**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 |  | Гребнев Е.Д. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2025

**Цель работы.**

Изучить принцип работы алгоритмов нахождения пути коммивояжера на графах.

**Задание.**

**Вариант 1**

Mетод 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:**

1. Метод ветвей и границ (алгоритм Литтла)
   * В худшем случае его сложность составляет O(n!), так как метод перебирает возможные маршруты, применяя ограничения для их отсечения.
   * Однако редукция матрицы (O(n²)) и оценка MST (O(n² log n)) помогают сократить количество ветвей, уменьшая среднее время выполнения.
2. Метод ближайшего соседа
   * Имеет сложность O(n²), так как в каждом из **n** шагов производится поиск ближайшего непосещённого города (O(n)).
   * Этот метод быстрый, но не гарантирует оптимального решения.
3. Генерация и редукция матрицы
   * Генерация матрицы выполняется за O(n²), так как каждый элемент заполняется случайным значением.
   * Редукция матрицы (вычитание минимумов строк и столбцов) требует O(n²) времени.
4. Минимальное остовное дерево (MST)
   * Используется в оценке нижней границы стоимости маршрута.
   * Применение алгоритма Прима даёт сложность O(n² log n).

В итоге:

* Метод Литтла (ветвей и границ) – экспоненциальная сложность O(n!) в худшем случае, но улучшается благодаря отсечениям.
* Ближайший сосед – O(n²), но без гарантии оптимального решения.
* Дополнительные вычисления (редукция, MST) – O(n² log n)

**Описание реализованных методов**

* ***generate\_matrix***Генерирует квадратную матрицу размера size x size со случайными значениями, где элементы на главной диагонали заменены на бесконечность (INF). Эти значения обозначают, что один город не может быть связан с самим собой.
  + **Аргументы:**
    - size: Размер матрицы.
    - seed: Начальное значение для генератора случайных чисел.
  + **Возвращает**: Генерированную матрицу в виде numpy.ndarray.
* ***print\_matrix***  
  Выводит матрицу расстояний в удобочитаемом виде с использованием цветного форматирования, чтобы представить информацию о расстояниях между городами.
  + **Аргументы**:
    - matrix: Матрица расстояний (numpy.ndarray).
  + **Возвращает**: Выводит матрицу в консоль в виде таблицы.
* ***export\_matrix***  
  Экспортирует матрицу в файл заданного типа. Поддерживаемые типы: txt, csv, json, npy.
  + **Аргументы:**
    - matrix: Матрица для экспорта.
    - filename: Имя файла для сохранения.
    - file\_type: Тип файла (по умолчанию txt).
  + **Возвращает**: Экспортирует матрицу в файл и выводит сообщение о успешной операции.
* ***solve\_tsp***  
  Основная функция, которая решает задачу коммивояжера с использованием одного из двух методов: алгоритма Литтла (little) или алгоритма ближайшего соседа (nearest).
  + **Аргументы:**
    - matrix: Матрица расстояний.
    - method: Метод решения задачи (little или nearest).
    - verbose: Флаг для вывода промежуточных результатов.
  + **Возвращает**: Лучшее решение задачи с путём и его стоимостью.
* ***tsp\_little\_algorithm***  
  Решает задачу коммивояжера с использованием алгоритма Литтла. Алгоритм использует метод ветвей и границ для поиска оптимального решения.
  + **Аргументы:**
    - matrix: Матрица расстояний.
    - verbose: Флаг для вывода промежуточных результатов.
  + **Возвращает**: Лучшее решение задачи.
* ***tsp\_nearest\_neighbor***  
  Решает задачу коммивояжера с использованием алгоритма ближайшего соседа. Алгоритм на каждом шаге выбирает ближайший не посещённый город и добавляет его в путь.
  + **Аргументы:**
    - matrix: Матрица расстояний.
    - verbose: Флаг для вывода промежуточных результатов.
  + **Возвращает**: Найденный путь и его стоимость.
* ***tsp\_branch\_and\_bound***  
  Реализует метод ветвей и границ для решения задачи коммивояжера. Эта рекурсивная функция выполняет поиск по возможным путям с использованием редукции стоимости и минимального остовного дерева (MST).
  + **Аргументы:**
    - matrix: Матрица расстояний.
    - current: Текущий город.
    - visited: Множество посещённых городов.
    - current\_cost: Текущая стоимость пути.
    - path: Текущий путь.
    - best: Лучшее решение (путь и стоимость).
    - selected\_edges: Выбранные рёбра.
    - verbose: Флаг для вывода промежуточных результатов.
* ***reduce\_cost\_matrix***  
  Функция редукции матрицы затрат, которая вычитает минимальные значения строк и столбцов. Это уменьшает размеры задачи и помогает ускорить решение.
  + **Аргументы:**
    - matrix: Матрица расстояний.
    - verbose: Флаг для вывода промежуточных результатов.
  + **Возвращает**: Редуцированную матрицу и общую стоимость редукции.
* ***minimum\_spanning\_tree***  
  Вычисляет минимальное остовное дерево (MST) для подмножества вершин. Это используется для оценки нижней границы стоимости пути в методе ветвей и границ.
  + **Аргументы:**
    - matrix: Матрица расстояний.
    - vertices: Множество вершин.
    - verbose: Флаг для вывода промежуточных результатов.
  + **Возвращает**: Стоимость минимального остовного дерева для заданных вершин.
* ***city\_index\_to\_name***  
  Конвертирует индекс города в его буквенное обозначение (например, 0 → 'A', 1 → 'B' и так далее).
  + **Аргументы:**
    - index: Индекс города.
  + **Возвращает**: Буквенное обозначение города.

**Описание алгоритма АБС.**

**Алгоритм АБС (Аппроксимация Ближайшего Соседа)** — это приближённый алгоритм решения задачи коммивояжера (TSP). Он основан на жадном подходе, который позволяет найти решение задачи с ограниченной точностью за приемлемое время.

**Основная идея:**

Алгоритм решает задачу, начиная с произвольного города и поочередно выбирает следующий город, который является ближайшим к текущему, пока не будут посещены все города.

**Шаги алгоритма АБС:**

1. **Выбор начального города**: Начинаем с произвольного города.
2. **Поиск ближайшего города**: На каждом шаге выбирается ближайший ещё не посещённый город. Расстояния между городами могут быть представлены в виде матрицы расстояний, где каждый элемент матрицы D(i,j)— это расстояние между городами i и j.
3. **Перемещение по маршруту**: После выбора ближайшего города, текущий город помечается как посещённый, и алгоритм продолжает искать ближайший город для следующего шага.
4. **Повторение до завершения**: Этот процесс продолжается до тех пор, пока все города не будут посещены. Когда все города будут посещены, алгоритм возвращается в начальный город, чтобы завершить цикл.

**Описание:**

* **Время работы**: Время работы алгоритма обычно составляет O(n^2), где n — количество городов. Это объясняется тем, что для каждого города необходимо найти ближайший ещё не посещённый город, что требует обхода всех оставшихся городов.
* **Точность**: Алгоритм не гарантирует нахождение оптимального решения. Однако, согласно теории, для задачи коммивояжера приближённый алгоритм АБС даёт решение, стоимость которого не превышает стоимости оптимального пути более чем в два раза (в худшем случае).

**Преимущества:**

1. **Простота реализации**: Алгоритм очень прост в реализации и не требует сложных вычислений или данных.
2. **Быстродействие**: Алгоритм работает достаточно быстро, что делает его подходящим для решения задач с большим числом городов.

**Недостатки:**

1. **Неоптимальность**: Алгоритм не всегда находит оптимальное решение, и в некоторых случаях может существенно отклоняться от оптимума.
2. **Чувствительность к начальной точке**: Результат может зависеть от того, с какого города начинается путь.

Код программы смотреть в приложении А.

**Тестирование.**

Тестируем и сравниваем Алгоритм Литтла И АБС

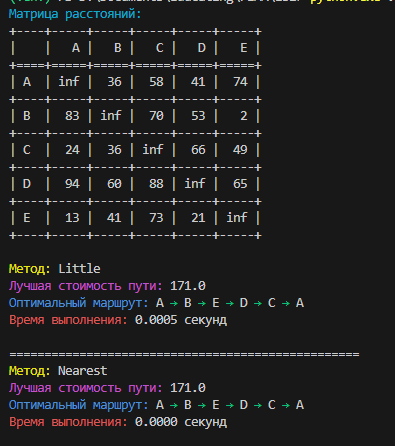


Рисунок 1. Тестирование на матрице 5x5

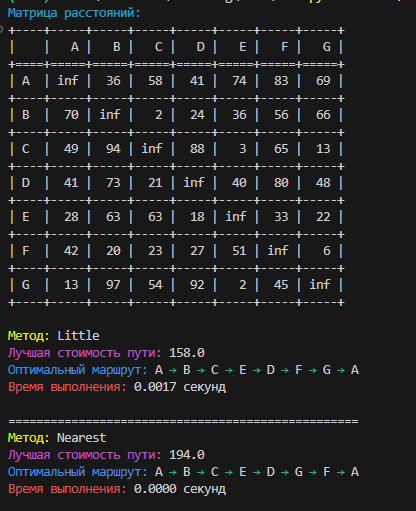


Рисунок 2. Тестирование на матрице 7x7

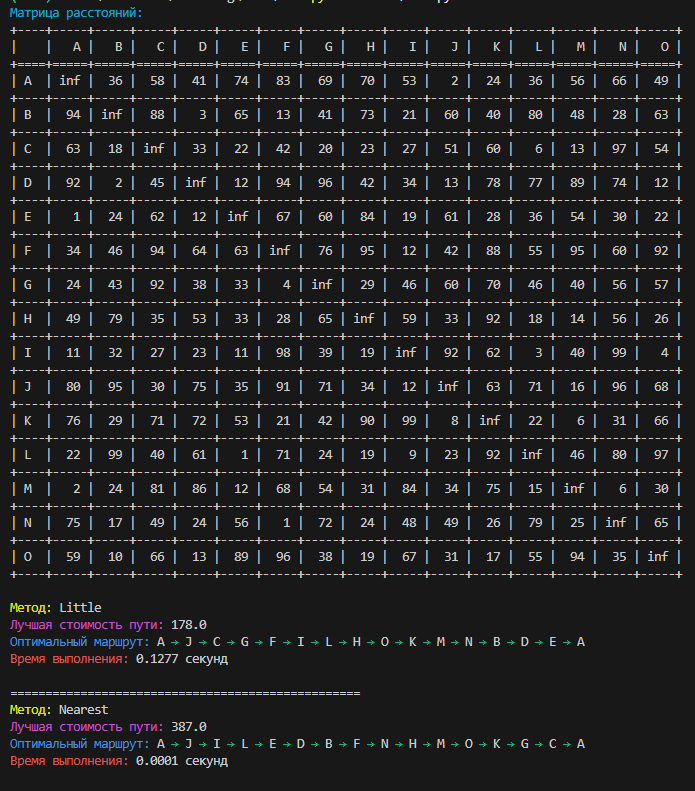


Рисунок 3. Тестирование на матрице 15x15

**Исследование.**

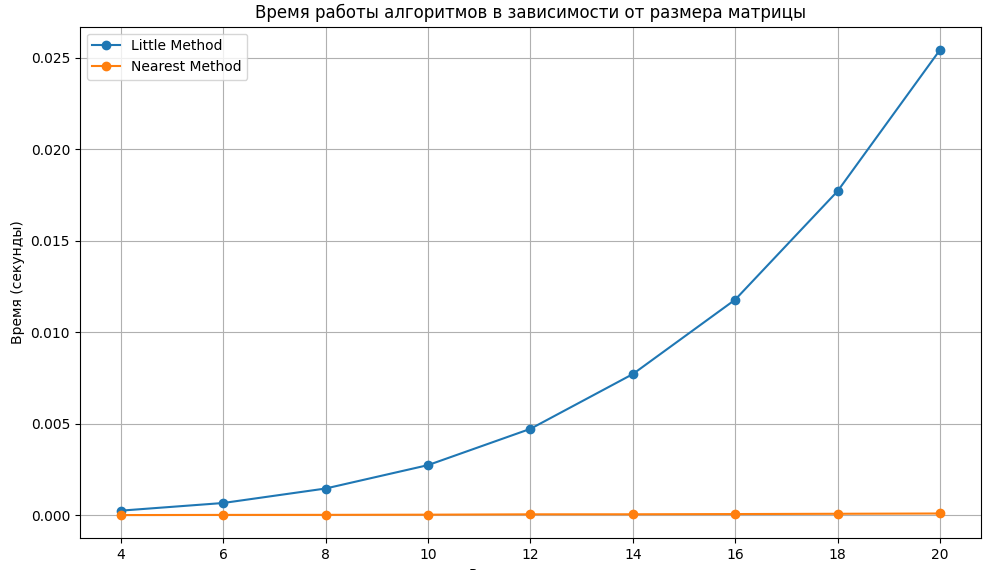


Рисунок 4 - График зависимости размера случайной матрицы от времени выполнения алгоритмов (диапазон значений матрицы 1-2)

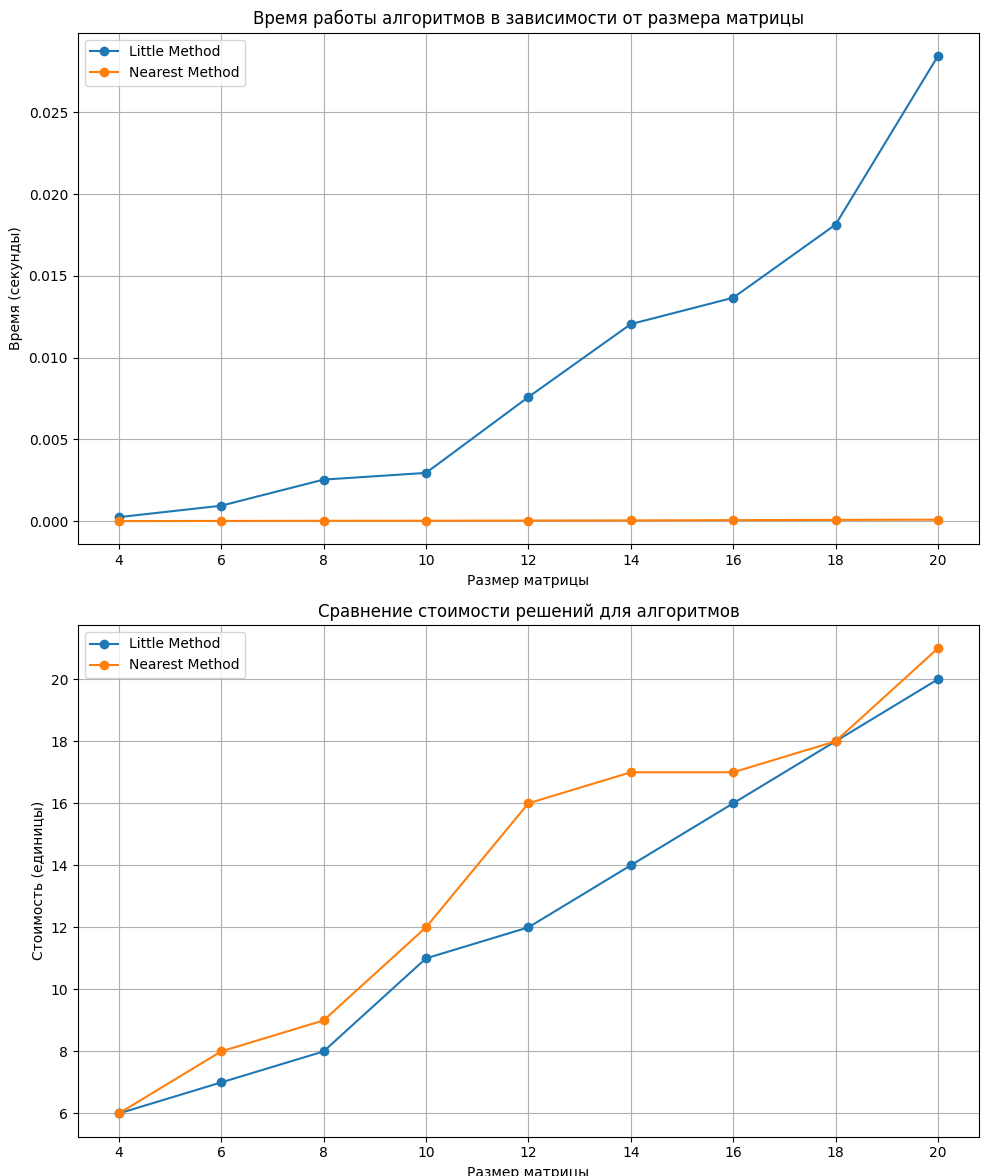


Рисунок 4 - График зависимости размера случайной матрицы от времени выполнения алгоритмов (диапазон значений матрицы 1-3)

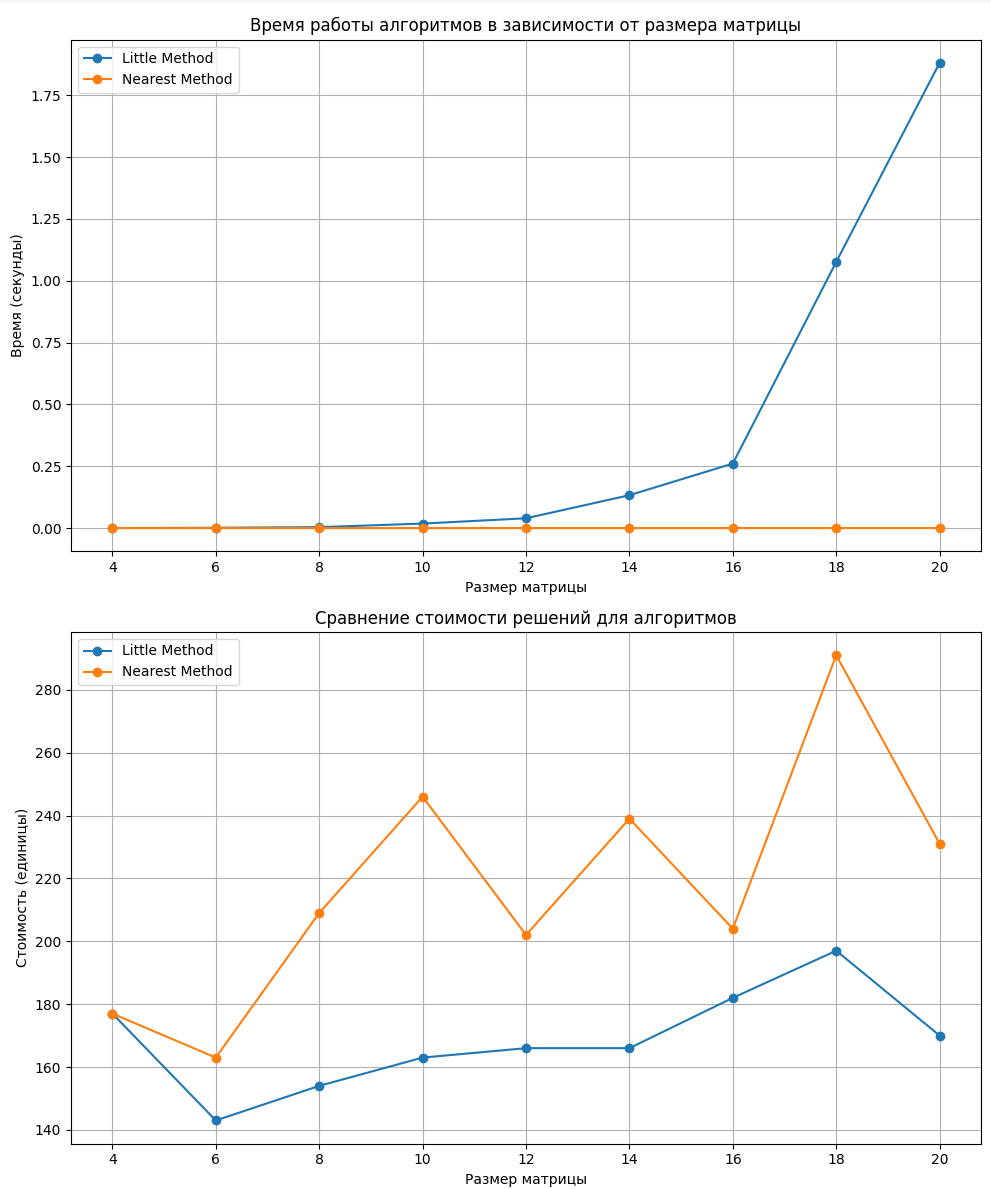


Рисунок 5 - График зависимости размера случайной матрицы от времени выполнения алгоритмов (диапазон значений матрицы 1-100)

**Выводы по тестированию:**

Мы наблюдаем закономерное снижение скорости выполнения алгоритмов с увеличением размера матрицы, что соответствует теоретическим оценкам. Кроме того, на производительность влияет распределение значений в матрице: большие числа увеличивают вычислительные затраты. Приближённый алгоритм демонстрирует стабильное и низкое время выполнения, что делает его эффективным для быстрого поиска решений.

При сравнении эффективности алгоритмов следует отметить, что алгоритм Литтла показывает лучшие результаты при значительных различиях между элементами матрицы (стоимостями путей). В случаях с небольшим разбросом значений приближённый алгоритм может давать схожие решения за более короткое время. Однако при хаотичном распределении чисел алгоритм Литтла значительно превосходит приближённый по качеству решения, несмотря на возросшие затраты по времени.

**Выводы.**

В ходе работы были реализованы и протестированы два алгоритма решения задачи коммивояжера: **алгоритм Литтла** и **метод ближайшего соседа**.

Выбор алгоритма зависит от требований к точности решения и допустимого времени выполнения:

* Если необходимо **найти абсолютно оптимальный маршрут**, следует использовать **алгоритм Литтла**, но учитывать его высокую сложность.
* Если важнее **быстрое получение приемлемого решения**, целесообразно применять **метод ближайшего соседа**.

**ПРИЛОЖЕНИЕ А**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

Название файла: main.py

import time

from tsp\_algorithms import solve\_tsp

from utils import generate\_matrix, print\_matrix, print\_solution, export\_matrix

if \_\_name\_\_ == '\_\_main\_\_':

    matrix = generate\_matrix(size=20, seed=52)

    # print\_matrix(matrix)

    # export\_matrix(matrix, file\_type='txt')

    verbose = False

    for method in ['little', 'nearest']:

        start\_time = time.time()

        best\_solution = solve\_tsp(matrix, method, verbose=verbose)

        end\_time = time.time()

        print\_solution(method, best\_solution, end\_time - start\_time)

Название файла: tsp\_algorithms.py

import heapq

import numpy as np

from math import inf

def reduce\_cost\_matrix(matrix, verbose=False):

    """Редуцирует матрицу затрат, вычитая минимальные значения строк и столбцов.

    Аргументы:

    matrix -- матрица затрат (numpy.ndarray)

    verbose -- флаг для вывода промежуточных результатов (bool, по умолчанию False)

    Возвращает:

    reduced\_matrix -- редуцированная матрица (numpy.ndarray)

    total\_reduction -- сумма всех вычтенных минимумов (float)

    """

    reduced\_matrix = matrix.copy()

    row\_min = np.min(reduced\_matrix, axis=1)

    row\_min[np.isinf(row\_min)] = 0

    reduced\_matrix -= row\_min[:, None]

    col\_min = np.min(reduced\_matrix, axis=0)

    col\_min[np.isinf(col\_min)] = 0

    reduced\_matrix -= col\_min

    if verbose:

        print("=== Редукция матрицы ===")

        print(f"Минимумы строк: {row\_min}")

        print(f"Минимумы столбцов: {col\_min}")

        print(f"Редуцированная матрица:\n{reduced\_matrix}")

    return reduced\_matrix, np.sum(row\_min) + np.sum(col\_min)

def minimum\_spanning\_tree(matrix, vertices, verbose=False):

    """Вычисляет минимальное остовное дерево (MST) для заданного множества вершин.

    Аргументы:

    matrix -- матрица затрат (numpy.ndarray)

    vertices -- множество вершин (set)

    verbose -- флаг для вывода промежуточных результатов (bool, по умолчанию False)

    Возвращает:

    total\_cost -- стоимость минимального остовного дерева (float)

    """

    if len(vertices) <= 1:

        return 0.0

    total\_cost = 0.0

    visited = set()

    start = next(iter(vertices))

    priority\_queue = [(0.0, start)]

    if verbose:

        print(f"=== Вычисление MST для вершин {vertices} ===")

    while priority\_queue:

        weight, u = heapq.heappop(priority\_queue)

        if u in visited:

            continue

        total\_cost += weight

        visited.add(u)

        for v in vertices - visited:

            edge\_weight = matrix[u][v]

            if edge\_weight != inf:

                heapq.heappush(priority\_queue, (edge\_weight, v))

                if verbose:

                    print(f"Добавлено ребро {u} -> {v} с весом {edge\_weight}")

    if verbose:

        print(f"MST оценка оставшихся вершин: {total\_cost}")

    return total\_cost if len(visited) == len(vertices) else inf

def tsp\_branch\_and\_bound(matrix, current, visited, current\_cost, path, best, selected\_edges, verbose=False):

    """

    Рекурсивно решает задачу коммивояжера методом ветвей и границ.

    matrix -- матрица затрат (numpy.ndarray)

    current -- текущий город (int)

    visited -- множество посещённых городов (set)

    current\_cost -- накопленная стоимость пути (float)

    path -- текущий путь (list)

    best -- словарь с лучшей найденной стоимостью и путём: {'cost': float, 'path': list}

    selected\_edges -- выбранные ребра для предотвращения циклов (dict)

    verbose -- флаг для вывода промежуточных результатов (bool)

    """

    num\_cities = len(matrix)

    # Если все города посещены, пытаемся вернуться в начальный город

    if len(visited) == num\_cities:

        return\_cost = matrix[current][0]

        if return\_cost == float('inf'):

            return

        total\_cost = current\_cost + return\_cost

        if total\_cost < best['cost']:

            best['cost'] = total\_cost

            best['path'] = path + [0]

        if verbose:

            print(f"✅ Найден полный путь {path + [0]} с общей стоимостью {total\_cost}")

        return

    # Формируем список кандидатов с сортировкой по стоимости перехода

    candidates = sorted(

        [city for city in range(num\_cities)

         if city not in visited and matrix[current][city] != float('inf')],

        key=lambda city: matrix[current][city]

    )

    for next\_city in candidates:

        cost\_to\_next = matrix[current][next\_city]

        # Создаем новую матрицу и модифицируем её для текущего перехода

        new\_matrix = matrix.copy()

        new\_matrix[current, :] = float('inf')

        new\_matrix[:, next\_city] = float('inf')

        if len(visited) + 1 < num\_cities:

            new\_matrix[next\_city][0] = float('inf')

        if current in selected\_edges:

            prev\_city = selected\_edges[current]

            new\_matrix[next\_city][prev\_city] = float('inf')

        # Копируем словарь выбранных ребер и обновляем его для текущего перехода

        new\_selected\_edges = selected\_edges.copy()

        new\_selected\_edges[current] = next\_city

        # Выполняем редукцию матрицы, возвращается новая матрица и стоимость редукции

        reduced\_matrix, reduced\_cost = reduce\_cost\_matrix(new\_matrix, verbose)

        new\_cost = current\_cost + cost\_to\_next + reduced\_cost

        # Оцениваем нижнюю границу, используя MST и сумму двух минимальных ребер

        remaining\_cities = set(range(num\_cities)) - visited - {next\_city}

        mst\_estimate = minimum\_spanning\_tree(reduced\_matrix, remaining\_cities, verbose) if remaining\_cities else 0

        # Находим сумму двух минимальных ребер для оставшихся городов

        min\_edges\_sum = sum(sorted([min(row[row != inf]) for i, row in enumerate(reduced\_matrix) if i in remaining\_cities])[:2])

        lower\_bound = new\_cost + min(mst\_estimate, min\_edges\_sum)

        if verbose:

            print(f"🔍 Рассматриваем путь {path + [next\_city]} (стоимость: {new\_cost}, нижняя граница: {lower\_bound})")

        # Продолжаем рекурсию только если нижняя граница ниже текущего лучшего результата

        if lower\_bound < best['cost']:

            tsp\_branch\_and\_bound(

                reduced\_matrix, next\_city, visited | {next\_city}, new\_cost,

                path + [next\_city], best, new\_selected\_edges, verbose

            )

def tsp\_little\_algorithm(matrix, verbose=False):

    """

    Решает задачу коммивояжера с использованием алгоритма Литтла.

    matrix -- матрица затрат (numpy.ndarray)

    verbose -- флаг для вывода промежуточных результатов (bool)

    Возвращает:

    best\_solution -- словарь с лучшим найденным путём и его стоимостью: {'cost': float, 'path': list}

    """

    if verbose:

        print("🚀 Запуск алгоритма Литтла...")

    best\_solution = {'cost': float('inf'), 'path': []}

    reduced\_matrix, initial\_cost = reduce\_cost\_matrix(matrix, verbose)

    tsp\_branch\_and\_bound(reduced\_matrix, 0, {0}, initial\_cost, [0], best\_solution, {}, verbose)

    return best\_solution

def tsp\_nearest\_neighbor(matrix, verbose=False):

    """Решает задачу коммивояжера с использованием алгоритма ближайшего соседа.

    Аргументы:

    matrix -- матрица затрат (numpy.ndarray)

    verbose -- флаг для вывода промежуточных результатов (bool, по умолчанию False)

    Возвращает:

    solution -- найденный путь и его стоимость (dict)

    """

    num\_cities = len(matrix)

    visited = {0}

    path = [0]

    total\_cost = 0

    current\_city = 0

    while len(visited) < num\_cities:

        next\_city = min((i for i in range(num\_cities) if i not in visited), key=lambda i: matrix[current\_city][i], default=None)

        if next\_city is None or matrix[current\_city][next\_city] == inf:

            return {'cost': inf, 'path': []}

        visited.add(next\_city)

        path.append(next\_city)

        total\_cost += matrix[current\_city][next\_city]

        current\_city = next\_city

    if matrix[current\_city][0] == inf:

        return {'cost': inf, 'path': []}

    path.append(0)

    total\_cost += matrix[current\_city][0]

    if verbose:

        print(f"🏁 Оптимальный путь найден: {path}, стоимость: {total\_cost}")

    return {'cost': total\_cost, 'path': path}

def solve\_tsp(matrix, method='little', verbose=False):

    """Решает задачу коммивояжера выбранным методом.

    Аргументы:

    matrix -- матрица затрат (numpy.ndarray)

    method -- метод решения ('little' или 'nearest\_neighbor') (str, по умолчанию 'little')

    verbose -- флаг для вывода промежуточных результатов (bool, по умолчанию False)

    Возвращает:

    solution -- найденный путь и его стоимость (dict)

    """

    return tsp\_little\_algorithm(matrix, verbose) if method == 'little' else tsp\_nearest\_neighbor(matrix, verbose)

Название файла: benchmark.py

import time

import numpy as np

import matplotlib.pyplot as plt

from tsp\_algorithms import solve\_tsp

from utils import generate\_matrix

from concurrent.futures import ProcessPoolExecutor

def run\_tsp\_method(matrix, method, runs):

    """Запуск метода решения задачи коммивояжера для одного метода и одной матрицы."""

    total\_cost, total\_time = 0, 0

    start\_time = time.time()

    for \_ in range(runs):

        best\_solution = solve\_tsp(matrix, method)

        total\_cost += best\_solution['cost']

    total\_time = time.time() - start\_time

    avg\_cost = total\_cost / runs

    avg\_time = total\_time / runs

    return avg\_cost, avg\_time

def benchmark\_tsp(matrix\_sizes, runs=10):

    """Запускает тестирование методов решения задачи коммивояжера для матриц разного размера и строит таблицу результатов."""

    start\_time\_bench = time.time()

    methods = ['little', 'nearest']

    results = []

    little\_times = []

    nearest\_times = []

    little\_costs = []

    nearest\_costs = []

    # Предварительная генерация всех матриц

    matrices = {size: generate\_matrix(size, seed=52) for size in matrix\_sizes}

    with ProcessPoolExecutor() as executor:

        for size in matrix\_sizes:

            matrix = matrices[size]

            row = [size]

            # Запуск методов в многозадачном режиме

            futures = {method: executor.submit(run\_tsp\_method, matrix, method, runs) for method in methods}

            avg\_costs = {}

            avg\_times = {}

            for method in methods:

                avg\_cost, avg\_time = futures[method].result()

                avg\_costs[method] = avg\_cost

                avg\_times[method] = avg\_time

                row.extend([avg\_cost, avg\_time])

                if method == 'little':

                    little\_times.append(avg\_time)

                    little\_costs.append(avg\_cost)

                elif method == 'nearest':

                    nearest\_times.append(avg\_time)

                    nearest\_costs.append(avg\_cost)

            # Расчет отклонения

            deviation = ((avg\_costs['nearest'] - avg\_costs['little']) / avg\_costs['little']) \* 100 if avg\_costs['little'] != 0 else float('inf')

            row.append(deviation)

            results.append(row)

    # Вывод результатов

    print("\nРезультаты бенчмарка:\n")

    print(f"{'Размер':<10}{'Little Cost':<15}{'Little Time':<15}{'Nearest Cost':<15}{'Nearest Time':<15}{'Deviation (%)':<15}")

    print("-" \* 85)

    for row in results:

        print(f"{row[0]:<10}{row[1]:<15.2f}{row[2]:<15.4f}{row[3]:<15.2f}{row[4]:<15.4f}{row[5]:<15.2f}")

    end\_time\_bench = time.time()

    print(f"Время выполнения бенчмарка: {end\_time\_bench - start\_time\_bench:.4f} секунд\n")

    # Построение графиков

    fig, axs = plt.subplots(2, 1, figsize=(10, 12))

    # График времени

    axs[0].plot(matrix\_sizes, little\_times, label='Little Method', marker='o')

    axs[0].plot(matrix\_sizes, nearest\_times, label='Nearest Method', marker='o')

    axs[0].set\_xlabel('Размер матрицы')

    axs[0].set\_ylabel('Время (секунды)')

    axs[0].set\_title('Время работы алгоритмов в зависимости от размера матрицы')

    axs[0].legend()

    axs[0].grid(True)

    # График стоимости

    axs[1].plot(matrix\_sizes, little\_costs, label='Little Method', marker='o')

    axs[1].plot(matrix\_sizes, nearest\_costs, label='Nearest Method', marker='o')

    axs[1].set\_xlabel('Размер матрицы')

    axs[1].set\_ylabel('Стоимость (единицы)')

    axs[1].set\_title('Сравнение стоимости решений для алгоритмов')

    axs[1].legend()

    axs[1].grid(True)

    plt.tight\_layout()

    plt.show()

if \_\_name\_\_ == '\_\_main\_\_':

    benchmark\_tsp([i for i in range(4, 21, 2)])

Название файла: utils.py

import numpy as np

from tabulate import tabulate

from colorama import Fore, Style

import json

INF = float('inf')

def city\_index\_to\_name(index):

    """Конвертирует индекс города в его буквенное обозначение (A, B, C и т. д.).

    Аргументы:

    index -- индекс города (int)

    Возвращает:

    Буквенное обозначение города (str)

    """

    return chr(ord('A') + index)

def generate\_matrix(size, seed=None):

    """Генерирует квадратную матрицу размера size x size со случайными значениями.

    Аргументы:

    size -- размер матрицы (int)

    seed -- начальное значение для генератора случайных чисел (int, по умолчанию None)

    Возвращает:

    matrix -- сгенерированная матрица (numpy.ndarray), диагональные элементы равны INF

    """

    if seed is not None:

        np.random.seed(seed)

    matrix = np.random.randint(1, 100, size=(size, size)).astype(float)

    np.fill\_diagonal(matrix, INF)

    return matrix

def export\_matrix(matrix, filename='export\_matrix', file\_type="txt"):

    """Экспортирует матрицу в заданный файл в зависимости от формата.

    Аргументы:

    matrix -- матрица (numpy.ndarray)

    filename -- имя файла (str), без расширения

    file\_type -- тип файла для экспорта ("txt", "csv", "json", "npy")

    Исключения:

    ValueError -- если передан неподдерживаемый тип файла или matrix не является numpy.ndarray

    """

    if not isinstance(matrix, np.ndarray):

        raise ValueError("Input matrix must be a numpy.ndarray")

    # Добавляем расширение файла в зависимости от типа

    filename\_with\_extension = f"{filename}.{file\_type}"

    if file\_type == "txt":

        np.savetxt(filename\_with\_extension, matrix, fmt='%g')

    elif file\_type == "csv":

        np.savetxt(filename\_with\_extension, matrix, delimiter=",", fmt='%g')

    elif file\_type == "json":

        with open(filename\_with\_extension, "w+") as f:

            json.dump(matrix.tolist(), f)

    elif file\_type == "npy":

        np.save(filename\_with\_extension, matrix)

    else:

        raise ValueError(f"Unsupported file type: {file\_type}. Supported types are 'txt', 'csv', 'json', 'npy'.")

    print(f"Matrix successfully exported to {filename\_with\_extension}")

def print\_matrix(matrix):

    """Выводит матрицу расстояний в удобочитаемом виде с использованием цветного форматирования.

    Аргументы:

    matrix -- матрица расстояний (numpy.ndarray)

    """

    headers = [city\_index\_to\_name(i) for i in range(len(matrix))]

    table = tabulate(matrix, headers=headers, showindex=headers, tablefmt="grid", numalign="right")

    print(f"{Fore.CYAN}Матрица расстояний:{Style.RESET\_ALL}\n{table}\n")

def print\_solution(method, best\_solution, elapsed\_time):

    """Выводит решение задачи коммивояжера в красивом формате с использованием цветного форматирования.

    Аргументы:

    method -- метод решения (str)

    best\_solution -- лучший найденный путь и его стоимость (dict)

    elapsed\_time -- время выполнения алгоритма (float)

    """

    path\_str = f"{Fore.GREEN} → {Style.RESET\_ALL}".join(city\_index\_to\_name(i) for i in best\_solution['path'])

    print(f"{Fore.YELLOW}Метод:{Style.RESET\_ALL} {method.capitalize()}")

    print(f"{Fore.MAGENTA}Лучшая стоимость пути:{Style.RESET\_ALL} {best\_solution['cost']}")

    print(f"{Fore.BLUE}Оптимальный маршрут:{Style.RESET\_ALL} {path\_str}")

    print(f"{Fore.RED}Время выполнения:{Style.RESET\_ALL} {elapsed\_time:.4f} секунд\n")

    print("=" \* 50)