МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе № 2

по дисциплине «Построение и Анализ Алгоритмов»

Тема: «Коммивояжёр»

Студент гр. 3343 Коршков А.А.

Преподаватель Жангиров Т. Р.

Санкт-Петербург
2025

Цель работы

Изучить различные алгоритмы для решения задачи коммивояжёра. Написать программу решения коммивояжёра через динамическое программирование (точный метод: итеративная реализация) и через АЛШ-2 (алгоритм лучшего соседа через МОД (минимальное остовное дерево)) с возможностью генерации матрицы весов (симметричной/несимметричной), сохранения её в файл и использования в программе в качестве входных данных.

Задание

Динамическое программирование

Напишите программу, решающую задачу коммивояжера. Нужно найти кратчайший маршрут, который проходит через все заданные города ровно один раз и возвращается в исходный город. Не все города могут быть напрямую связаны друг с другом.

Входные данные:

- n - количество городов ($5 \leq n \leq 15$).
- Матрица расстояний между городами размером n×n, где graph[i][j] обозначает расстояние от города i до города j. Если graph[i][j]=0 (и $i \neq j$), это означает, что прямого пути между городами нет.

Выходные данные:

- Минимальная стоимость маршрута, проходящего через все города и возвращающегося в начальный город.
- Оптимальный путь в виде последовательности посещаемых городов, начинающейся и заканчивающейся в начальном городе.
- Если такого пути не существует, вывести "no path".

Sample Input 1:

5

0 1 13 23 7

12 0 15 18 28

21 29 0 33 28

23 19 34 0 38

5 40 7 39 0

Sample Output 1:

78

0 4 2 3 1 0

Sample Input 2:

3

0 1 0

1 0 1

0 1 0

Sample Output 2:

no path

Задание варианта:

№8. Точный метод: динамическое программирование (не МВиГ), итеративная реализация.

Приближённый алгоритм: АЛШ-2.

Примечания для варианта:

Требование перед сдачей: прохождение кода в задании 3.1 на Stepik.

Замечание к варианту 8 АЛШ-2 начинать со стартовой вершины.

Основные теоретические положения

Описание алгоритмов:

Динамическое программирование. Итеративная реализация. Алгоритм Хельда-Капра.

В задаче коммивояжёра (TSP) необходимо помочь торговцу посетить все города один раз, при этом с наименьшей стоимостью. Между вершинами есть дороги с определённой стоимостью или их может не существовать. В алгоритме Хельда-Карпа задача заключается в построении гамильтонова цикла, который пройдёт по каждому городу один раз.

В реализации точного алгоритма Хельда–Карпа каждый возможный поднабор городов кодируется в виде битовой маски, что позволяет компактно представлять уже посещённые вершины и динамически наращивать путь. На этапе инициализации мы присваиваем нулевую стоимость путю, начинающемуся из города 0, а все остальные состояния заполняем “бесконечностью”.

Затем для каждой маски и для каждой вершины u, уже входящей в маску, мы рассматриваем переходы во все ещё необработанные вершины v, вычисляем новый возможный путь через u→v и обновляем стоимость соответствующего состояния dp[mask ∪ {v}][v], если найден более дешёвый маршрут.

Благодаря хранению предков (parent) мы можем после полного перебора восстановить сам маршрут. Общий размер таблицы dp равен 2^n × n, а заполнение каждой “ячейки” требует проверки всех возможных предыдущих вершин, что даёт временную сложность O(n^2 · 2^n). При этом в ходе выполнения отчётливо видно, как постепенно “растут” подмножества, от одного посещённого города до полного набора, и как обновляются минимальные стоимости.

В конце, когда все города уже включены (mask = (1<<n)–1), мы перебираем возможные “концы” маршрута i и добавляем стоимость возвращения в город 0, выбирая наименьшую сумму для завершения гамильтонова цикла. Если такая сумма остаётся бесконечной, значит, обход всех городов с возвратом в исходную точку невозможен; иначе мы печатаем оптимальную стоимость и сам маршрут в порядке посещения.

Алгоритм лучшего соседа. Нахождение оценки L через МОД. Алгоритм Прима.

Алгоритм лучшего соседа (АЛШ) является приближённым алгоритмом решения задачи коммивояжёра. Данный алгоритм необходим, если нужно вычислить путь быстрее и «проще» или сделать оценку границ при поиске для точной реализации. В большинстве случаев конечная стоимость пути выше, в некоторых случаях может равняться стоимости пути для точной реализации, но количество проходимых вершин и сами вершины идентичны (может отличаться порядок прохождения).

Алгоритм лучшего соседа (АЛШ) очень схож с алгоритмом ближайшего соседа (АБС). Если в АБС мы идём только вперёд и добавляем дугу с минимальным весом (s), то в АЛШ мы добавляем дугу с минимальным значением (s + L), где L - нижняя оценка стоимости остатка решения. Данное L можно рассчитывать на основе МОД, используя алгоритм Прима.

За основу берётся идея жадного поэтапного расширения маршрута с учётом оценки нижней границы оставшегося пути. Сначала мы отмечаем город 0 как посещённый и постепенно, на каждом шаге, из текущей вершины выбираем следующий город v, минимизирующий сумму.

Оценка нижней границы (L) вычисляется с помощью алгоритма Прима на оставшемся подграфе из не посещённых вершин: строится минимальное остовное дерево, и его суммарный вес служит “минимально возможной” дополнительной стоимостью, ведь любой гамильтонов цикл должен соединить эти вершины.

Таким образом, вместо того чтобы смотреть лишь на ближайший соседний город по весу ребра (как в классическом АБС), в АЛШ мы учитываем и более «глобальную» информацию о том, как устроен остаток графа. При каждом выборе ребра мы помечаем вершину как посещённую и добавляем стоимость перехода к общей сумме. После того как все города посещены, алгоритм пытается вернуться в исходную вершину 0— если ребро отсутствует (вес 0), выводится “no path”, иначе к итоговой стоимости добавляется расстояние до города 0, и формируется приближённый цикл.

Такой подход значительно сокращает время работы по сравнению с точным перебором, при этом даёт гарантированную оценку пригодности каждого потенциального шага и часто оказывается достаточно близким к оптимальному маршруту.

Сложность алгоритмов:

Итеративная реализация (алгоритма Хельда-Капра):

Сложность по времени: $O(n^2 * 2^n)$

Сложность по памяти: $O(n * 2^n)$

Приближённый алгоритм лучшего соседа через МОД:

Сложность по времени: $O(n^4)$

Сложность по памяти: $O(n^2)$

Выполнение работы

Описание работы

Был создан parser.py содержащий функцию get_args() -> Namespace, которая возвращает аргументы командной строки через модуль argparse. В нём есть флаги для запуска генерации матрицы (-g, --generate) , настройки количества городов (-c, --count), максимального веса ребра (--max-weight), симметричности матрицы (-s), файл, куда нужно сохранить сгенерированную матрицу (-o, --output), файл, откуда получить информацию о матрице (-i, --input), метод решения задачи (--method {exact, approx}) и вывод справки.

Файл loader.py содержит функции для генерации, сохранения и загрузки матрицы.

generate_mx(n: int, symmetric: bool = *False*, max_weight: int = 100) -> list[list[int]] – генерирует матрицу размера n, симметричную/несимметричную, с заданным максимальным размером ребра.

load_mx(file_name: str) -> tuple[int, list[list[int]]] – загружает матрицу весов из файла, возвращает количество городов и саму матрицу.

write_mx(file_name: str, weight_matrix: list[list[int]]) -> *None* – сохраняет матрицу в заданный файл

Файл tsp_exact.py содержит функцию tsp_dp(n: int, graph: list[list[int]]) -> *None* для решения задачи коммивояжёра методом динамического программирования (алгоритм Хельда-Капра).

Файл tsp_approx.py содержит функции для решения задачи коммивояжёра приближённым алгоритмом АЛШ-2 (Алгоритм лучшего соседа через оставное дерево).

prim_mst_cost(graph, nodes) -> int | float - Функция для вычисления стоимости минимального остовного дерева с использованием алгоритма Прима.

tsp_alsh2(n: int, graph: list[list[int]]) -> None – решение задачи через АЛШ-2

В main.py есть функция main() -> *None*

Тестирование

Таблица 1 – Тестирование алгоритмов

№	Входные данные	Выходные данные	Комментарии
1	5 0 1 13 23 7 12 0 15 18 28 21 29 0 33 28 23 19 34 0 38 5 40 7 39 0	78 0 4 2 3 1 0	Точный метод. Матрица несимметричная. Результат вычислен корректно
2	3 0 1 0 1 0 1 0 1 0	no path	Точный метод. Пути нет.
3	6 0 29 16 8 33 39 29 0 11 24 9 13 16 11 0 39 29 13 8 24 39 0 18 50 33 9 29 18 0 50 39 13 13 50 50 0	77 0 3 4 1 5 2 0	Точный метод. Матрица симметричная. Результат вычислен корректно.
4	5 0 1 13 23 7 12 0 15 18 28 21 29 0 33 28 23 19 34 0 38 5 40 7 39 0	84 0 4 2 1 3 0	Приближённый алгоритм. Матрица несимметричная. В отличии от точного метода конечная стоимость выше, но результат был получен быстрее. Количество пройденных городов и сами города не изменились, однако порядок городов изменился (3,1) -> (1,3).
5	3 0 1 0	no path	Приближённый алгоритм. Пути нет.

	1 0 1 0 1 0		
6	6 0 29 16 8 33 39 29 0 11 24 9 13 16 11 0 39 29 13 8 24 39 0 18 50 33 9 29 18 0 50 39 13 13 50 50 0	98 0 3 4 1 2 5 0	Приближённый алгоритм. Матрица симметричная. Как и в случае 4 алгоритм отработал быстро и с большей стоимостью. Порядок городов изменён (5, 2) -> (2, 5).

Выводы

Был реализован алгоритм, решающий алгоритм Коммивояжёра через динамическое программирование с итеративным подходом и через приближённый алгоритм лучшего соседа через МОД. Также написаны функции для генерации, загрузки и сохранения матрицы весов.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
"""
Главный файл программы.

Вариант 8 Точный метод: динамическое программирование (не МВиГ), ите-
ративная
реализация.
Приближённый алгоритм: АЛШ-2.
Требование перед сдачей: прохождение кода в задании 3.1 на Stepik.
Замечание к варианту 8: АЛШ-2 начинать со стартовой вершины.
"""
from argparse import Namespace

from modules.tsp_exact import tsp_dp
from modules.tsp_approx import tsp_alsh2

from modules.loader import load_mx, write_mx, generate_mx
from modules.parser import get_args


def main() -> None:
    """
    Главная функция программы.
    :return: None
    """
    args: Namespace = get_args()

    # Генерация матрицы
    if args.generate:
        file_name: str = args.output
        n: int = args.count
        if not 5 <= args.count <= 15:
            print("Значение аргумента должно быть в диапазоне [5,
15].")
            return
        graph_mx: list[list[int]] = generate_mx(n, args.symmetric,
args.max_weight)
        if args.output:
            file_name = args.output
        try:
            write_mx(file_name, graph_mx)
        except IOError:
            print(f"Ошибка записи в файл '{file_name}'. Проверьте
права доступа.")
            return
        print(
            f"Матрица {n}×{n}"
            f"{' (симметричная)' if args.symmetric else ''} "
            f"сохранена в файл '{file_name}'"
        )
        return
```

```
    if args.input:
        # Чтение из файла
        try:
            n, graph_mx = load_mx(args.input)
        except FileNotFoundError:
            print(f"Файл '{args.input}' не найден.")
            return
        except ValueError:
            print("Неверный формат файла.")
            return
    else:
        # Чтение с stdin
        n: int = int(input())
        graph_mx: list[list[int]] = [[int(i) for i in input().split()]
for _ in range(n)]

    # Выбор метода решения
    if args.method == "exact":
        # Точный метод
        tsp_dp(n, graph_mx)
    elif args.method == "approx":
        # Приближённый алгоритм
        tsp_alsh2(n, graph_mx)
    else:
        # Неизвестный метод
        print(
            "Неизвестный метод решения задачи коммивояжёра. "
            "Используйте 'exact' или 'approx'."
        )
        return


if __name__ == '__main__':
    main()
```

Название файла: parser.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
"""
Это файл содержит функции для парсинга аргументов командной строки.
"""
from argparse import ArgumentParser, Namespace, RawTextHelpFormatter


def get_args() -> Namespace:
    """
    Получить аргументы командной строки.
    :return: Аргументы командной строки
    """
    # Создание парсера аргументов
    parser: ArgumentParser = ArgumentParser(
        description=(
            "description:\n"
            "Решение задачи коммивояжёра двумя методами:\n"
            "    1) Точный (ДП, рекурсивно, старт из 0)\n"
            "    2) Приближённый (АЛШ-2, вариант 8)\n"
            "Кроме того, можно сгенерировать матрицу (произвольную или
```

```
симметричную),\n"
            "сохранить её в файл и затем читать из файла."
        ),
        epilog=(
            "epilog:\n"
            "ДП (DP) - Динамическое программирование\n"
            "АЛШ-2 - Алгоритм лучшего соседа (на основе МОД) \n"
            "В файле матрицы весов на первой строке указывается коли-
чество городов (n), "
            "на следующих n строках — матрица.\n"
        ),
        formatter_class=RawTextHelpFormatter)

    # Параметры командной строки
    parser.add_argument("-i", "--input", type=str, dest="input",
                        help=(
                            "Имя файла с матрицей. "
                            "Если не указан, читаем из stdin."
                        ))
    parser.add_argument("-o", "--output", type=str, dest="output",
default="output.txt",
                        help="Имя файла, в который сохраняем матрицу.
По умолчанию output.txt.")
    parser.add_argument("-g", "--generate", action="store_true",
dest="generate",
                        help="Сгенерировать новую матрицу весов.")
    parser.add_argument("-c", "--count", type=int, default=10,
dest="count",
                        help="Количество городов (5 ≤ n ≤ 15). По
умолчанию 10.")
    parser.add_argument("-s", "--symmetric", action="store_true",
dest="symmetric",
                        help="Сгенерировать симметричную матрицу. По
умолчанию нет.")
    parser.add_argument("--max-weight", type=int, default=100,
dest="max_weight",
                        help="Максимальный вес ребра. По умолчанию
100.")
    parser.add_argument("--method", choices=["exact", "approx"], de-
fault="exact",
                        help=(
                            "Метод решения: exact — точный (DP), "
                            "approx — приближённый (АЛШ-2). "
                            "По умолчанию exact."
                        ))
    args: Namespace = parser.parse_args()
    return args
```

Название файла: loader.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
"""
Модуль для загрузки и сохранения матрицы весов
"""
from random import randint


def generate_mx(n: int, symmetric: bool = False, max_weight: int =
```

```
100) -> list[list[int]]:
    """
    Функция для генерации матрицы весов.
    :param symmetric: Если True, то матрица будет симметричной
    :param n: Размерность матрицы
    :param max_weight: Максимальный вес ребра
    :return: list[list[int]] Матрица весов
    """
    mx: list[list[int]] = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(i + 1, n):
            w: int = randint(0, max_weight)
            mx[i][j]: int = w
            mx[j][i]: int = w if symmetric else randint(0, max_weight)
    return mx


def load_mx(file_name: str) -> tuple[int, list[list[int]]]:
    """
    Функция для загрузки файла.
    :param file_name: Имя файла
    :return: list[str] | None Список строк из файла
    """
    with open(file=file_name, mode="rt", encoding="UTF-8") as file:
        n: int = int(file.readline().strip())
        return n, [[int(i) for i in line.strip().split()] for line in
file.readlines()]


def write_mx(file_name: str, weight_matrix: list[list[int]]) -> None:
    """
    Функция для записи в файл.
    :param weight_matrix: Матрица весов
    :param file_name: Имя файла
    :return: None
    """
    with open(file=file_name, mode="wt", encoding="UTF-8") as file:
        file.write(f"{len(weight_matrix)}\n")
        for row in weight_matrix:
            file.write(" ".join(map(str, row)) + "\n")
        return None
```

Название файла: tsp_approx.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
"""
Данный модуль содержит реализацию приближенного решения задачи ком-
мивояжера
через алгоритм лучшего соседа (на основе МОД).
"""
INF: float = float("inf")  # Бесконечность для инициализации рас-
стояний


def prim_mst_cost(graph: list[list[int]], nodes: list[int]) -> int
| float:
    """
```

```
        Функция для вычисления стоимости минимального остовного дерева
        с использованием алгоритма Прима.
        Приближённый алгоритм.
        :param graph: Матрица смежности графа
        :param nodes: Список вершин, для которых нужно найти MST
        :return: Стоимость MST или INF, если нет пути
        """
        print(f"\n  [MST] Начало расчета для узлов: {nodes}")
        visited: list[bool] = [False] * len(graph)
        min_edge: list[int | float] = [INF] * len(graph)
        min_edge[nodes[0]]: int = 0  # Стартовая вершина
        total: int = 0

        for step in range(len(nodes)):
            u: int = -1
            for v in nodes:
                if not visited[v] and (u == -1 or min_edge[v] <
min_edge[u]):
                    u: int = v
            print(f"  [MST] Шаг {step + 1}: Выбрана вершина {u} с весом
{min_edge[u]}")
            if min_edge[u] == INF:
                print("  [MST] Невозможно построить MST!")
                return INF
            visited[u]: bool = True
            total += min_edge[u]
            print(f"  [MST] Добавлено в MST: {u}, текущая стоимость:
{total}")
            for v in nodes:
                if graph[u][v] != 0 and not visited[v]:
                    new_weight: int = graph[u][v]
                    if new_weight < min_edge[v]:
                        print(f"  [MST] Обновлен вес для вершины {v}
-> {graph[u][v]}")
                        min_edge[v]: int = new_weight
        print(f"  [MST] Итоговая стоимость MST: {total}\n")
        return total


    def tsp_alsh2(n: int, graph: list[list[int]]) -> None:
        """
        Функция для решения задачи коммивояжёра методом АЛШ-2 (Алгоритм
лучшего соседа).
        :param n: Количество городов
        :param graph: Матрица весов
        :return: None
        """
        visited: list[bool] = [False] * n
        visited[0]: int = True

        path: list[int] = [0]  # Начинаем со стартовой вершины
        total_cost: int = 0
        current: int = 0

        print("Начало алгоритма. Стартовый город: 0")
        for step in range(n - 1):  # Последовательно добавляем лучший
следующий город
```

```
            print(f"\nШаг {step + 1}: Текущий путь: {' → '.join(map(str,
path))}")
            print(f"Текущая стоимость: {total_cost}")
            print(f"Ищем следующий город из {current}...")

            best_next: int = -1
            best_cost: float = INF
            for v in range(n):
                if not visited[v] and graph[current][v] != 0:
                    remaining: list[int] = [i for i in range(n) if not
visited[i] and i != v]
                    l_bound: int | float = prim_mst_cost(graph, re-
maining) if remaining else 0
                    cost: int | float = graph[current][v] + l_bound
                    print(
                        f"  Город {v}: стоимость перехода={graph[cur-
rent][v]}, "
                        f"оценка MST={l_bound}, общая оценка={cost}"
                    )
                    if cost < best_cost:
                        best_cost: int = cost
                        best_next: int = v
            if best_next == -1:
                print("no path")
                return
            total_cost += graph[current][best_next]
            current: int = best_next
            path.append(current)
            visited[current]: bool = True

        # Завершаем цикл, возвращаясь в стартовый город
        print(f"\nФинальный переход из {current} в стартовый город 0")
        if graph[current][0] == 0:
            print("no path")
            return
        total_cost += graph[current][0]
        path.append(0)
        print("\nРезультат:")
        print(total_cost)
        print(" ".join(map(str, path)))
```

Название файла: tsp_exact.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
"""
Данный модуль содержит реализацию точного решения задачи коммивояжера
через динамическое программирование.
"""
INF: float = float("inf")  # Бесконечность для инициализации расстоя-
ний


def tsp_dp(n: int, graph: list[list[int]]) -> None:
    """
    Функция для решения задачи коммивояжёра методом динамического про-
граммирования.
    Итеративная реализация.
    Алгоритм Хельда-Карпа.
```

```
    :param n: Количество городов
    :param graph: Матрица весов
    :return: None
    """
    # Инициализация таблиц DP и предков
    dp: list[list[int | float]] = [[INF] * n for _ in range(1 << n)]
    parent: list[list[int]] = [[-1] * n for _ in range(1 << n)]
    print("Инициализация DP таблицы. Начальная точка 0 с стоимостью
0")
    dp[1][0]: int = 0  # Стартуем из города 0

    # Перебор всех подмножеств вершин
    print("\nЭтапы обновления путей:")
    for mask in range(1 << n):
        print(f"\nОбработка маски: {bin(mask)}")
        for u in range(n):
            if not (mask & (1 << u)):
                print(f"  Город {u} не в маске — пропуск")
                continue
            print(f"  Текущий город u={u}")
            for v in range(n):
                if mask & (1 << v) or graph[u][v] == 0:
                    print(f"  Город v={v} уже в маске")
                    continue
                if graph[u][v] == 0:
                    print(f"    Нет пути из {u} в {v} — пропуск")
                    continue
                next_mask: int = mask | (1 << v)
                new_cost: int = dp[mask][u] + graph[u][v]
                print(
                    f"    Проверка перехода u={u} -> v={v}:
next_mask={bin(next_mask)}, "
                    f"возможная стоимость={new_cost} (текущая
dp={dp[next_mask][v]})"
                )
                if new_cost < dp[next_mask][v]:
                    dp[next_mask][v]: int = new_cost
                    parent[next_mask][v]: int = u
                    print(f"    Обновление: dp[{bin(next_mask)}][{v}]
= {new_cost}, parent={u}")

    # Поиск минимального пути возвращения в начальный город
    full_mask: int = (1 << n) - 1
    min_cost: int | float = INF
    last: int = -1
    print("\nПоиск минимального пути возврата в город 0:")
    for i in range(1, n):
        cost: int | float = dp[full_mask][i] + graph[i][0] if
graph[i][0] != 0 else INF
        print(f"  Город {i}: стоимость пути через него = {cost}")
        if cost < min_cost:
            min_cost: int = dp[full_mask][i] + graph[i][0]
            last: int = i

    # Если не найден путь, выводим сообщение (наименьшая стоимость
остаётся бесконечностью)
    if min_cost == INF:
        print("no path")
```

```
    return

# Восстановление пути
print(f"\nМинимальная стоимость: {min_cost}")
print("Восстановление пути:")
path: list[int] = [0]
mask: int = full_mask
while last != -1:
    print(f"  Текущий узел: {last}, маска: {bin(mask)}")
    path.append(last)
    temp: int = parent[mask][last]
    mask ^= (1 << last)
    last: int = temp
path.reverse()

print("\nРезультат:")
print(min_cost)
print(" ".join(map(str, path)))
```