
                        Logger.withIndent {
```

```
                            Logger.log("Complete path found: ${path + startVertex} with cost $totalCost")
                        }
                        if (totalCost < minCost) {
                            minCost = totalCost
                            bestPath = path + startVertex
                        }
                    }
                    return
                }

                val halfSumEstimate = calculateHalfSumMinEdges(visited)
                val mstEstimate = calculateMSTWeight(visited, currentVertex)
                val lowerBound = currentCost + maxOf(halfSumEstimate, mstEstimate)

                Logger.withIndent {
                    Logger.log("Lower bound = $lowerBound, current minCost = $minCost")
                }

                if (lowerBound >= minCost) {
                    Logger.withIndent {
                        Logger.log("Pruned branch (bound >= minCost)")
                    }
                    return
                }

                val neighbors = (0 until n).filter {
                    it != currentVertex && it !in visited && matrix[currentVertex][it] != Int.MAX_VALUE
                }.sortedBy { matrix[currentVertex][it] }

                for (neighbor in neighbors) {
                    val newCost = currentCost + matrix[currentVertex][neighbor]
                    Logger.withIndent {
                        Logger.log("Go to $neighbor (edge: ${matrix[currentVertex][neighbor]}, newCost: $newCost)")
                    }
                    if (newCost < minCost) {
                        Logger.withIndent {
                            dfs(path + neighbor, visited + neighbor, newCost)
                        }
                    }
                }
            }


            dfs(listOf(startVertex), setOf(startVertex), 0)

            return if (minCost == Int.MAX_VALUE) -1 to emptyList()
            else minCost to bestPath
        }
    }
```

```
     class ApproximateTspStrategy : TspStrategy {
         override fun solve(matrix: Array<IntArray>, startVertex: Int):
Pair<Int, List<Int>> {
             val n = matrix.size
             if (n == 0) return -1 to emptyList()
             if (n == 1) return 0 to listOf(startVertex, startVertex)

             val path = mutableListOf<Int>()
             path.add(startVertex)

             for (i in 0 until n) {
                 if (i != startVertex) path.add(i)
             }

             path.add(startVertex)

             var currentCost = calculatePathCost(matrix, path)
             var improved: Boolean
             var iterations = 0
             val maxIterations = n * n

             Logger.log("Initial path: $path")
             Logger.log("Initial cost: $currentCost")

             do {
                 improved = false
                 Logger.log("Iteration $iterations")
                 Logger.withIndent {
                     for (i in 1 until n) {
                         for (j in i + 1 until n) {
                             val newPath = path.toMutableList()
                             newPath[i] = path[j].also { newPath[j] =
path[i] }

                             val newCost = calculatePathCost(matrix,
newPath)
                             Logger.log("Trying swap ($i, $j): cost =
$newCost")

                             if (newCost < currentCost) {
                                 Logger.log("Improved! Previous cost:
$currentCost, New cost: $newCost")
                                 path.clear()
                                 path.addAll(newPath)
                                 currentCost = newCost
                                 improved = true
                             }
                         }
                     }
                 }
                 iterations++
             } while (improved && iterations < maxIterations)

             Logger.log("Final path: $path")
             Logger.log("Final cost: $currentCost")


             return currentCost to path
```

```
        }

        private fun calculatePathCost(matrix: Array<IntArray>, path:
List<Int>): Int {
            var cost = 0
            for (i in 0 until path.size - 1) {
                val from = path[i]
                val to = path[i + 1]
                if (matrix[from][to] == Int.MAX_VALUE) return
Int.MAX_VALUE
                cost += matrix[from][to]
            }
            return cost
        }
    }
```