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

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Построение и анализ алгоритмов»

Тема: Коммивояжер (TSP)

Студент гр. 1304 Заика Т.П.

Преподаватель Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Изучить методы решения задачи коммивояжера, разработать алгоритм решения на основе метода ветвей и границ с оптимизациями.

Задание.

Дана карта городов в виде ассиметричного, неполного графа $G = (V, E)$, где $V(|V|=n)$ – это вершины графа, соответствующие городам; $E(|E|=m)$ – это ребра между вершинами графа, соответствующие путям сообщения между этими городами.

Каждому ребру m_{ij} (переезд из города i в город j) можно сопоставить критерий выгодности маршрута (вес ребра) равный w_i (натуральное число [1, 1000]), $m_{ij}=inf$, если $i=j$.

Если маршрут включает в себя ребро m_{ij}, то $x_{ij}=1$, иначе $x_{ij}=0$.

Требуется найти минимальный маршрут (минимальный гамильтонов цикл):

$$minW = ¿\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij}w_{ij}¿$$

Выполнение работы.

В ходе работы было определено, что в качестве метода решения задачи коммивояжера используется метод ветвей и границ с следующими оптимизациями:

1. За начальный лучший результат берется результат, найденный жадным алгоритмом.

2. Исключаются решения с весом пути больше, чем текущий лучший результат.

3. Исключаются решения с весом пути больше, чем текущий вес пути плюс вес минимального остового дерева из оставшихся вершин.

Разработаны следующие классы и методы для реализации решения:

1. Класс вершины *Vertex* с следующими методами:

1) *get_distance_cost* — возвращает стоимость пути между вершиной и смежной с ней;

2) *set_distance_cost* — устанавливает стоимость пути между вершиной и смежной с ней;

3) *sort_indexes* — сортирует индексы смежных вершин с данной;

4) *get_sorted_indexes* — возвращает список отсортированных индексов смежных вершин с данной.

2. Класс графа *Graph* с следующими методами:

1) *get_graph_size* — возвращает количество вершин в графе;

2) *get_distance_cost_between_vertexes* — возвращает стоимость пути между двумя вершинами;

3) *set_distance_cost_between_vertexes* — устанавливает стоимость пути между двумя верщинами;

4) *get_way_cost* — возвращает стоимость пути;

5) *get_vertex* — возвращает вершину графа;

6) *sort_indexes_of_each_vertex* — сортирует индексы смежных вершин у каждой вершины графа.

3. Класс решения *Solution* с следующими методами:

1) *__find_best_way_by_greedy* — ищет минимальный гамильтонов цикл жадным алгоритмом;

2) *__find_min_spanning_tree_cost* — ищет стоимость минимального остового дерева;

3) *__find_best_way_by_branch_and_bound* — ищет минимальный гамильтонов цикл методом ветвей и границ;

4) *find_best_way* — ищет минимальный гамильтонов цикл методом ветвей и границ с оптимизацией.

4. Класс решения задачи коммивояжера, считывающий исходные данные с файла, *TSPSolverFromFile* с следующими методами:

1) *__init_graph_form_file* — инициализирует граф из исходных данных из файла;

2) *__format_way* — форматирует список индексво вершин минимального гамильтонова цикла для заданного формата ответа;

3) *solve* — решает задачу коммивояжера;

4) *print_answer* — печатает ответ на задачу коммивояжера в заданном формате.

Разработанный программный код см. в приложении А.

Тестирование представлено в приложении Б.

Среднее время выполнения на приведенных тестах — 0.5 мс

Выводы.

Исследованы, изучены методы решения задачи коммивояжера, разработан алгоритм решения на основе метода ветвей и границ с оптимизациями. Использованы оптимизации на основе начального лучшего найденного пути жадным алгоритмом, исключения решений, чей текущий вес результата превышает текущий лучший результат, а также исключения решений, чей текущий вес результата плюс вес минимального остового дерева из оставшихся вершин превышает текущий лучший результат. Также предусмотрен ответ на задачу в том случае, если в графе нет гамильтонова цикла. Все предложенные тесты приведены в приложении Б и успешно пройдены.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
import time
# Этот модуль предоставляет различные функции для управления значениями времени
from copy import deepcopy
# Операция глубокого копирования произвольных объектов Python
from math import inf
# Положительная бесконечность с плавающей запятой


class Vertex:
    """Класс вершины графа"""
    def __init__(self, amount_of_neighbours: int) -> None:
        """
        Инициализирует вершину
        :param amount_of_neighbours: количество смежных вершин с данной
        """
        self.__distances = dict()
        for i in range(amount_of_neighbours):
            self.__distances[i] = inf
        self.__sorted_indexes = list()

    def get_distance_cost(self, index: int) -> float:
        """
        Возвращает стоимость пути между вершиной и смежной с ней
        :param index: индекс смежной вершины
        :return: стоимость пути между вершиной и смежной с ней
        """
        return self.__distances[index]

    def set_distance_cost(self, index: int, cost: int) -> None:
        """
        Устанавливает стоимость пути между вершиной и смежной с ней
        :param index: индекс смежной вершины
        :param cost: стоимость пути между вершиной и смежной с ней
        :return:
        """
        self.__distances[index] = cost

    def sort_indexes(self) -> None:
        """
        Сортирует индексы смежных вершин с данной
        :return:
        """
        self.__sorted_indexes = list(map(lambda x: x[0], sorted(self.__distances.items(), key=lambda x: x[1])))

    def get_sorted_indexes(self) -> list[int]:
        """
        Возвращает список отсортированных индексов смежных вершин с данной
```

```
            :return: список отсортированных индексов смежных вершин с
данной
            """
            return self.__sorted_indexes


    class Graph:
        """Класс графа"""
        def __init__(self, size: int) -> None:
            """
            Инициализирует граф
            :param size: количество вершин в графе
            """
            self.__size = size
            self.__graph_matrix = [Vertex(size) for _ in range(size)]

        def get_graph_size(self) -> int:
            """
            Возвращает количество вершин в графе
            :return: количество вершин в графе
            """
            return self.__size

        def   get_distance_cost_between_vertexes(self,   source:   int,
destination: int) -> float:
            """
            Возвращает стоимость пути между двумя вершинами
            :param source: индекс первой вершины
            :param destination: индекс второй вершины
            :return: стоимость пути между двумя вершинами
            """
            return
self.__graph_matrix[source].get_distance_cost(destination)

        def   set_distance_cost_between_vertexes(self,   source:   int,
destination: int, cost: int) -> None:
            """
            Устанавливает стоимость пути между двумя вершинами
            :param source: индекс первой вершины
            :param destination: индекс второй вершины
            :param cost: стоимость пути между двумя вершинами
            :return:
            """
            self.__graph_matrix[source].set_distance_cost(destination,
cost)

        def get_way_cost(self, distances: list[int]) -> int:
            """
            Возварщает стоимость пути
            :param distances: список индексов вершин
            :return: стоимость пути
            """
            summary_cost = 0

            for i in range(1, len(distances)):
                summary_cost     +=     self.__graph_matrix[distances[i-
1]].get_distance_cost(distances[i])
```

```
            return summary_cost

        def get_vertex(self, index: int) -> Vertex:
            """
            Возвращает вершину графа
            :param index: индекс вершины
            :return: вершина графа
            """
            return self.__graph_matrix[index]

        def sort_indexes_of_each_vertex(self) -> None:
            """
            Сортирует индексы смежных вершин у каждой вершины графа
            :return:
            """
            for vertex in self.__graph_matrix:
                vertex.sort_indexes()


    class Solution:
        """Класс решения"""
        def __init__(self, graph_matrix) -> None:
            """
            Инициализирует класс решения
            :param graph_matrix: матрица графа
            """
            self.__graph_matrix = graph_matrix
            self.__best_way = list()
            self.__best_way_cost = inf

        def __find_best_way_by_greedy(self, start_vertex_index: int =
0, excluded_vertexes: list[int] = None) -> list[int]:
            """
            Ищет минимальный гамильтонов цикл жадным алгоритмом
            :param start_vertex_index: индекс начальной вершины
            :param excluded_vertexes: список индексов просмотренных
вершин
            :return: список индексов вершин в минимальном гамильтоновом
цикле
            """
            if excluded_vertexes is None:
                excluded_vertexes = list()
            greedy_way = [start_vertex_index]

            while len(greedy_way) + len(excluded_vertexes) <
self.__graph_matrix.get_graph_size():
                cur_vertex_index = greedy_way[-1]
                sorted_indexes =
self.__graph_matrix.get_vertex(cur_vertex_index).get_sorted_indexes()
                for index in sorted_indexes:
                    if (index in greedy_way) or (index in
excluded_vertexes):
                        continue
                    greedy_way.append(index)
                    break
                if cur_vertex_index == greedy_way[-1]:
                    break
            greedy_way.append(start_vertex_index)
```

```
            return greedy_way

        def __find_min_spanning_tree_cost(self, start_vertex_index:
int, excluded_vertexes: list[int]) -> int:
            """
            Ищет стоимость минимального остового дерева
            :param start_vertex_index: индекс начальной вершины
            :param excluded_vertexes: список индексов просмотренных
вершин
            :return: стоимость минимального остового дерева
            """
            min_spanning_tree_cost = 0
            visited_vertexes = []
            visited_indexes_stack = [start_vertex_index]

            while len(visited_indexes_stack) > 0:
                cur_vertex_index = visited_indexes_stack[-1]
                min_cost_index = -1
                sorted_indexes                                          =
self.__graph_matrix.get_vertex(cur_vertex_index).get_sorted_indexes()
                for index in sorted_indexes:
                    if index in visited_vertexes or index in
visited_indexes_stack or index in excluded_vertexes:
                        continue
                    if
self.__graph_matrix.get_distance_cost_between_vertexes(cur_vertex_index,
index) == inf:
                        break
                    min_cost_index = index
                if min_cost_index == -1:

visited_vertexes.append(visited_indexes_stack.pop())
                else:
                    visited_indexes_stack.append(min_cost_index)
                    min_spanning_tree_cost                              +=
self.__graph_matrix.get_distance_cost_between_vertexes(cur_vertex_index,
min_cost_index)

            return min_spanning_tree_cost

        def __find_best_way_by_branch_and_bound(self, curr_way:
list[int], curr_way_cost: int) -> None:
            """
            Ищет минимальный гамильтонов цикл методом ветвей и границ
            :param curr_way: текущий путь
            :param curr_way_cost: стоимость текущего пути
            :return:
            """
            if curr_way_cost >= self.__best_way_cost:
                return
            if len(curr_way) == self.__graph_matrix.get_graph_size():
                new_cost          =          curr_way_cost          +
self.__graph_matrix.get_distance_cost_between_vertexes(curr_way[-1],
curr_way[0])
                if new_cost < self.__best_way_cost:
                    new_way = deepcopy(curr_way)
                    new_way.append(curr_way[0])
```

```
                    self.__best_way_cost = new_cost
                    self.__best_way = new_way
                return
            if curr_way_cost + self.__graph_matrix.get_graph_size() -
len(curr_way) + 1 >= self.__best_way_cost:
                return
            if                    curr_way_cost                    +
self.__find_min_spanning_tree_cost(curr_way[-1],      curr_way)      >=
self.__best_way_cost:
                return

            for i in range(self.__graph_matrix.get_graph_size()):
                if i in curr_way:
                    continue
                curr_way.append(i)
                curr_way_cost                                        +=
self.__graph_matrix.get_distance_cost_between_vertexes(curr_way[-2],
curr_way[-1])
                self.__find_best_way_by_branch_and_bound(curr_way,
curr_way_cost)
                curr_way.pop()

        def find_best_way(self) -> list[int]:
            """
            Ищет минимальный гамильтонов цикл методом ветвей и границ с
оптимизацией
            :return: список индексов вершин в минимальном гамильтоновом
цикле
            """
            self.__best_way = self.__find_best_way_by_greedy(0, [])
            self.__best_way_cost                                     =
self.__graph_matrix.get_way_cost(self.__best_way)

            for i in range(self.__graph_matrix.get_graph_size()):
                self.__find_best_way_by_branch_and_bound([i], 0)

            return self.__best_way


    class TSPSolverFromFile:
        """Класс решения задачи коммивояжера, считывающий исходные
данные с файла"""
        def __init__(self, file_name: str) -> None:
            """
            Инициализирует класс решения задачи коммивояжера,
считывающий исходные данные с файла
            :param file_name: имя файла для считывания исходных данных
            """
            self.__file_name = file_name
            self.__size = None
            self.__graph_matrix = None
            self.__solver = None
            self.__answer_way = list()
            self.__answer_way_cost = None
            self.__work_time = None

        def __init_graph_from_file(self) -> None:
            """
```

```
            Инициализирует граф из исходных данных из файла
            :return:
            """
            file = open(self.__file_name, 'r')
            self.__size = int(file.readline())
            self.__graph_matrix = Graph(self.__size)
            self.__solver = Solution(self.__graph_matrix)

            vertex_index = 0
            for line in file:
                formatted_row = list(map(lambda x: inf if x.strip() in
["inf", "-"] else int(x), line.split()))
                for i in range(len(formatted_row)):

self.__graph_matrix.set_distance_cost_between_vertexes(vertex_index,   i,
formatted_row[i])
                vertex_index += 1

            file.close()

        def __format_way(self) -> None:
            """
            Форматирует    список    индексов    вершин    минимального
гамильтонова цикла для заданного формата ответа
            :return:
            """
            for i in range(len(self.__answer_way)):
                self.__answer_way[i] += 1

        def solve(self) -> None:
            """
            Решает задачу коммивояжера
            :return:
            """
            self.__init_graph_from_file()
            start_time = time.time()
            self.__graph_matrix.sort_indexes_of_each_vertex()
            self.__answer_way = self.__solver.find_best_way()
            self.__answer_way_cost                                          =
self.__graph_matrix.get_way_cost(self.__answer_way)
            self.__work_time = (time.time() - start_time) * 1000
            self.__format_way()

        def print_answer(self) -> None:
            """
            Печатает ответ на задачу коммивояжера в заданном формате
            :return:
            """
            if self.__answer_way_cost != inf:
                print(self.__answer_way,        self.__answer_way_cost,
str(self.__work_time) + "mc")
            else:
                print("В данном графе нет гамильтонова цикла")


    if __name__ == "__main__":
        solver = TSPSolverFromFile("input.txt")
```

```
solver.solve()
solver.print_answer()
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Таблица Б.1 - Примеры тестовых случаев

№ п/п	Входные данные	Выходные данные	Комментарии
1.	4 inf 1 2 2 - inf 1 2 - 1 inf 1 1 1 - inf	[1, 2, 3, 4, 1] 4 0.022649765014648438mc	Верный результат на тесте из примера
2.	4 inf inf inf inf - inf 1 2 - 1 inf 1 1 1 - inf	В данном графе нет гамильтонова цикла	Граф с одной висячей вершиной
3.	3 inf 10 1 10 inf 2 3 7 inf	[1, 2, 3, 1] 15 0.05316734313964844mc	Полный симметричный граф из трех вершин
4.	2 inf 2 2 inf	[1, 2, 1] 4 0.028133392333984375mc	Полный симметричный граф из двух вершин
5.	20 Симметричный полный граф с случайными весами ребер между вершинами (аналитическое представление теста в таблице не является конструктивным в силу размера матрицы)	[1, 4, 16, 6, 9, 15, 13, 18, 20, 17, 2, 11, 7, 14, 12, 3, 8, 10, 19, 5, 1] 282 2.078533172607422mc	Предельный случай размера матрицы графа, предложенный в задании