

МИНОБРНАУКИ РОССИИ
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.

Студент гр. 3343

Лихацкий В. Р.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Изучить принцип работы алгоритмов нахождения пути коммивояжера на графах.

Задание.

Вариант 1

Метод Ветвей и Границ: Алгоритм Литтла. Приближённый алгоритм: 2-приближение по МиД (Алгоритм двойного обхода минимального остовного дерева).Замечание к варианту 1 АДО МОД является 2-приближением только для евклидовой матрицы. Начинать обход МОД со стартовой вершины.

Независимо от варианта, при сдаче работы должна быть возможность генерировать матрицу весов (произвольную или симметричную; для варианта 4- симметричную), сохранять её в файл и использовать в качестве входных данных

Описание алгоритма Литтла.

Алгоритм Литтла- это алгоритм для решения задачи коммивояжера. Он основан на методе ветвей и границ. На вход подаётся матрица смежности графа. Сначала алгоритм проводит редукцию матрицы (находит минимальные элементы в каждой строке и вычитает их из каждого элемента строки, то же самое со столбцами). После этого происходит поиск “тяжёлого нуля”, рассматривается элемент в этой же строке и в этом же столбце, среди них выбирается минимальный. Далее происходит ветвление: из левой ветви удаляется строка и столбец содержащие “тяжёлый ноль”, попутно находя самый большой путь содержащий ребро и запрещая движение из конца этого пути в начало, чтобы не образовать цикл, в правой ветви элемент относительно которого проводилось ветвление становится ∞ . На каждом шаге запоминается его стоимость. При нахождении первого решения, делаем его минимально возможным, чтобы в дальнейшем отсекал ветви которые

заведомо больше минимального. В случае нахождения еще меньшего решения, уже оно становится минимальным. Это помогает избежать полного обхода дерева решений. Таким образом, в итоге получается оптимальное решение для данной матрицы смежности.

Оценка сложности по времени:

Поиск элемента со значением 0, который имеет наибольшее значение суммы минимальных элементов: $O(n^2)$

Редукция матрицы: $O(n^2)$

Вызов метода `solve`: в худшем случае происходит проход по всему бинарному дереву поиска, количество элементов в нем равно 2^{n-1} , где n - размерность матрицы смежности. На каждом уровне рекурсии выполняются операции поиска элемента со значением 0 и редукции матрицы, каждая из которых имеет сложность $O(n^2)$.

Следовательно, сложность рекурсивного вызова метода равна $O((2^{n-1}) * n^2)$. Таким образом, общая сложность алгоритма равна $O((2^{n-1}) * n^2)$.

Оценка сложности по памяти:

Рекурсивный вызов метода `method_Little`: в худшем случае происходит проход по всему бинарному дереву поиска, количество элементов в нем равно 2^{n-1} , где n - количество вершин в графе. На каждом уровне рекурсии создается копия матрицы смежности графа, что занимает $O(n^2)$ памяти. Следовательно, общая сложность по памяти равна $O((2^{n-1}) * n^2)$.

Описание реализованных классов

Класс LittleTSP

Реализует алгоритм Литтла.

Методы:

1. **Contstructor(matrix,start_city)**—Инициализирует объект класса:

- `matrix` — матрица весов графа;
- `start_city` — стартовая вершина (индекс начинается с 0);
- Настраивает начальные значения: `best_cost` (лучшая стоимость) и `best_path` (оптимальный путь).

2. **solve()**— Основной метод, реализующий алгоритм:

- Использует приоритетную очередь для хранения узлов (частичных путей);
- Расширяет узлы, добавляя непосещённые города;
- Обновляет лучшее решение при нахождении полного пути;
- Возвращает оптимальный путь (номера городов +1) и его стоимость.

3. **reduceMatrix(matrix)** – Редуцирует матрицу по столбцам и строкам.

Описание алгоритма АДО МОД.

Алгоритм приближенного решения задачи коммивояжера находит приближенное решение, основываясь на методе минимального остовного дерева (МОД) и проходя по графу МОД с помощью поиска в глубину. Сначала алгоритм находит МОД в заданном графе, используя алгоритм Прима. Затем на основе МОД строится новый граф, в котором каждая вершина соединена с ближайшей вершиной в МОД. Поиск в глубину используется для обхода построенного графа, начиная с заданной начальной вершины. На выходе алгоритм возвращает гамильтонов цикл, который

соответствует обходу графа МОД с помощью поиска в глубину, и стоимость этого цикла, которая вычисляется как сумма весов ребер на цикле.

Сложность по времени алгоритма:

Сложность алгоритма Прима в наихудшем случае составляет $O(V^2+E)$, где V - количество вершин в графе, E - количество ребер. Это происходит из-за необходимости просматривать все вершины в каждой итерации. Кроме того, на каждой итерации необходимо искать минимальное ребро из множества непосещённых вершин, что также занимает $O(V^2)$ времени. Сложность второго обхода по полученному дереву равна $O(V+E)$. Это происходит из-за необходимости просмотреть все вершины и ребра в дереве. Таким образом, общая сложность алгоритма равна $O(V^2 + E)$.

Сложность по памяти алгоритма:

Относительно памяти, в алгоритме Прима используется двумерный массив вершин размером n^2 , массив посещенных вершин размером n , массив ребер размером $n-1$. Во втором обходе используется массив пути по вершинам размером n . Таким образом, общая сложность по памяти составляет $O(n^2 + 3n) = O(n^2)$.

Класс ApproximateTSP: Реализует алгоритм для нахождения приближённого решения задачи коммивояжёра через построение минимального остовного дерева и поиск в глубину.

Constructor(matrix, start_city) -инициализирует объект класса, сохраняя исходную матрицу и создавая пустой список для остовного дерева.

findMst(self)

Реализует алгоритм Прима для построения минимального остовного дерева, начиная с заданной вершины.

dfsTraversal (mst)

Выполняет поиск в глубину (DFS) для обхода остоного дерева и построения пути.

solve()

Находит приближённое решение задачи коммивояжёра, используя минимальное остовное дерево и поиск в глубину, и возвращает путь.

Код программы смотреть в приложении А.

Тестирование.

Тестируем Алгоритм Литтла, потому что он даёт детерминированное решение.

Ввод	Вывод	Ожидаемый результат
<pre>[inf, 14, 20, 19, 18] [11, inf, 19, 18, 19] [12, 15, inf, 12, 18] [10, 13, 16, inf, 13] [12, 10, 10, 12, inf]</pre>	Лучший путь: [1, 5, 3, 4, 2] Минимал ьная стоимост ь: 64	Результат верный

<pre>[inf, 14, 19, 20, 14, 13, 13] [10, inf, 20, 13, 20, 15, 11] [14, 17, inf, 13, 10, 14, 16] [16, 13, 18, inf, 13, 16, 16] [10, 13, 14, 19, inf, 18, 15] [15, 20, 17, 13, 19, inf, 20] [16, 14, 12, 16, 17, 18, inf]</pre>	<p>Лучший путь: [1, 6, 4, 2, 7, 3, 5] Минимал ьная стоимост ь: 82</p>	Результат верный
<pre>[inf, 10, 13] [12, inf, 15] [10, 11, inf]</pre>	<p>Лучший путь: [1, 2, 3] Минимал ьная стоимост ь: 35</p>	Результат верный

Рисунок 2—Путь коммивояжёра

Исследование.

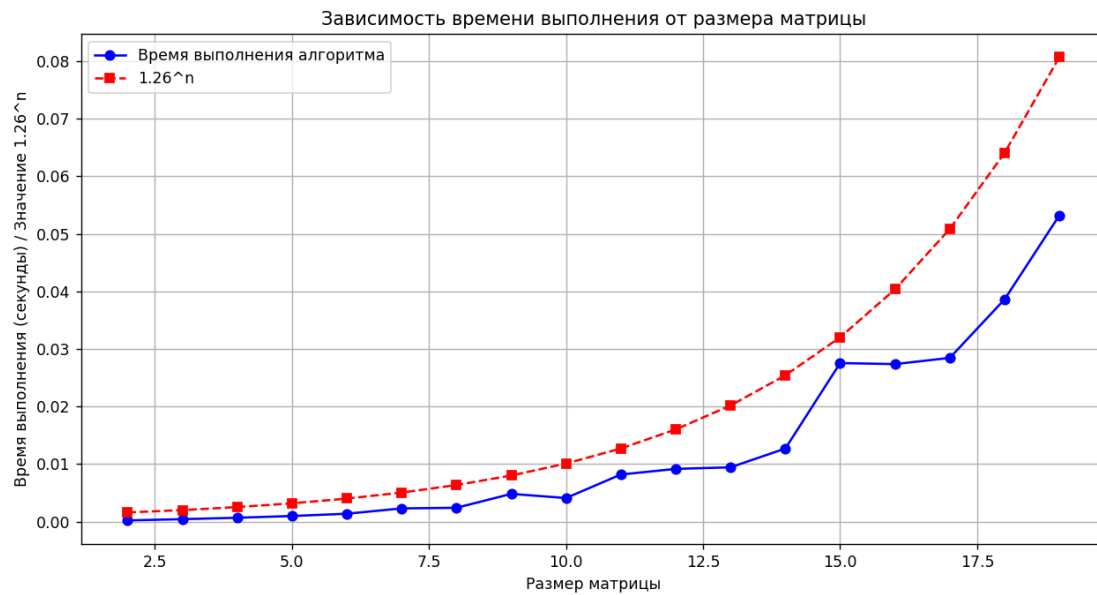


Рисунок 3 - График зависимости размера случайной матрицы от времени выполнения алгоритма Литтла

Экспериментальная сложность алгоритма примерно $O(c^n)$, где c примерно равно 1.26 (эмпирический результат).

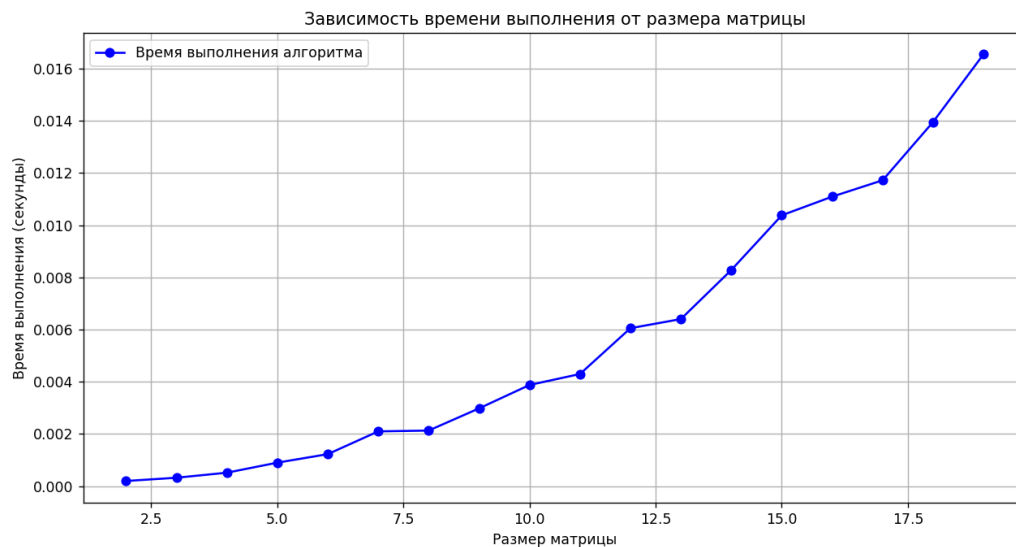


Рисунок 4 - График зависимости размера случайной матрицы от времени выполнения алгоритма АДО МОД

Можно увидеть, что алгоритм АДО МОД выполняется гораздо быстрее, но он не даёт точного решения. Этот алгоритм можно использовать для поиска нижней границы для алгоритма Литтла, который выполняется медленнее, но даёт самое оптимальное решение. Если матрица удовлетворяет неравенству треугольника, то алгоритм перестройки двойного обхода остоного дерева AST получает Гамильтонов цикл не более чем в 2 раза хуже оптимального для любого примера задачи коммивояжера.

Выводы.

В результате работы была написана программа, решающая поставленную задачу с использованием приближённого алгоритма АДО МОД и оптимального алгоритма Литтла. Программа была протестирована, результаты тестов совпали с ожидаемыми.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: approximate.ts

```
import { type Path, type Matrix } from "../common";
type Edge = [number, number, number]; // [cost, u, v]

export class ApproximateTSP {
  private matrix: Matrix;
  private n: number;
  private start: number;

  constructor(matrix: Matrix, startCity: number) {
    this.matrix = matrix.map(row => [...row]);
    this.n = matrix.length;
    this.start = startCity;
  }

  private findMST(): [number, number][] {
    const mst: [number, number][] = [];
    const visited = new Set<number>();
    const heap: Edge[] = [];

    visited.add(this.start);

    for (let nextCity = 0; nextCity < this.n; nextCity++) {
      if (nextCity !== this.start &&
this.matrix[this.start][nextCity] !== Infinity) {
        this.heapPush(heap, [this.matrix[this.start][nextCity],
this.start, nextCity]);
      }
    }

    while (heap.length > 0 && visited.size < this.n) {
      const [cost, u, v] = this.heapPop(heap);

      if (!visited.has(v)) {
        visited.add(v);
        mst.push([u, v]);

        for (let nextCity = 0; nextCity < this.n; nextCity++) {
```

```

        if (!visited.has(nextCity) && this.matrix[v][nextCity]
!= Infinity) {
            this.heapPush(heap, [this.matrix[v][nextCity], v,
nextCity]);
        }
    }
}

return mst;
}

private dfsTraversal(mst: [number, number][]): number[] {
    const adjacencyList: number[][] = Array(this.n).fill(0).map(() =>
[]);

    for (const [u, v] of mst) {
        adjacencyList[u].push(v);
        adjacencyList[v].push(u);
    }

    const path: number[] = [];
    const stack: number[] = [this.start];
    const visited = new Set<number>();

    while (stack.length > 0) {
        const node = stack.pop()!;

        if (!visited.has(node)) {
            visited.add(node);
            path.push(node);

            for (let i = adjacencyList[node].length - 1; i >= 0; i--)
{
                const neighbor = adjacencyList[node][i];
                if (!visited.has(neighbor)) {
                    stack.push(neighbor);
                }
            }
        }
    }
}

```

```

        return path;
    }

    solve(): [Path, number] {
        const mst = this.findMST();

        const path = this.dfsTraversal(mst);

        const fullPath = [...path, this.start];

        let cost = 0;
        for (let i = 0; i < fullPath.length - 1; i++) {
            cost += this.matrix[fullPath[i]][fullPath[i + 1]];
        }

        return [fullPath.map(x => x + 1), cost];
    }

    private heapPush(heap: Edge[], edge: Edge): void {
        heap.push(edge);
        heap.sort((a, b) => a[0] - b[0]);
    }

    private heapPop(heap: Edge[]): Edge {
        return heap.shift()!;
    }
}

```

Название файла: little.ts

```

import { Node, type Matrix, type Path } from "../common"

export class LittleTSP {
    private originalMatrix: Matrix;
    private n: number;
    private start: number;
    private bestCost: number;
    private bestPath: Path;

    constructor(matrix: Matrix, startCity: number) {
        this.originalMatrix = matrix.map(row => [...row]);
    }
}

```

```

    this.n = matrix.length;
    this.start = startCity;
    this.bestCost = Infinity;
    this.bestPath = [];
  }

  private reduceMatrix(matrix: Matrix): [Matrix, number] {
    const newMatrix = matrix.map(row => [...row]);
    let reductionCost = 0;

    for (let i = 0; i < newMatrix.length; i++) {
      const minVal = Math.min(...newMatrix[i].filter(x => x !==
Infinity));
      if (minVal !== Infinity && minVal > 0) {
        reductionCost += minVal;
        newMatrix[i] = newMatrix[i].map(x => x === Infinity ?
Infinity : x - minVal);
      }
    }

    for (let j = 0; j < newMatrix[0].length; j++) {
      const col = newMatrix.map(row => row[j]);
      const minVal = Math.min(...col.filter(x => x !== Infinity));
      if (minVal !== Infinity && minVal > 0) {
        reductionCost += minVal;
        for (let i = 0; i < newMatrix.length; i++) {
          if (newMatrix[i][j] !== Infinity) {
            newMatrix[i][j] -= minVal;
          }
        }
      }
    }

    return [newMatrix, reductionCost];
  }

  private deepCopyMatrix(matrix: Matrix): Matrix {
    return matrix.map(row => [...row]);
  }

  solve(): [Path, number] {

```

```

    const [reducedMatrix, reductionCost] =
this.reduceMatrix(this.deepCopyMatrix(this.originalMatrix));

    const root = new Node(
        [this.start],
        reducedMatrix,
        reductionCost
    );

    const heap: Node[] = [root];
    this.heapify(heap);

    while (heap.length > 0) {
        const currentNode = this.heapPop(heap);

        if (currentNode.path.length === this.n) {
            const finalCost = currentNode.cost +

currentNode.matrix[currentNode.path[currentNode.path.length -
1]][this.start];

            if (finalCost < this.bestCost) {
                this.bestCost = finalCost;
                this.bestPath = [...currentNode.path, this.start];
            }
            continue;
        }

        if (currentNode.cost >= this.bestCost) {
            continue;
        }

        const lastCity = currentNode.path[currentNode.path.length -
1];

        for (let nextCity = 0; nextCity < this.n; nextCity++) {
            if (currentNode.matrix[lastCity][nextCity] !== Infinity) {
                const newMatrix =
this.deepCopyMatrix(currentNode.matrix);

                for (let i = 0; i < this.n; i++) {

```

```

        newMatrix[i][nextCity] = Infinity;
    }
    newMatrix[lastCity] = new
Array(this.n).fill(Infinity);

    const [reducedNewMatrix, reduction] =
this.reduceMatrix(newMatrix);
    const newCost = currentNode.cost +
currentNode.matrix[lastCity][nextCity] + reduction;

    const newPath = [...currentNode.path, nextCity];

    const child = new Node(newPath, reducedNewMatrix,
newCost);

    this.heapPush(heap, child);
    }
    }
    }

    const resultPath = this.bestPath.map(x => x + 1);
    return [resultPath, this.bestCost];
}

private heapify(heap: Node[]): void {
    heap.sort((a, b) => a.cost - b.cost);
}

private heapPush(heap: Node[], node: Node): void {
    heap.push(node);
    heap.sort((a, b) => a.cost - b.cost);
}

private heapPop(heap: Node[]): Node {
    return heap.shift()!;
}
}

```

Название файла: common.ts

```

export type Matrix = number[][];
export type Path = number[];

```

```
export class Node {  
  constructor(  
    public path: Path,  
    public matrix: Matrix,  
    public cost: number,  
  ) {}  
}
```