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 _____ Атоян М.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)

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 MatrixReader предназначен для парсинга матрицы из текствого файла.

Методы:

1. **__init__(self, file_path)**

   Инициализирует объект класса. Сохраняет путь к файлу с матрицей.

2. **read_matrix(self)**

   Читает матрицу из файла, парсит её в поле matrix.

3. **print_matrix(self)**

   Форматированный вывод матрицы.

Класс LittleAlgorithm

Реализует алгоритм Литтла.

Методы:

1. **__init__(self, 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(self)** — 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. **reduce_matrix(self, 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 **Approximate**: Реализует алгоритм для нахождения приближённого решения задачи коммивояжёра через построение минимального отставного дерева и поиск в глубину.

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

find_mst(self)

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

dfs_traversal(self, mst)

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

solve(self]

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
[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

Рисунок 2 – Путь коммивояжёра

Исследование.

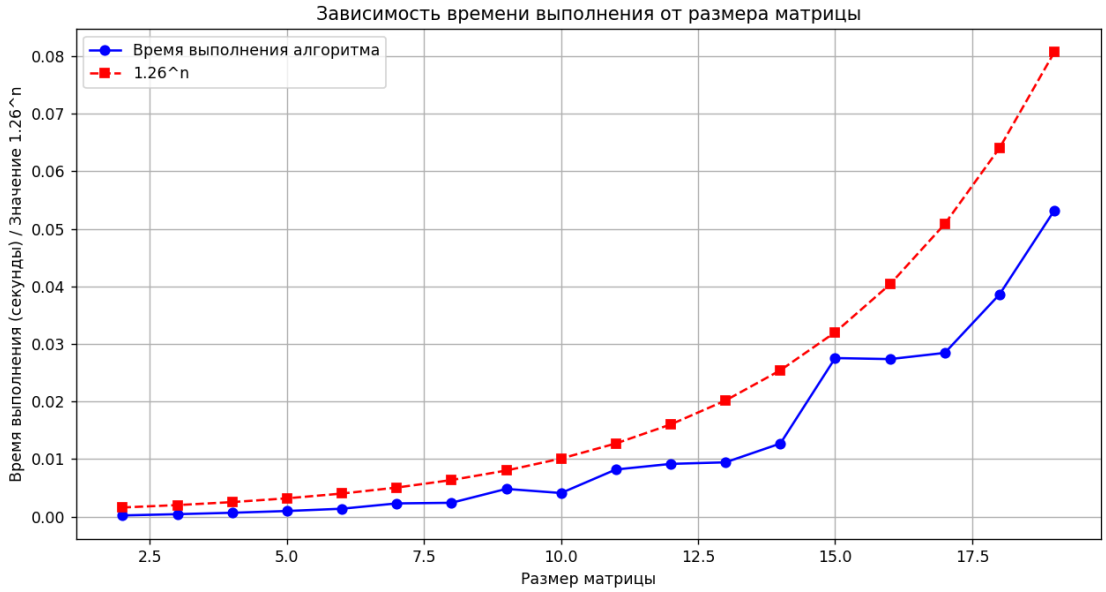

Рисунок 3 - График зависимости размера случайной матрицы от времени выполнения алгоритма Литтла

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: mst_dfs_algorithm.py

```
from math import inf
import heapq

class ApproximateTSP:
    def __init__(self, matrix, start_city):
        self.matrix = matrix
        self.n = len(matrix)
        self.start = start_city

    def find_mst(self):
        mst = []  # Список рёбер
        visited = set()
        heap = []  # Куча для выбора рёбер

        visited.add(self.start)
        for next_city in range(self.n):
            if next_city != self.start and self.matrix[self.start][next_city] !=
  inf:
                heapq.heappush(heap, (self.matrix[self.start][next_city],
  self.start, next_city))

        while heap:
            cost, u, v = heapq.heappop(heap)
            if v not in visited:
                visited.add(v)
                mst.append((u, v))
                for next_city in range(self.n):
                    if next_city not in visited and self.matrix[v][next_city] !=
  inf:
                        heapq.heappush(heap, (self.matrix[v][next_city], v,
  next_city))
        return mst

    def dfs_traversal(self, mst):
        adjacency_list = {i: [] for i in range(self.n)}
        for u, v in mst:
            adjacency_list[u].append(v)
            adjacency_list[v].append(u)

        path = []
        stack = [self.start]
        visited = set()

        while stack:
            node = stack.pop()
            if node not in visited:
                visited.add(node)
                path.append(node)
                for neighbor in reversed(adjacency_list[node]):
                    if neighbor not in visited:
                        stack.append(neighbor)
        return path

    def solve(self):
        mst = self.find_mst()
```

```
        path = self.dfs_traversal(mst)

        path.append(self.start)

        cost = 0
        for i in range(len(path) - 1):
            cost += self.matrix[path[i]][path[i + 1]]

        return path, cost
```

Название файла: little_algorithm.py

```
from math import inf
import heapq
import copy

class LittleTSP:
    def __init__(self, matrix, start_city):
        self.original_matrix = matrix
        self.n = len(matrix)
        self.start = start_city
        self.best_cost = inf
        self.best_path = []

    class Node:
        def __init__(self, path, matrix, cost):
            self.path = path          # Текущий путь
            self.matrix = matrix      # Текущая редуцированная матрица
            self.cost = cost          # Нижняя граница стоимости

        def __lt__(self, other):
            return self.cost < other.cost

    def reduce_matrix(self, matrix):
        reduction_cost = 0

        for i in range(len(matrix)):
            min_val = min(matrix[i])
            if min_val != inf and min_val > 0:
                reduction_cost += min_val
                matrix[i] = [x - min_val for x in matrix[i]]

        for j in range(len(matrix[0])):
            col = [matrix[i][j] for i in range(len(matrix))]
            min_val = min(col)
            if min_val != inf and min_val > 0:
                reduction_cost += min_val
                for i in range(len(matrix)):
                    matrix[i][j] -= min_val

        return matrix, reduction_cost

    def solve(self):
        initial_matrix = copy.deepcopy(self.original_matrix)
        reduced_matrix, reduction_cost = self.reduce_matrix(initial_matrix)

        root = self.Node(
            path=[self.start],
            matrix=reduced_matrix,
            cost=reduction_cost
```

```
        )

        heap = []
        heapq.heappush(heap, root)

        while heap:
            current_node = heapq.heappop(heap)

            if len(current_node.path) == self.n:
                final_cost = current_node.cost +
  current_node.matrix[current_node.path[-1]][self.start]
                if final_cost < self.best_cost:
                    self.best_cost = final_cost
                    self.best_path = current_node.path + [self.start]
                continue

            if current_node.cost >= self.best_cost:
                continue

            last_city = current_node.path[-1]
            for next_city in range(self.n):
                if current_node.matrix[last_city][next_city] != inf:
                    # Создаём новую матрицу для дочернего узла
                    new_matrix = copy.deepcopy(current_node.matrix)

                    # Удаляем строку last_city и столбец next_city
                    for i in range(self.n):
                        new_matrix[i][next_city] = inf
                    new_matrix[last_city] = [inf] * self.n

                    # Редуцируем новую матрицу
                    reduced_new_matrix, reduction =
  self.reduce_matrix(new_matrix)
                    new_cost = current_node.cost +
  current_node.matrix[last_city][next_city] + reduction

                    # Создаём новый узел
                    new_path = current_node.path.copy()
                    new_path.append(next_city)

                    child = self.Node(
                        path=new_path,
                        matrix=reduced_new_matrix,
                        cost=new_cost
                    )

                    heapq.heappush(heap, child)

        return list(x + 1 for x in self.best_path), self.best_cost
```

Название файла: matrix_reader.py

```
from math import inf

class MatrixReader:
    def __init__(self, file_path):
        self.file_path = file_path
        self.matrix = []

    def read_matrix(self):
        try:
```

```
          with open(self.file_path, 'r') as file:
              for line in file:
                  row = list(int(x) if x != "inf" else inf for x in
line.strip().split())
                  self.matrix.append(row)
          return self.matrix
      except FileNotFoundError:
          print(f"Файл {self.file_path} не найден.")
          return None
      except ValueError:
          print("Ошибка: файл содержит некорректные данные (ожидались
числа).")
          return None

  def print_matrix(self):
      if not self.matrix:
          print("Матрица пуста.")
          return
      for row in self.matrix:
          print(" ".join(map(str, row)))
```