

**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

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 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: $O(n^2)$

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 ($O(n^2)$) и оценка MST ($O(n^2 \log n)$) помогают сократить количество ветвей, уменьшая среднее время выполнения.

2. Метод ближайшего соседа

- Имеет сложность $O(n^2)$, 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 $O(n^2)$, так как каждый элемент заполняется случайным значением.
- Редукция матрицы (вычитание минимумов строк и столбцов) требует $O(n^2)$ времени.

4. Минимальное оставное дерево (MST)

- Используется в оценке нижней границы стоимости маршрута.
- Применение алгоритма Прима даёт сложность $O(n^2 \log n)$.

В итоге:

- Метод Литтла (ветвей и границ) – экспоненциальная сложность $O(n!)$ в худшем случае, но улучшается благодаря отсечениям.
- Ближайший сосед – $O(n^2)$, но без гарантии оптимального решения.
- Дополнительные вычисления (редукция, MST) – $O(n^2 \log n)$

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 ребра.
- `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

```
(base) [1]: %%time
Матрица расстояний:
+---+---+---+---+---+
|   | A | B | C | D | E |
+---+---+---+---+---+
| A | inf | 36 | 58 | 41 | 74 |
+---+---+---+---+---+
| B | 83 | inf | 70 | 53 | 2 |
+---+---+---+---+---+
| C | 24 | 36 | inf | 66 | 49 |
+---+---+---+---+---+
| D | 94 | 60 | 88 | inf | 65 |
+---+---+---+---+---+
| E | 13 | 41 | 73 | 21 | inf |
+---+---+---+---+---+
=====
Метод: Little
Лучшая стоимость пути: 171.0
Оптимальный маршрут: A → B → E → D → C → A
Время выполнения: 0.0005 секунд
=====
Метод: Nearest
Лучшая стоимость пути: 171.0
Оптимальный маршрут: A → B → E → D → C → A
Время выполнения: 0.0000 секунд
```

Рисунок 1. Тестирование на матрице 5x5

Матрица расстояний:							
	A	B	C	D	E	F	G
A	inf	36	58	41	74	83	69
B	70	inf	2	24	36	56	66
C	49	94	inf	88	3	65	13
D	41	73	21	inf	40	80	48
E	28	63	63	18	inf	33	22
F	42	20	23	27	51	inf	6
G	13	97	54	92	2	45	inf

Метод: Little
Лучшая стоимость пути: 158.0
Оптимальный маршрут: A → B → C → E → D → F → G → A
Время выполнения: 0.0017 секунд

Метод: Nearest
Лучшая стоимость пути: 194.0
Оптимальный маршрут: A → B → C → E → D → G → F → A
Время выполнения: 0.0000 секунд

Рисунок 2. Тестирование на матрице 7x7

Матрица расстояний:															
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
A	inf	36	58	41	74	83	69	70	53	2	24	36	56	66	49
B	94	inf	88	3	65	13	41	73	21	60	40	80	48	28	63
C	63	18	inf	33	22	42	20	23	27	51	60	6	13	97	54
D	92	2	45	inf	12	94	96	42	34	13	78	77	89	74	12
E	1	24	62	12	inf	67	60	84	19	61	28	36	54	30	22
F	34	46	94	64	63	inf	76	95	12	42	88	55	95	60	92
G	24	43	92	38	33	4	inf	29	46	60	70	46	40	56	57
H	49	79	35	53	33	28	65	inf	59	33	92	18	14	56	26
I	11	32	27	23	11	98	39	19	inf	92	62	3	40	99	4
J	80	95	30	75	35	91	71	34	12	inf	63	71	16	96	68
K	76	29	71	72	53	21	42	90	99	8	inf	22	6	31	66
L	22	99	40	61	1	71	24	19	9	23	92	inf	46	80	97
M	2	24	81	86	12	68	54	31	84	34	75	15	inf	6	30
N	75	17	49	24	56	1	72	24	48	49	26	79	25	inf	65
O	59	10	66	13	89	96	38	19	67	31	17	55	94	35	inf

Метод: Little
Лучшая стоимость пути: 178.0
Оптимальный маршрут: A → J → C → G → F → I → L → H → O → K → M → N → B → D → E → A
Время выполнения: 0.1277 секунд

Метод: Nearest
Лучшая стоимость пути: 387.0
Оптимальный маршрут: A → J → I → L → E → D → B → F → N → H → M → O → K → G → C → A
Время выполнения: 0.0001 секунд

Рисунок 3. Тестирование на матрице 15x15

Исследование.

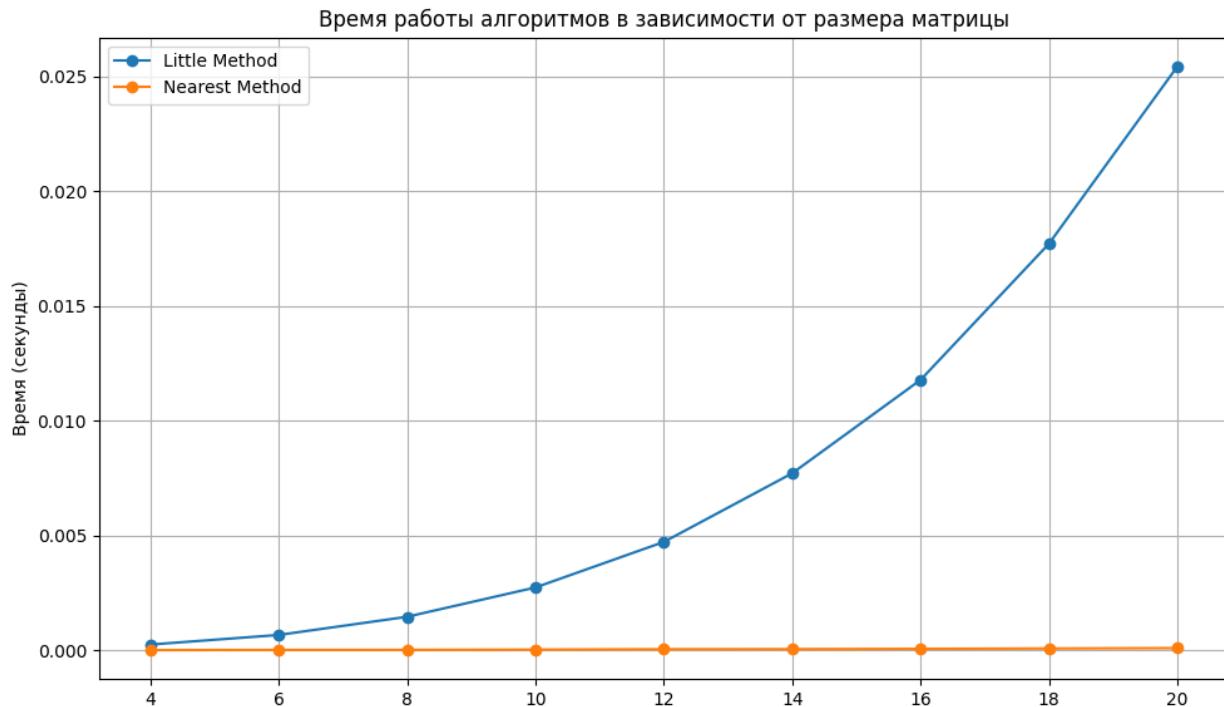


Рисунок 4 - График зависимости размера случайной матрицы от времени выполнения алгоритмов (диапазон значений матрицы 1-2)

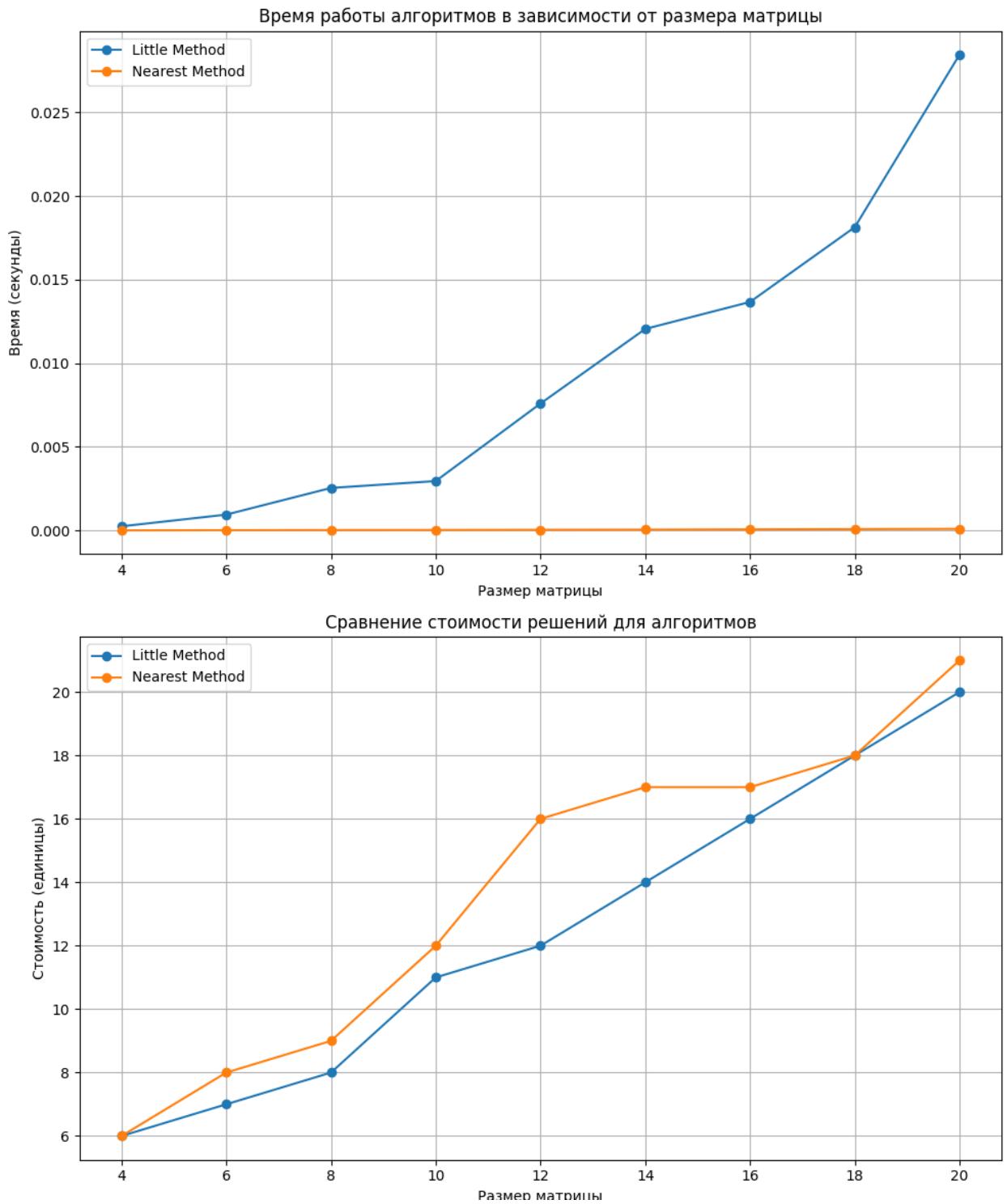


Рисунок 4 - График зависимости размера случайной матрицы от времени выполнения алгоритмов (диапазон значений матрицы 1-3)

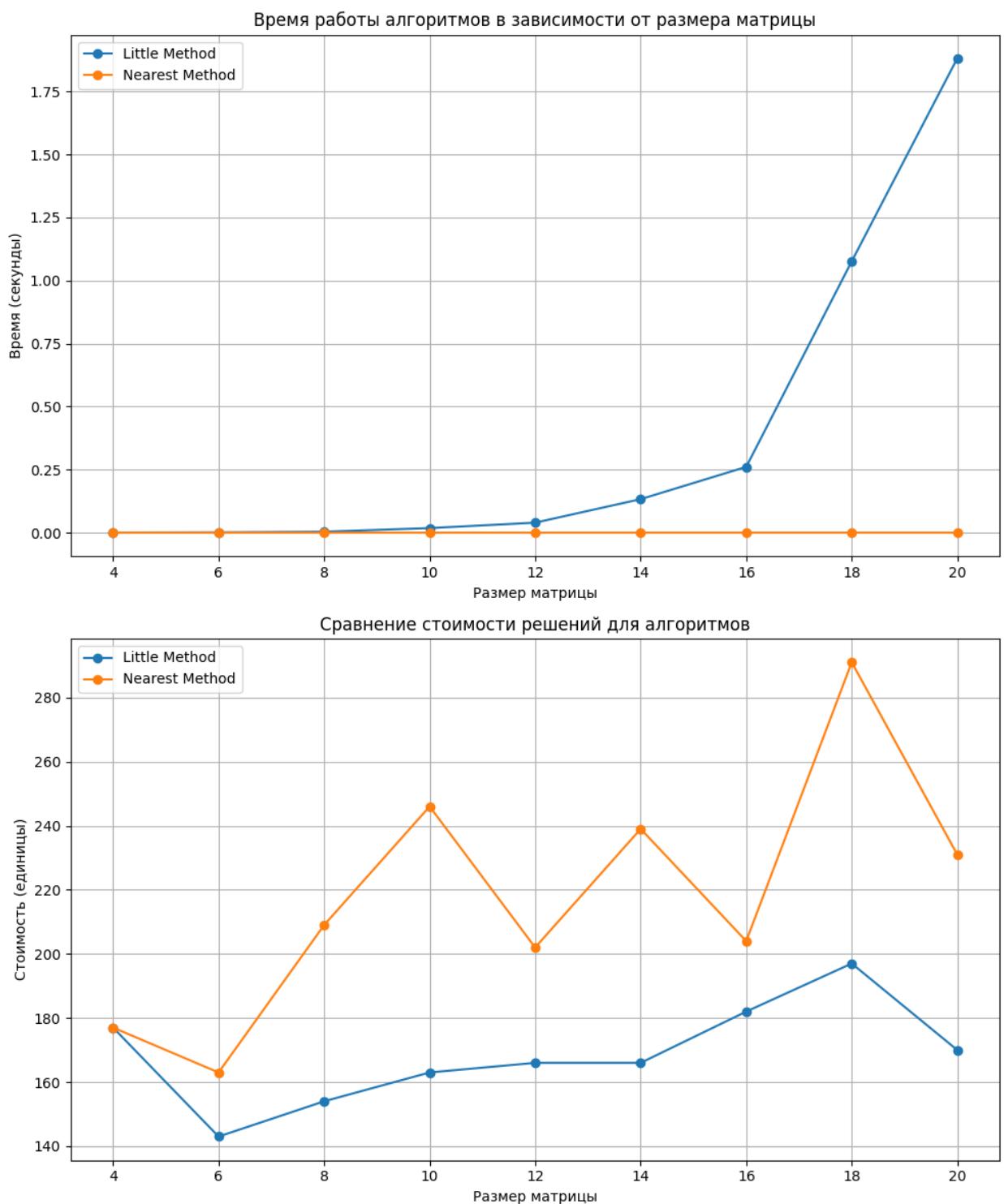


Рисунок 5 - График зависимости размера случайной матрицы от времени выполнения алгоритмов (диапазон значений матрицы 1-100)

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 выченных 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 оствовное 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 оствовного 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"\u001b[36mМатрица расстояний:\u001b[0m\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"\u001b[32m → \u001b[0m".join(city_index_to_name(i)
for i in best_solution['path'])
    print(f"\u001b[33mМетод: \u001b[0m {method.capitalize()}")
    print(f"\u001b[35mЛучшая стоимость пути: \u001b[0m{best_solution['cost']}")
    print(f"\u001b[34mОптимальный маршрут: \u001b[0m {path_str}")
    print(f"\u001b[31mВремя выполнения: \u001b[0m{elapsed_time:.4f} секунд\n")
    print("==" * 50)

```