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 _____ Малиновский А.А.

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 Литла - 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 inf. На каждом шаге запоминается его стоимость. При нахождении первого решения, делаем его минимально возможным, чтобы в дальнейшем отсекать ветви которые

заведомо больше минимального. В случае нахождения еще меньшего решения, уже оно становиться 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)

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 -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 MatrixHandler предназначен для работы с матрицами , включает методы для их обработки и анализа, используется алгоритмом Литтла.

Методы:

1. **__init__(self, matrix: list[list])**

   Инициализирует объект класса, сохраняя переданную матрицу в атрибуте self.matrix.

2. **__len__(self)**

   Возвращает количество строк в матрице.

3. **__getitem__(self, index: int)**

Возвращает строку матрицы по указанному индексу. Если индекс выходит за пределы, вызывает исключение IndexError.

4. **print_matrix(self)**

   Выводит матрицу построчно на экран.

5. **min_except(self, lst: list, idx: int)**

Возвращает минимальный элемент списка, исключая элемент с указанным индексом.

6. **reduct(self)**

Выполняет редукцию матрицы: вычитает минимальный элемент из каждой строки и каждого столбца. Возвращает общую стоимость редукции. Если встречается бесконечность (math.inf), возвращает -1.

7. **find_heavy_zero(self)**

Находит "тяжелый ноль" — нулевой элемент с максимальной суммой минимальных элементов в его строке и столбце. Возвращает координаты этого элемента.

8. **find_longest_path(self, solution: dict, edge: tuple)**

Находит самый длинный путь в графе на основе текущего решения и ребра. Возвращает путь в виде списка.

9. **forbid_cycles(self, path: list, i_index: list, j_index: list)**

Запрещает ребро, которое замыкает цикл в графе, устанавливая соответствующее значение в матрице на math.inf.

10. **delete_row_column(self, i: int, j: int, solution: dict, i_index: list, j_index: list)**

    Удаляет строку и столбец из матрицы по указанным индексам, обновляет решение и запрещает циклы, если они возникают.

Класс LittleAlgorithm

Реализует алгоритм Литтла.

1. **__init__(self, matrix: list[list]) —** Инициализирует объект класса, сохраняя матрицу и настраивая начальные значения.

2. answer(self, start: int) -> list[int] — Возвращает путь, начиная с указанной вершины, на основе лучшего решения.

3. collect_tree_data(self, current_cost: int, parent_node: str, step_cost: float, i_index: list[int], j_index: list[int], branching_arc: tuple[int, int] = None) -> str — Собирает данные о текущем состоянии матрицы для визуализации дерева решений.

4. handle2x2(self, tmp_solution: dict, current_cost: float, i_index: list[int], j_index: list[int]) -> None — Обрабатывает матрицу 2x2, обновляя лучшее решение.

5. method_Little(self, matrix: list[list], tmp_solution: dict, cur_cost: float, i_index: list[int], j_index: list[int], parent_node: str = None,

branching_arc: tuple[int, int] = None) -> None — Реализует алгоритм Литтла, рекурсивно выполняя ветвление и ограничение.

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

__init__(self, matrix: list[list[int]]) -> None

Инициализирует объект класса, сохраняя исходную матрицу и создавая пустой список для остовного дерева.

prim(self, start: int) -> None

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

dfs(self, matr: list[list[int]], start: int, res: list[int]) -> None

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

def find_res(self, start: int) -> list[int]

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

Также созданы классы Menu для возможности выбора алгоритма, MatrixCreator для создания, сохранения и загрузки матрицы, Visualiser для отображения пути коммивояжёра и построения дерева решений для алгоритма Литтла.

Код программы смотреть в приложении А.

Тестирование.

Тестируем Алгоритм Литтла, потому что он даёт оптимальное решение.

.

Ввод	Вывод	Ожидаемый результат
[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]	Лучший путь: [1, 5, 3, 4, 2] Минимальная стоимость: 64	Результат верный
[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]	Лучший путь: [1, 6, 4, 2, 7, 3, 5] Минимальная стоимость: 82	Результат верный
[inf, 10, 13] [12, inf, 15] [10, 11, inf]	Лучший путь: [1, 2, 3] Минимальная стоимость: 35	Результат верный

Также были можно увидеть дерево решений для алгоритма Литтла и также итоговый путь коммивояжёра

Пример для матрицы

```
[inf, 15, 11, 17, 19]
[18, inf, 13, 11, 16]
[20, 16, inf, 16, 15]
[11, 11, 16, inf, 18]
[17, 20, 18, 12, inf]
```

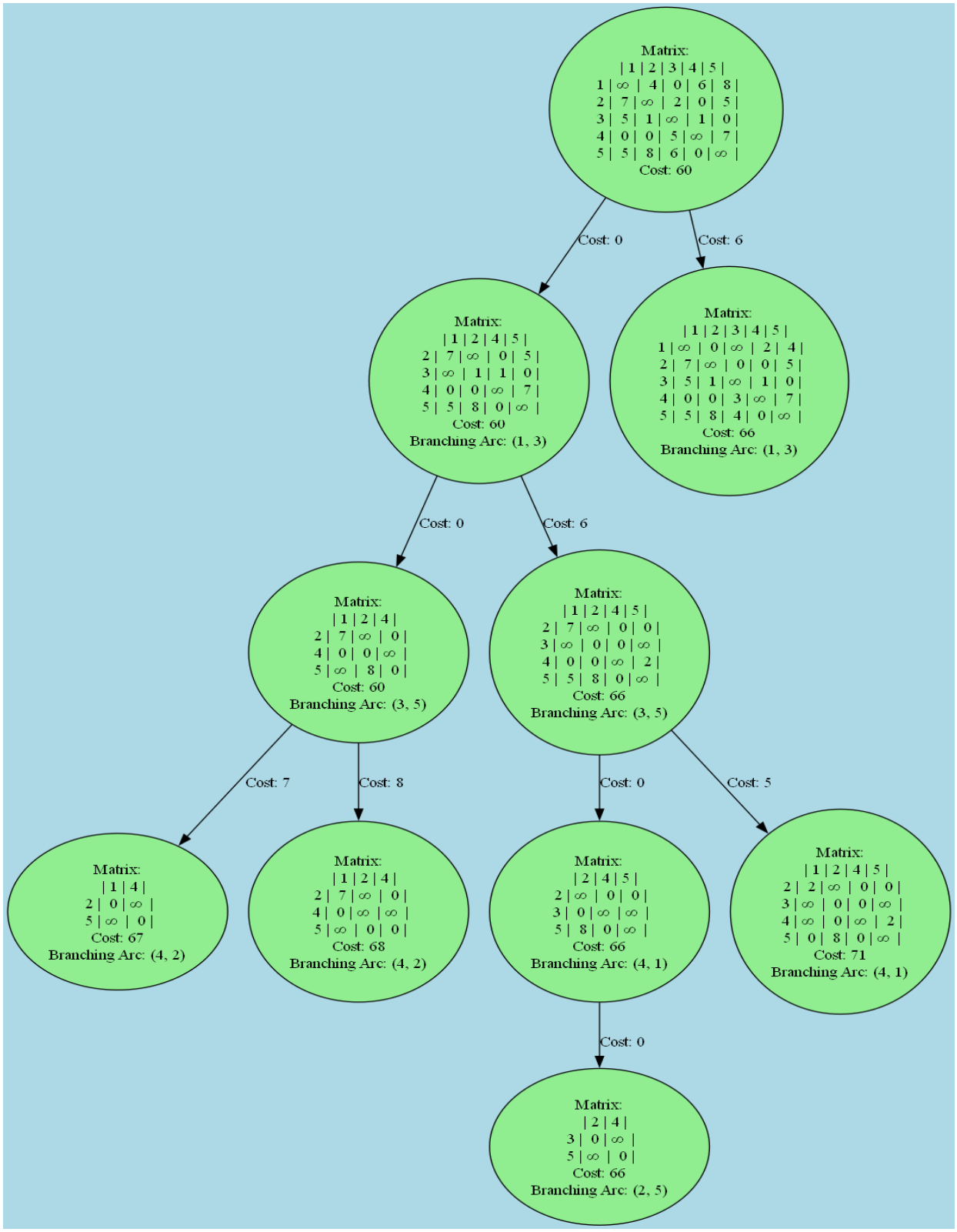

Рисунок 1 – Дерево принятия решения для алгоритма Литтла

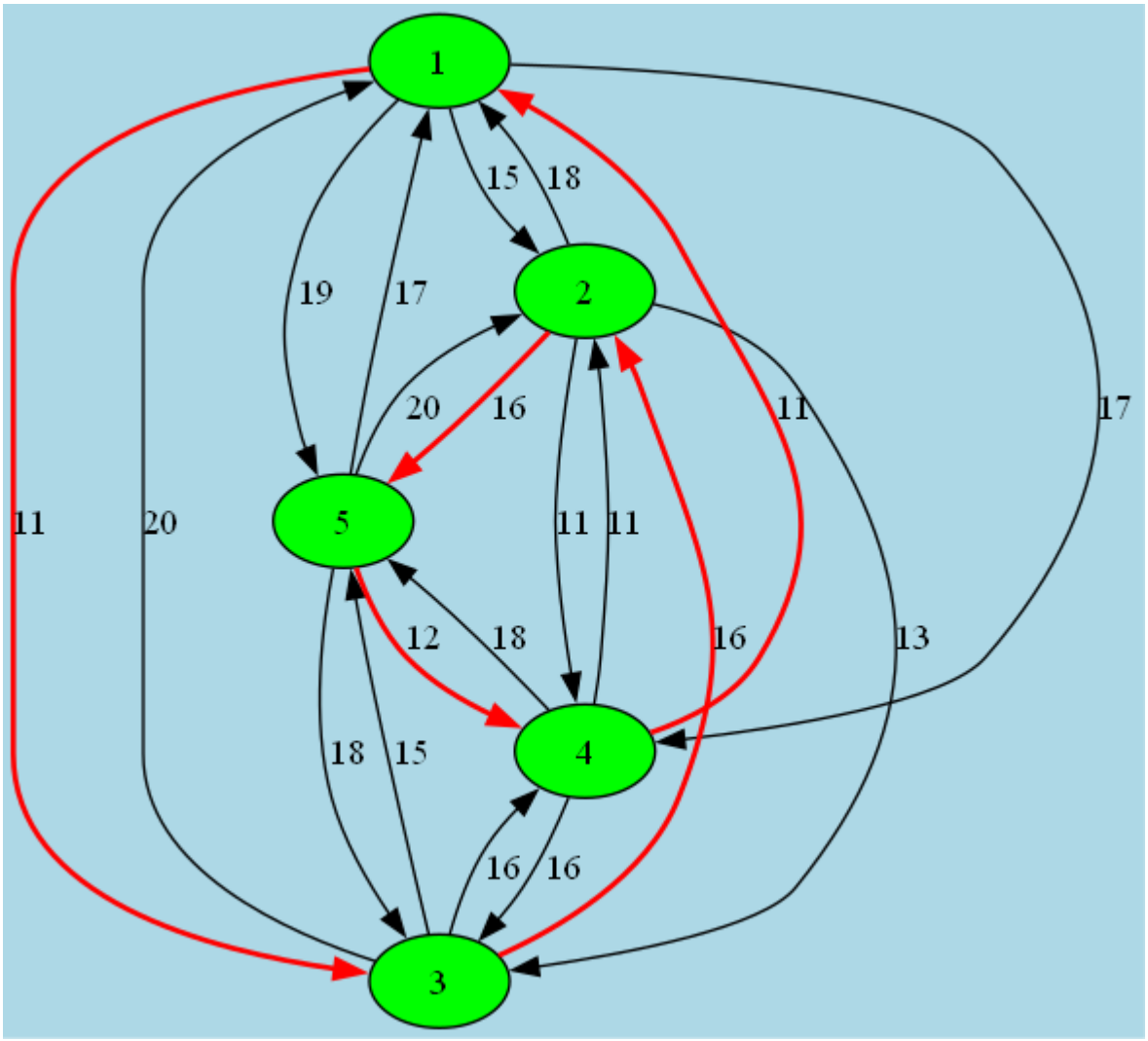

Рисунок 2 – Путь коммивояжёра

Исследование.

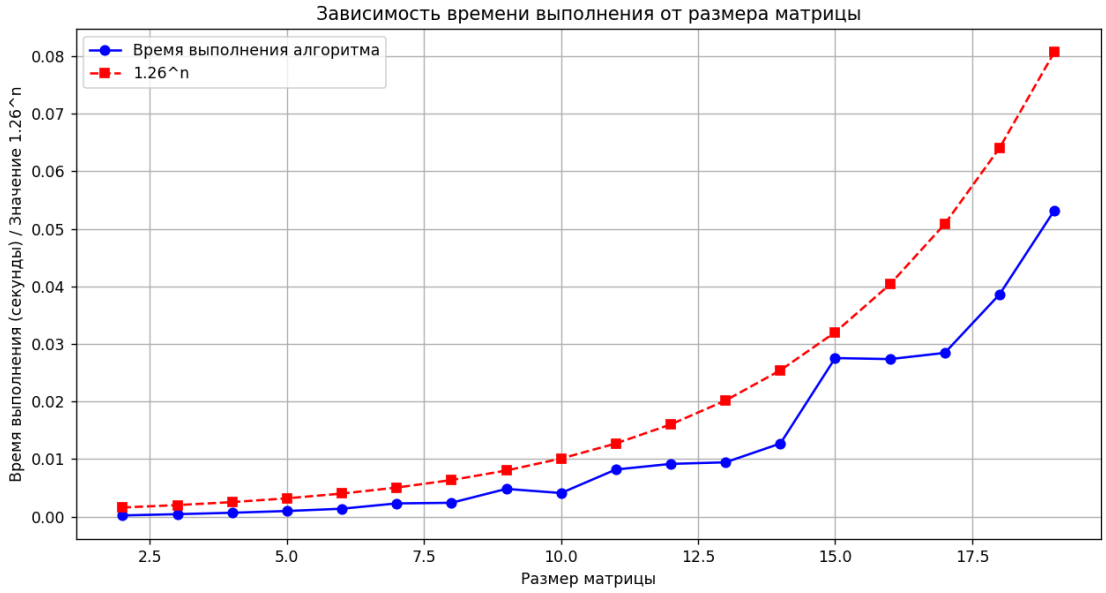

Рисунок 3 - График зависимости размера случайной матрицы от времени выполнения алгоритма Литтла

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 остовного 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: ADO_MOD_algorithm.py

```
# import copy
import math
from MatrixCreator import MatrixCreator
from MatrixHandler import MatrixHandler
from Visualiser import Visualiser

class ADO_MOD_algorithm:
    def __init__(self, matrix: list[list[int]]) -> None:
        self.src_matrix=copy.deepcopy(matrix)
        self.matrix=matrix
        self.ost=[]

    def prim(self, start: int) -> None:
        # будущее остовное дерево
        self.ost = [[0 for i in range(len(self.matrix))] for i in
  range(len(self.matrix))]
        visited = [start]           #вершины, уже включенные в дерево
        weight = []
        while(len(visited) != len(self.matrix)):
            for row in self.ost:
                print(row)
            print("Включенные вершины:", *[x+1 for x in visited])
            min_w = math.inf
            i, j = 0, 0
            # проходим по всем вершинам остовного дерева
            for elem in visited:
                if min_w > min(self.matrix[elem]):
                    print(self.matrix[elem])
                    # находим самое легкое ребро
                    min_w = min(self.matrix[elem])
                    i = elem
                    j = self.matrix[elem].index(min_w)
                    print("Найдена новая вершина для добавления:", j+1, " вес
  ребра = ", min_w)
            if j not in visited:
                print("Добавляем вершину в МОД")
                weight.append(min_w)
                visited.append(j)
                self.ost[i][j] = min_w
                self.ost[j][i] = min_w
            self.matrix[i][j] = math.inf
            print(f'Вычеркнем ребро {i+1} {j+1} из графа')

    #поиск в глубину
    def dfs(self, matr: list[list[int]], start: int, res: list[int]) -> None:
        for i, elem in enumerate(matr[start]):
            if i not in res:
                res.append(i)
                print("Текущий путь:", [x+1 for x in res])
                self.dfs(matr, i, res)
            if len(res) == len(matr):
                return

    def find_res(self, start: int) -> list[int]:
        start=start-1
```

```
            self.prim( start)
            way = [start]
            print("Запущен поиск в глубину")
            self.dfs(self.ost, start, way)
            i, j = 0, 1
            cost = 0
            while j < len(way):
                cost += self.src_matrix[way[i]][way[j]]
                i+=1
                j+=1
            cost += self.src_matrix[way[i]][way[0]]
            print("Полученный приближенный путь коммивояжера:", ' - '.join(str(x)
      for x in [i + 1 for i in way]))
            print("Его стоимость:", cost)
            return [x+1 for x in way ]

    if __name__ == '__main__':
        matrix = [
            [math.inf, 27, 43, 16, 30, 26],
            [7, math.inf, 16, 1, 30, 25],
            [20, 13, math.inf, 35, 5, 0],
            [21, 16, 25, math.inf, 18, 18],
            [12, 46, 27, 48, math.inf, 5],
            [23, 5, 5, 9, 5, math.inf]
        ]
        matrix=[[math.inf, 7, 2],
    [3, math.inf, 6],
    [7, 4, math.inf],]
        b=copy.deepcopy(matrix)
        #matrix=MatrixCreator().generate_euclidean_matrix(size=10,dimensions=10)
        #MatrixCreator().save_matrix(matrix,"matrix.txt")
        #matrix=MatrixCreator().load_matrix("matrix.txt")
        start =2
        solution=ADO_MOD_algorithm(matrix).find_res(start)
        Visualiser().visualise_graph(b,solution)#endif
```

Название файла: LittleAlgorithm.py

```
import copy
  import math
  import time
  from MatrixCreator import MatrixCreator
  from MatrixHandler import MatrixHandler
  from Visualiser import Visualiser


  class LittleAlgorithm:
      def __init__(self, matrix:list[list[int]])->None:
          self.matrix_handler = MatrixHandler(matrix)
          self.best_solution = {}
          self.min_cost = math.inf
          self.tree_data = []

      def answer(self, start: int) -> list[int]:
          next_node = self.best_solution[start]
          res = [start]
          while next_node != start:
              res.append(next_node)
              next_node = self.best_solution[next_node]
          return res
```

```
    def collect_tree_data(self, current_cost: int, parent_node: str,
step_cost: float, i_index: list[int],
                          j_index: list[int], branching_arc: tuple[int, int]
= None) -> str:
        j_index_shifted = [j + 1 for j in j_index]
        header = "   | " + " | ".join(map(str, j_index_shifted)) + " |"
        matrix_rows = []
        for i, row in enumerate(self.matrix_handler.matrix):
            row_str = f"{i_index[i]+1:2} | " + " | ".join(
                f"{'∞' if x == math.inf else x:2}" for x in row
            ) + " |"
            matrix_rows.append(row_str)

        # Объединяем все строки
        matrix_str = "\n".join([header] + matrix_rows)
        node_label = f"Matrix:\n{matrix_str}\nCost: {current_cost}"
        if branching_arc:
            node_label += f"\nBranching Arc: {branching_arc}"
        node_id = str(len(self.tree_data))
        self.tree_data.append(("node", node_id, node_label))
        if parent_node is not None:
            self.tree_data.append(("edge", parent_node, node_id, f"Cost:
{step_cost}"))

        return node_id

    def handle2x2(self, tmp_solution: dict, current_cost: int, i_index:
list[int], j_index: list[int]) -> None:
        for i in range(len(self.matrix_handler)):
            for j in range(len(self.matrix_handler)):
                if self.matrix_handler[i][j] == math.inf:
                    # Вершина (i + 1) % 2 связывается с вершиной j (другая
вершина в столбце).
                    tmp_solution[i_index[(i + 1) % 2] + 1] = j_index[j] + 1
                    # Вершина i связывается с вершиной (j + 1) % 2 (другая
вершина в строке).
                    tmp_solution[i_index[i] + 1] = j_index[(j + 1) % 2] + 1
                    self.best_solution = tmp_solution
                    self.min_cost = current_cost

    def method_Little(self, matrix: list[list], tmp_solution: dict, cur_cost:
int, i_index: list[int],
                      j_index: list[int], parent_node: str = None,
branching_arc: tuple[int, int] = None) -> None:
        self.matrix_handler.matrix = matrix
        print("Матрица на текущем шаге")
        self.matrix_handler.print_matrix()
        step_cost = self.matrix_handler.reduct()
        print("Редуцированная матрица")
        self.matrix_handler.print_matrix()

        if step_cost == -1:
            return

        current_cost = cur_cost + step_cost
        node_id = self.collect_tree_data(current_cost, parent_node,
step_cost,i_index,j_index, branching_arc)

        if current_cost >= self.min_cost:
            print("Текущая стоимость пути уже не будет выгоднее чем рекорд,
конец рекурсии", self.min_cost)
```

```
            return

        if len(self.matrix_handler) == 2:
            print("Размерность матрицы = 2, конец этой ветки рекурсии")
            self.handle2x2(tmp_solution,current_cost,i_index,j_index)
            print("Найденное решение:", self.best_solution)
            return

        i, j = self.matrix_handler.find_heavy_zero()
        if i is None or j is None:
            return

        print("Начата левая ветвь:")
        new_solution = tmp_solution.copy()
        left_matrix = copy.deepcopy(matrix)
        left_i_index = i_index[:]
        left_j_index = j_index[:]
        self.matrix_handler.matrix = left_matrix
        self.matrix_handler.delete_row_column(i, j, new_solution,
left_i_index, left_j_index)
        self.method_Little(left_matrix, new_solution, current_cost,
left_i_index, left_j_index, node_id,
                           branching_arc=(i_index[i] + 1, j_index[j] + 1))

        print("Начата правая ветвь:")
        matrix[i][j] = math.inf
        self.method_Little(matrix, tmp_solution, current_cost, i_index[:],
j_index[:], node_id,
                           branching_arc=(i_index[i] + 1, j_index[j] + 1))

if __name__ == '__main__':
    start = 1
    a = [
        [math.inf, 25, 40, 31, 27],
        [5, math.inf, 17, 30, 25],
        [19, 15, math.inf, 6, 1],
        [9, 50, 24, math.inf, 6],
        [22, 8, 7, 10, math.inf]
    ]
    #a=MatrixCreator.load_matrix(a,'matrix.txt')
    a=MatrixCreator().generate_matrix(1,10,10,20,)
    b = copy.deepcopy(a)
    little_algo = LittleAlgorithm(a)
    i_index = [i for i in range(len(a))]
    j_index = [j for j in range(len(a))]
    little_algo.method_Little(a, {}, 0, i_index, j_index)
    print("Лучший путь:", little_algo.answer(start))
    print("Минимальная стоимость:", little_algo.min_cost)
    Visualiser().visualise_solution_tree(little_algo.tree_data)
    Visualiser().visualise_graph(b,little_algo.answer(start))
```

Название файла: MatrixHandler.py

```
import math

class MatrixHandler:

    def __init__(self, matrix:list[list])->None:

        self.matrix = matrix
```

```
def __len__(self)->int:
    return len(self.matrix)

def __getitem__(self, index:int)->None:
    if 0 <= index < len(self.matrix):
        return self.matrix[index]
    else:
        raise IndexError("Index out of range")

def print_matrix(self)->None:
    for row in self.matrix:
        print(row)

def min_except(self, lst:int, idx:int)->list:
    return min([x for i, x in enumerate(lst) if i != idx])

def reduct(self)->int:
    d = 0
    # Редукция строк
    for i, row in enumerate(self.matrix):
        min_row = min(row)
        if min_row == math.inf:
            return -1
        # вычитаем из всех элементов строки минимальный
        self.matrix[i] = [elem - min_row for elem in row]
        # добавляем к стоимости d
        d += min_row

    # Редукция столбцов
    for i in range(len(self.matrix)):
        min_column = min([row[i] for row in self.matrix])
        if min_column == math.inf:
            return -1
        for row in self.matrix:
            # вычитаем из всех элементов столбца минимальный
            row[i] -= min_column
        # добавляем к стоимости d
        d += min_column
    return d

def find_heavy_zero(self)->int:
```

```
        d_max = 0
        res = None
        for i in range(len(self.matrix)):
            for j in range(len(self.matrix)):
                if self.matrix[i][j] == 0:
                    # находим сумму минимальных элементов
                    tmp = self.min_except(self.matrix[i], j) +
self.min_except([row[j] for row in self.matrix], i)
                    # если найденная сумма больше рекорда, перезаписываем
рекорд и координаты тяжелого нуля
                    if tmp > d_max or not res:
                        d_max = tmp
                        res = (i, j)
        return res

    def find_longest_path(self, solution: dict, edge: tuple)->list:
        start, end = edge
        path = []

        # Ищем путь в одну сторону (от start к end)
        current = start
        while current in solution.keys():
            path.append(solution[current])
            current = solution[current]

        # Ищем путь в другую сторону (от end к start)
        current = start
        path.insert(0, current)
        while current in solution.values():
            for key in solution.keys():
                if (solution[key]==current):
                    current=key
            path.insert(0, current)

        return path

    def forbid_cycles(self, path: list, i_index: list, j_index: list)-
>None:
        if len(path) < 2:
            return
```

```
            # Запрещаем ребро, которое замыкает цикл
            restore_i = path[-1]
            restore_j = path[0]
            print(f"Текущий путь содержащий удаляемое ребро: {path}")
            print(f"Запрещаем ребро[{restore_i}, {restore_j}]")
            if restore_i-1 in i_index and restore_j-1 in j_index:
                i_for_inf = i_index.index(restore_i-1)
                j_for_inf = j_index.index(restore_j-1)
                self.matrix[i_for_inf][j_for_inf] = math.inf  # запрещаем
        движение по обратному ребру



          def delete_row_column(self, i:int, j:int, solution:dict, i_index:list,
        j_index:list) -> None:
            restore_i = i_index[i]  # находим, каким вершинам графа
        соответствуют эти индексы
            restore_j = j_index[j]
            solution[restore_i + 1] = restore_j + 1  # обновляем решение

        path=self.find_longest_path(solution,(restore_i+1,solution[restore_i +
        1]))
            self.forbid_cycles(path,i_index,j_index)

            # Удаляем строку и столбец
            i_index.pop(i)
            j_index.pop(j)
            self.matrix.pop(i)  # удаляем строку
            for row in self.matrix:
                row.pop(j)  # удаляем столбец
```

Название файла: Visualiser.py

```
from graphviz import Digraph
  import math

  class Visualiser:
      def visualise_graph(self,adj_matrix, highlight_nodes=None):
          if highlight_nodes is None:
              highlight_nodes = []

          highlight_edges = []
          for i in range(len(highlight_nodes) - 1):
              highlight_edges.append((highlight_nodes[i], highlight_nodes[i +
  1]))
          highlight_edges.append((highlight_nodes[-1],highlight_nodes[0]))

          dot = Digraph()
```

```
            dot.attr(size="15,15",bgcolor="lightblue")
            dot.attr()

            # Добавление вершин
            for i in range(len(adj_matrix)):
                dot.node(str(i + 1), str(i + 1),style="filled",fillcolor="green")

            # Добавление рёбер
            for i in range(len(adj_matrix)):
                for j in range(len(adj_matrix[i])):
                    if adj_matrix[i][j] != math.inf:
                        if (i + 1, j + 1) in highlight_edges:
                            dot.edge(str(i + 1), str(j + 1),
  label=str(adj_matrix[i][j]), color="red", penwidth="2.0")
                        else:
                            dot.edge(str(i + 1), str(j + 1),
  label=str(adj_matrix[i][j]))

            dot.render('images/salesman_way', format='png', cleanup=True)

        def visualise_solution_tree(self,tree_data):
            graph = Digraph()
            graph.attr(size="50,50", bgcolor="lightblue")
            for item in tree_data:
                if item[0] == "node":
                    _, node_id, label = item
                    graph.node(node_id,
  label,style="filled",fillcolor="lightgreen")
                elif item[0] == "edge":
                    _, from_node, to_node, label = item
                    graph.edge(from_node, to_node, label)
            graph.render('images/Little_tree', format='png', cleanup=True)
```

Название файла: MatrixCreator.py

```
import math
import random

class MatrixCreator:
    def load_matrix(self,filename:str):
        f = open(filename)
        matrix = []
        for line in f:
            tmp = line.split(' ')
            res = []
            for x in tmp:
                try:
                    res.append(int(x))
                except ValueError:
                    res.append(math.inf)
            matrix.append(res)
        f.close()
        return matrix

    def generate_matrix(self,sym:bool, size:int, min_val:int, max_val:int):
        matrix = [[0 for j in range(size)] for i in range(size)]
        if sym:
            for i in range(size):
                for j in range(i, size):
                    a = random.randint(min_val, max_val)
```

```
                    matrix[i][j] = a
                    matrix[j][i] = a
                    if i == j:
                        matrix[i][j] = math.inf
        else:
            for i in range(size):
                for j in range(size):
                    a = random.randint(min_val, max_val)
                    matrix[i][j] = a
                    if i == j:
                        matrix[i][j] = math.inf
        return matrix

    def generate_euclidean_matrix(self, size: int, dimensions: int = 2, min_val:
  int = 0, max_val: int = 10):
        # Генерация случайных точек в n-мерном пространстве
        points = [[random.randint(min_val, max_val) for _ in range(dimensions)]
  for _ in range(size)]

        # Создание матрицы расстояний
        matrix = [[0 for _ in range(size)] for _ in range(size)]

        for i in range(size):
            for j in range(size):
                if i == j:
                    matrix[i][j] = math.inf
                else:
                    # Вычисление евклидова расстояния между точками i и j
                    distance = math.sqrt(sum((points[i][k] - points[j][k]) ** 2
  for k in range(dimensions)))
                    matrix[i][j] = int(distance)
                    matrix[j][i] = int(distance) # Матрица симметрична

        return matrix

    def save_matrix(self,matrix:list[list], filename:str):
        f = open(filename, 'w')
        for elem in matrix:
            string = ' '.join([str(x) for x in elem])
            f.write(string+'\n')
        f.close()

if __name__ == '__main__':
    #matrix=MatrixCreator().generate_matrix(False,5,2,11)
    #MatrixCreator().save_matrix(matrix,"matr.txt")
    matrix=MatrixCreator().load_matrix("matr.txt")
    print(matrix)
```

Название файла: main.py

```
import copy
from MatrixCreator import MatrixCreator
from LittleAlgorithm import LittleAlgorithm
from ADO_MOD_algorithm import ADO_MOD_algorithm
from Visualiser import Visualiser


class Menu:
```

```
  def __init__(self):
      self.matrix_creator = MatrixCreator()
      self.src_matrix=None
      self.matrix = None

  def display_menu(self):
      print("1. Создать матрицу")
      print("2. Загрузить матрицу")
      print("3. Сохранить матрицу")
      print("4. Выбрать алгоритм (ADO_MOD или Литтла)")
      print("5. Выход")

  def create_matrix(self):
      sym = input("Симметричная матрица? (y/n): ").lower() == 'y'
      size = int(input("Введите размер матрицы: "))
      min_val = int(input("Введите минимальное значение: "))
      max_val = int(input("Введите максимальное значение: "))
      self.src_matrix = self.matrix_creator.generate_matrix(sym, size,
min_val, max_val)
      print("Матрица создана:")
      self.print_matrix()

  def load_matrix(self):
      filename = input("Введите имя файла для загрузки: ")
      self.src_matrix = self.matrix_creator.load_matrix(filename)
      print("Матрица загружена:")
      self.print_matrix()

  def save_matrix(self):
      if  self.src_matrix is None:
          print("Матрица не создана или не загружена.")
          return
      filename = input("Введите имя файла для сохранения: ")
      self.matrix_creator.save_matrix( self.src_matrix, filename)
      print(f"Матрица сохранена в файл {filename}.")

  def choose_algorithm(self):
      if self.src_matrix is None:
          print("Матрица не создана или не загружена.")
          return
      print("1. Алгоритм ADO_MOD")
```

```
        print("2. Алгоритм Литтла")
        self.matrix=copy.deepcopy(self.src_matrix)
        choice = input("Выберите алгоритм: ")
        if choice == '1':
            start = int(input("Введите начальную вершину: "))
            solution = ADO_MOD_algorithm(self.matrix).find_res(start)
            Visualiser().visualise_graph(self.src_matrix, solution)
        elif choice == '2':
            start = int(input("Введите начальную вершину: "))
            little_algo = LittleAlgorithm(self.matrix)
            i_index = [i for i in range(len(self.matrix))]
            j_index = [j for j in range(len(self.matrix))]
            little_algo.method_Little(self.matrix, {}, 0, i_index,
j_index)
            print("Лучший путь:", little_algo.answer(start))
            print("Минимальная стоимость:", little_algo.min_cost)
            Visualiser().visualise_solution_tree(little_algo.tree_data)
            Visualiser().visualise_graph(self.src_matrix,
little_algo.answer(start))
        else:
            print("Неверный выбор.")

    def print_matrix(self):
        for row in self.src_matrix:
            print(row)

    def run(self):
        end=False
        while not end:
            self.display_menu()
            choice = input("Выберите действие: ")
            if choice == '1':
                self.create_matrix()
            elif choice == '2':
                self.load_matrix()
            elif choice == '3':
                self.save_matrix()
            elif choice == '4':
                self.choose_algorithm()
            elif choice == '5':
                print("Выход из программы.")
```

```
                    end=True
                else:
                    print("Неверный выбор. Пожалуйста, выберите снова.")


    if __name__ == '__main__':
        menu=Menu()
        menu.run()
```

Название файла: benchmarking.py

```
import time
import matplotlib.pyplot as plt
from MatrixCreator import MatrixCreator
from ADO_MOD_algorithm import ADO_MOD_algorithm
from LittleAlgorithm import LittleAlgorithm

import matplotlib
matplotlib.use('TkAgg')

def test_algorithm(matrix_sizes, num_tests_per_size):
    results = {}

    for size in matrix_sizes:
        times = []
        for _ in range(num_tests_per_size):
            matrix = MatrixCreator().generate_matrix(0, size, 10, 20)
            little_algo = LittleAlgorithm(matrix)
            i_index = [i for i in range(size)]
            j_index = [j for j in range(size)]

            start_time = time.time()
            #ADO_MOD_algorithm(matrix).find_res(1)
            #little_algo.method_Little(matrix, {}, 0, i_index, j_index)
            end_time = time.time()

            times.append(end_time - start_time)

        avg_time = sum(times) / num_tests_per_size
        results[size] = avg_time
        print(f"Size: {size}, Average Time: {avg_time:.4f} seconds")

    return results

def plot_results(results):
    sizes = list(results.keys())
    times = list(results.values())


    plt.figure(figsize=(12, 6))

    # График времени выполнения алгоритма
    plt.plot(sizes, times, marker='o', linestyle='-', color='b', label='Время
  выполнения алгоритма')


    plt.title('Зависимость времени выполнения от размера матрицы')
    plt.xlabel('Размер матрицы')
```

```
    plt.ylabel('Время выполнения (секунды)')
    plt.grid(True)
    plt.legend()
    plt.show()

if __name__ == '__main__':
    matrix_sizes = range(2, 20)  # Размеры матриц от 2 до 19
    num_tests_per_size = 50  # Количество тестов для каждого размера матрицы

    results = test_algorithm(matrix_sizes, num_tests_per_size)
    plot_results(results)
```