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 Шаврин А.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):

$$\min W = \sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij}\, w_{ij}$$

Входные параметры:

Матрица графа из текстового файла.

inf 1 2 2

- inf 1 2

- 1 inf 1

1 1 - inf

Выходные параметры:

Кратчайший путь, вес кратчайшего пути, скорость решения задачи.

[1, 2, 3, 4, 1], 4, 0mc

// Задача должна решаться на размере матрицы 20x20 не дольше 3 минут в среднем.

Выполнение работы.

1. В ходе работы было определено, что в качестве оптимизации метода ветвей и границ для решения задачи коммивояжера будут использоваться следующие методы:

- Исключаются решения с весом пути больше, чем текущий лучший результат.
- Исключаются решения с весом пути плюс вес минимального остовного дерева от оставшихся вершин больше, чем текущий лучший результат.

2. Сначала был написан класс Node.

Данный класс имеет следующие приватные поля:

- __children_weight – словарь, в котором ключ – номер дочерней вершины, значение – вес дуги до нее.
- __children – список номеров дочерних вершин.

Данный класс имеет следующие методы:

- __init__ - инициализирует начальные параметры (принимает целочисленную степень вершины).
- __getitem__ - возвращает вес дуги до переданного номера дочернего узла (принимает номер дочернего узла).
- __setitem__ - устанавливает вес дуги до переданного номера дочернего узла (принимает номер дочернего узла).
- sort_children – сортирует список номеров дочерних узлов по возрастанию веса дуг до них.
- get_children – возвращает список номеров дочерних узлов.

3. Затем был написан класс Graph.

Данный класс имеет следующие приватные поля:

- __count_nodes – количество узлов в графе.
- __nodes – список узлов графа.

Данный класс имеет следующие методы:

- __init__ - инициализирует начальные параметры (принимает целочисленное значение количества узлов).
- sort – сортирует списки дочерних узлов у всех узлов.
- __len__ - возвращает количество узлов графа.
- __getitem__ - возвращает объект узла с переданным номером (принимает номер узла).

4. Затем был написан класс Solver, реализующий нахождение гамильтонова цикла.

Данный класс имеет следующие приватные поля:

- __graph – объект класса графа.
- __start_node_number – номер узла, с которого начинается решение.
- __best_hamiltonian_cycle – лучший гамильтонов цикл.
- __best_hamiltonian_cycle_weight – лучший вес гамильтонова цикла.

Данный класс имеет следующие методы:

- __init__ - инициализирует начальные параметры (принимает граф).

- __lower_bound – вычисляет нижнюю границу для метода ветвей и границ (принимает номер стартового узла и текущий гамильтонов путь).
- __find_next_node_number – находит номер лучшего узла для следующей итерации (принимает номер стартового узла, текущий гамильтонов путь и матрицу смежности посещенных ветвей).
- __branches_and_boundaries_algorithm – реализует алгоритм метода ветвей и границ (принимает текущий номер узла, текущий гамильтонов путь, вес текущего гамильтонова пути и матрицу смежности посещенных ветвей).
- find_hamiltonian_cycle – реализует поиск гамильтонова цикла.

5. После был написан класс App, являющийся оберткой над всей задачей.

Данный класс имеет следующие приватные поля:

- __garph – объект класса Graph.
- __solver – объект класса Solver.
- __answer_hamiltonian_cycle – гамильтонов цикл.
- __answer_hamiltonian_cycle_weight – вес гамильтонова цикла.
- __work_time – время работы решения.

Данный класс имеет следующие методы:

- __init__ - инициализирует начальные параметры (принимает имя файла для считывания графа).
- __init_grap – создает объект графа и заполняет его (принимает имя файла для считывания графа).
- __format_answer – приводит ответ к нужному виду.
- solve – реализует решение всей задачи.
- __str__ - приводит объект класса к строковому виду.

- __repr__ - возвращает официальную строковую версию объекта класса.

6. В конце была написана функция main, в которой происходит решение задачи на 6-ти тестовых файлах.

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

Выводы.

Изучено решение задачи коммивояжера по построению в графе гамильтонова цикла с минимальной стоимостью ребер, при помощи метода ветвей и границ.

В работе используется рекурсивный обход графа, а для уменьшения временных затрат используются следующие оптимизации:

- Исключаются решения с весом пути больше, чем текущий лучший результат (стандартное решение).
- Исключаются решения с весом пути плюс вес минимального остовного дерева от оставшихся вершин больше, чем текущий лучший результат.

При невозможности построения гамильтонова цикла, программа выводит соответствующее сообщение.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
from time import time
from copy import deepcopy
from math import inf


class Node:
    """
    Graph node class that stores the weights of branches to neighboring nodes and a sorted list
    of numbers of child nodes by the cost of the path to them
    """

    def __init__(self, node_degree: int) -> None:
        """
        This method initializes the initial parameters of the graph node
        :param node_degree: maximum number of child nodes (number of outgoing branches)
        """

        self.__children_weight = dict()
        for node_number in range(node_degree):
            self.__children_weight[node_number] = inf
        self.__children = list()

    def __getitem__(self, child_node_number: int) -> float:
        """
        This method returns the weight of the branch to the passed child node
        :param child_node_number: the number of the child node to which you need to get the cost of the path
        :return: weight of the branch to the passed child node
        """

        return self.__children_weight[child_node_number]

    def __setitem__(self, child_node_number: int, weight: int) -> None:
        """
        This method sets the weight of the arc to the passed child node
        :param child_node_number: the number of the child node to which you need to set the cost of the path
        :param weight:  weight of the branch to the passed child node
        :return: None
        """

        self.__children_weight[child_node_number] = weight
```

```
        def sort_children(self) -> None:
            """
            This method sorts the list of child node numbers
            :return: None
            """

            self.__children = list(map(lambda x: x[0],
sorted(self.__children_weight.items(), key=lambda x: x[1])))

        def get_children(self) -> list[int]:
            """
            This method returns a sorted list of child node numbers
            :return: sorted list of child node numbers
            """

            return self.__children


    class Graph:
        """A graph class that stores graph nodes and organizes work
with itself"""

        def __init__(self, count_nodes: int) -> None:
            """
            This method initializes the initial parameters of the graph
            :param count_nodes: count of nodes in the graph
            """

            self.__count_nodes = count_nodes
            self.__nodes = [Node(self.__count_nodes) for _ in
range(count_nodes)]

        def sort(self) -> None:
            """
            This method sorts the lists of child nodes in all nodes of
the graph
            :return: None
            """

            for node in self.__nodes:
                node.sort_children()

        def __len__(self) -> int:
            """
            This method returns the number of nodes in the graph
            :return: count of nodes in the graph
            """

            return self.__count_nodes

        def __getitem__(self, node_number: int) -> Node:
            """
            This method returns a node object with the passed node
number
            :param node_number: the number of the node to get
            :return: a node object with the passed node number
            """
```

```
            return self.__nodes[node_number]


    class Solver:
        """Solver - a class solving the problem of finding a
Hamiltonian path"""

        def __init__(self, graph: Graph) -> None:
            """
            This method initializes the initial parameters of the
Solver
            :param graph: node graph
            """

            self.__graph = graph
            self.__start_node_number = 0
            self.__best_hamiltonian_cycle = list()
            self.__best_hamiltonian_cycle_weight = inf

        def __lower_bound(self, start_node_number: int, current_path:
list[int]) -> int:
            """
            This method calculates the lower bound
            (the minimum weight of the spanning tree from vertices not
included in the current path)
            :param start_node_number: the number of the node from which
the method will start
            :param current_path: the current fragment of the
Hamiltonian cycle
            :return: minimum weight of the spanning tree
            """

            min_spanning_tree_weight = 0
            visited_nodes_numbers = []
            visited_nodes_numbers_stack = [start_node_number]

            while len(visited_nodes_numbers_stack) > 0:
                best_node_index = -1
                min_weight = inf
                current_node_number = visited_nodes_numbers_stack[-1]

                for child_node_number in
self.__graph[current_node_number].get_children():
                    child_weight =
self.__graph[current_node_number][child_node_number]

                    if (child_node_number not in current_path) and
(child_weight < min_weight)\
                            and (child_node_number not in
visited_nodes_numbers)\
                            and (child_node_number not in
visited_nodes_numbers_stack):
                        min_weight = child_weight
                        best_node_index = child_node_number
                        break

                if best_node_index == -1:
```

```
visited_nodes_numbers.append(visited_nodes_numbers_stack.pop())
                else:
                    visited_nodes_numbers_stack.append(best_node_index)
                    min_spanning_tree_weight += min_weight

            return min_spanning_tree_weight

        def __find_next_node_number(self, current_node_number: int,
current_path: list[int], visited_branches: list[list[bool]]) -> int:
            """
            This method selects the best next node number
            :param current_node_number: the number of the node from
which the method will start
            :param current_path: the current fragment of the
Hamiltonian cycle
            :param visited_branches: the adjacency matrix in which at
the intersection of
            the i and j node is true if this branch has already been
checked
            :return: the best next node number
            """

            current_node = self.__graph[current_node_number]
            min_weight = inf
            best_node_number = -1

            for child_node_number in current_node.get_children():
                if (current_node[child_node_number] < min_weight) and
(child_node_number not in current_path)\
                        and (child_node_number !=
self.__start_node_number or len(current_path) == len(self.__graph) - 1):
                    if not
visited_branches[current_node_number][child_node_number]:
                        min_weight = current_node[child_node_number]
                        best_node_number = child_node_number

            if best_node_number != -1:
                visited_branches[current_node_number][best_node_number]
= True

            return best_node_number

        def __branches_and_boundaries_algorithm(self,
current_node_number: int, current_path: list[int], current_path_weight:
int, visited_branches: list[list[bool]]) -> None:
            """This method searches for the minimum Hamiltonian cycle
by the method of branches and boundaries
            :param current_node_number: the number of the node from
which the method will start
            :param current_path: list of node numbers in the current
Hamiltonian path
            :param current_path_weight: weight of the current
Hamiltonian path
            :param visited_branches: the adjacency matrix in which at
the intersection of
            the i and j node is true if this branch has already been
checked
```

```
            :return: None
            """

            if                    current_path_weight                    >
self.__best_hamiltonian_cycle_weight:
                return

            if                    current_path_weight                    +
self.__lower_bound(current_node_number,          current_path)          >
self.__best_hamiltonian_cycle_weight:
                return

            if len(current_path) == len(self.__graph):
                if                  current_path_weight                  <
self.__best_hamiltonian_cycle_weight:
                    self.__best_hamiltonian_cycle_weight                 =
current_path_weight
                    self.__best_hamiltonian_cycle = current_path.copy()
                return

            while len(current_path) != len(self.__graph):
                next_node_number                                         =
self.__find_next_node_number(current_node_number,           current_path,
visited_branches)

                if next_node_number == -1:
                    return

                new_path = current_path.copy()
                new_path_weight     =     int(current_path_weight     +
self.__graph[current_node_number][next_node_number])
                new_visited_branches = deepcopy(visited_branches)
                new_path.append(next_node_number)


self.__branches_and_boundaries_algorithm(next_node_number,      new_path,
new_path_weight, new_visited_branches)

        def find_hamiltonian_cycle(self) -> (list[int], int):
            """
            This method is a wrapper over the recursive method of
branches and bounds
            :return: Tuple from the list of cycle nodes and path
weights
            """

            visited_branches      =      [[False      for      _      in
range(len(self.__graph))] for _ in range(len(self.__graph))]
            for first_node_number in range(len(self.__graph)):
                for second_node_number in range(len(self.__graph)):
                    if
self.__graph[first_node_number][second_node_number] == inf:
                        visited_branches[first_node_number].append(True)


self.__branches_and_boundaries_algorithm(self.__start_node_number, [], 0,
visited_branches)
```

```
            self.__best_hamiltonian_cycle.insert(0,
self.__start_node_number)

            return                        (self.__best_hamiltonian_cycle,
self.__best_hamiltonian_cycle_weight)


    class App:
        """App - a class that is a wrapper over the solution of the
problem"""

        def __init__(self, file_name: str) -> None:
            """
            This method initializes the initial parameters of the App
            :param file_name: the name of the file to read the graph
            """

            self.__init_graph(file_name)
            self.__solver = Solver(self.__graph)
            self.__answer_hamiltonian_cycle = None
            self.__answer_hamiltonian_cycle_weight = None
            self.__work_time = None

        def __init_graph(self, file_name: str) -> None:
            """
            A method that implements reading a graph from an input file
            :return: None
            """

            file = open(file_name, 'r')
            size = int(file.readline())
            self.__graph = Graph(size)

            node_number = 0
            for line in file:
                formatted_line = list(map(lambda x: inf if x.strip() in
["inf", "-"] else int(x), line.split()))
                for i in range(len(formatted_line)):
                    self.__graph[node_number][i] = formatted_line[i]
                node_number += 1

            file.close()

        def __format_answer(self) -> None:
            """
            This method translates the node numbers into the desired
form and translates work time into milliseconds
            :return: None
            """

            for i in range(len(self.__answer_hamiltonian_cycle)):
                self.__answer_hamiltonian_cycle[i] += 1

            self.__work_time = int(self.__work_time * 1000000)/1000

        def solve(self) -> None:
            """
            This method solves the whole problem
```

```
            :return: None
            """

            start_time = time()
            self.__graph.sort()
            self.__answer_hamiltonian_cycle,
self.__answer_hamiltonian_cycle_weight                                    =
self.__solver.find_hamiltonian_cycle()
            self.__work_time = (time() - start_time)

        def __str__(self) -> str:
            """
            This method returns string version app answer
            :return:
            """

            if len(self.__answer_hamiltonian_cycle) != 1:
                self.__format_answer()
                return             f"{self.__answer_hamiltonian_cycle},
{self.__answer_hamiltonian_cycle_weight}, {str(self.__work_time)}mc"
            else:
                return "No Hamiltonian cycle found"

        def __repr__(self) -> str:
            """This  method  returns  official  string  version  app
answer"""

            return str(self)


    def main():
        for i in range(1, 6):
            print('\n' + "-" * 15 + f"Test {i}" + "-" * 15)
            app = App(f"test_{str(i)}.txt")
            app.solve()
            print(app)


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

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Таблица Б.1 - Примеры тестовых случаев

№ п/п	Входные данные	Выходные данные
1.	4 inf 1 2 2 - inf 1 2 - 1 inf 1 1 1 - inf	----------Test 1---------- [1, 2, 3, 4, 1], 4, 0.435mc
2.	3 inf 1 1 - inf - 1 - inf	----------Test 2---------- No Hamiltonian cycle found
3.	3 inf 60 1 5 inf 2 1 5 inf	----------Test 4---------- [1, 2, 3, 4, 6, 5, 1], 16, 1.919mc
4.	20 inf 5 65 2 18 26 21 67 48 17 91 74 78 21 35 26 85 87 21 43 35 inf 88 63 24 43 46 75 5 22 3 27 87 55 50 25 65 14 10 68 3 68 inf 24 44 28 19 17 13 66 43 93 38 63 42 34 58 6 91 36 12 50 75 inf 87 62 89 21 60 41 45 89 68 35 32 9 16 88 23 75 84 19 89 90 inf 93 69 52 71 3 62 62 23 71 77 93 68 24 20 38 17 77 48 19 70 inf 22 43 5 63 43 78 10 25 91 8 89 79 35 50 8 29 93 19 94 15 inf 20 60 79 43 82 5 7 61 60 49 30 25 15 7 1 76 60 64 20 1 inf 12 4 42 15 75 100 34	----------Test 5---------- [1, 5, 10, 17, 16, 6, 9, 15, 13, 18, 4, 8, 2, 11, 7, 14, 20, 19, 12, 3, 1], 163, 336.292mc

	71 9 35 69 79 7 41 90 38 88 68 22 49 inf 91 87 50 58 81 6 47 48 6 100 78 21 20 72 97 90 22 30 78 50 inf 22 47 26 71 72 59 11 100 30 41 15 60 98 97 34 45 7 55 1 47 inf 8 47 38 35 97 15 53 61 95 64 51 21 64 55 92 64 41 68 66 56 inf 70 25 77 84 55 87 82 48 95 23 49 54 88 34 60 97 18 76 43 40 inf 54 46 22 77 1 84 42 50 63 93 4 73 53 79 66 73 17 95 10 29 inf 1 27 71 11 85 5 69 80 81 11 76 68 83 28 67 16 45 74 1 84 inf 74 81 100 15 26 20 54 97 47 16 8 56 80 42 84 20 83 76 62 61 inf 84 30 74 64 27 12 61 96 41 46 12 83 96 37 34 100 46 53 36 11 inf 13 87 49 94 70 50 4 75 58 96 60 24 9 100 76 10 61 16 98 30 inf 25 4 63 85 47 77 49 32 4 29 16 50 82 11 76 71 33 92 70 8 inf 6 91 29 72 5 36 43 55 22 95 63 87 52 33 40 5 60 2 41 16	