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

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Построение и анализ алгоритмов»
Тема: Задача коммивояжера

Студентка гр. 1304 ___ Чернякова А.Д.
Преподаватель ___ Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Изучить задачу коммивояжера по построению минимального по весу ребер гамильтонового пути, реализовать задачу методом ветвей и границ.

Задача.

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):

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

Выполнение работы.

Для решения задачи Коммивояжера реализован метод ветвей и границ.

Создан класс *Matrix*, который содержит функции:

__init__(self, matrix, local_border, not_branch_array, parent_matrix, start_city, finish_city, map) - данный метод принимает на вход двумерный массив расстояний между городами matrix, локальную нижнюю границу матрицы *local_border*, список еще не ветвящихся матриц, родительскую матрицу *parrent_matrix* (матрица, от которой ветвится текущая), начальный город *start_city*, который соединяется с городом *finish_city*, и изначальную карту/матрицу соединения всех городов, для подсчета итогового пути. Соответственно инициализируются поля *self.matrix, self.local_border, self.not_branch_array, self.parent_matrix, self.start_city, self.finish_city, self.map*. Также инициализируется поле *self.inf = 20000* для

обозначения бесконечно длинного пути (так как путь из одного города в другой не превышает *1000*), *self.width* и *selth.height* - ширина и высота матрицы, рассчитываются исходя из поля *self.matrix*.

replace_no_way(self) - метод ничего не принимает на вход и ничего не возвращает, заменяет в *self.matrix* все невозможные пути *'-'* и пути-бесконечности *'inf'* на значение *self.inf*, чтобы было удобно работать с матрицей. Также инициализируется поле *self.map* - является копией *self.matrix*. Данная функция вызывается один раз для первоначально считанной карты/матрицы из файла.

row_reduction(self) - метод редукции строк, который ничего не принимает на вход. Метод находит минимальное значение в каждой строке матрицы - константы приведения для строк, затем производит редукцию строк: из каждого элемента в каждой строке вычитает соответствующее ей значение минимума и возвращает список констант приведения для строк.

column_reduction(self) - метод редукции столбцов, который ничего не принимает на вход. Метод находит минимальное значение в каждом столбце матрицы, затем производит редукцию столбцов: из каждого элемента в каждом столбце вычитает соответствующее ему значение минимума и возвращает список констант приведения для столбцов.

change_boarder(self) - метод ничего не принимает и ничего не возвращает, рассчитывает и изменяет значение локальной нижней границы *self.local_border*, прибавляя к предыдущему значению сумму констант приведения для строк и сумму констант приведения для столбцов.

get_boarder(self) - метод ничего не принимает и возвращает значение нижней локальной границы - поле *self.local_border*.

null_evaluate(self, w, h) - метод вычисления оценок нулевых клеток, принимает на вход координаты нулевой клетки - на пересечении каких городов она находится в матрице. Для нулевой клетки преобразованной матрицы рассчитывается «оценка». Ею будет сумма минимума по строке и минимума по

столбцу, на пересечении которых находится данная клетка с нулем. При этом сама нулевая клетка для которой вычисляется оценка не учитывается. Метод возвращает оценку.

if_equale_null_evaluate(self, null_row_array) - метод принимает на вход массив элементов (вид элемента: [оценка нулевой клетки, координата x нулевой клетки, координата y нулевой клетки]). Метод необходим в случае когда все нулевые клетки одной строки имеют одинаковую оценку. В некоторых случаях возможен выбор любой клетки, в некоторых необходимо выбирать определенную. Данный метод реализует правильный выбор и возвращает новый список *new_null_row_array,* если *null_row_array* был изменен, в противном случае возвращается список, который и был принят на вход.

max_null_evaluate(self) - метод ничего не принимает на вход, для каждой нулевой клетки вызывает метод *self.null_evaluate(self, w, h)* для расчета оценки и *self.if_equale_null_evaluate(null_row_array)*, вычисляет нулевую клетку с максимальной оценкой и возвращает список вида [максимальная оценка нулевой клетки, координата *x* нулевой клетки с максимальной оценкой, координата *y* нулевой клетки с максимальной оценкой]

get_parent_matrix(self) - метод ничего не принимает на вход и возвращает значения поля *self.parent_matrix*

is_positive_city(self) - метод ничего не принимает на вход, возвращает *True*, если *self.start_city >= 0*. При каждом ветвлении матрица делится на две ветви: ветвь решения, где мы включаем в маршрут выбранный отрезок пути *self.start_city - self.finish_city*, и ветвь решения, где не включаем. Для второй ветви *self.start_city* и *self.finish_city* обозначаются с противоположным знаком для удобства в решении, так как ветвление матрицы в двух разных случаях происходит по-разному.

get_evaluate_position(self) - метод ничего не принимает, возвращает *[self.start_city, self.finish_city]*

get_cost(self, start_city, finish_city) - метод принимает на вход город из

которого идет путь и в какой город идет и возвращает стоимость пути в первоначальной считанной карте, которую хранит каждая матрица.

do_sequence(self, edges) - метод принимает на вход список ребер *edges* для построения гамильтонова пути и возвращает путь.

everything_is_inf(self) - метод ничего не принимает на вход, проверяет все ли элементы кроме одного в матрице заполнены нулями: если да, то возвращает *True,* в противном случае - *False.*

solve_(self) - главный метод (ничего не принимает и ничего не возвращает), который вызывается считанной первоначальной матрицей из *main.* Переменной *start_time* присваивается время начала алгоритма, вызываются методы *self.replace_no_way()* для преобразования матрицы в рабочий вид, *self.change_boarder()* для расчета первоначальной минимальной локальной границы, *self.solve()* для рекурсивной реализации задачи, переменной *end_time* присваивается время окончания алгоритма, рассчитывается время работы алгоритма и выводится в консоль в миллисекундах.

solve(self) - рекурсивный метод, отвечающий за логику реализации метода ветвей и границ. Метод рассчитывает оценки нулевых клеток, сравнивает локальные границы матриц, контролирует массив еще не ветвившихся матриц, создает ветви решения, которые рекурсивно вызывают *solve()*. Условие выхода из рекурсии - функция *self.everything_is_inf()* вернет *True*. Тогда рассчитывается длина гамильтонова и сам путь, они же выводятся в консоль. Если путь не найден, то выводится сообщение об его отсутствии.

Вне класса *Matrix* реализована функция *inputs()*, которая считывает матрицу из текстового файла *test.txt*.

При запуске программы main.py, происходит считывание из файла и результат присваивается переменной *start_map*, создается объект *matrix = Matrix(start_map, 0, [], None, -1, -1, None)* и вызывается метод *solve_()* для данного объекта.

Исходный код программы представлен в приложении А

Тестирование программы представлено в таблице 1 в приложении Б

Выводы.

Изучена задача коммивояжера по построению в графе минимального гамильтонова цикла. Для решении задачи реализован метод ветвей и границ: на каждом этапе после нахождения нулевой клетки с максимальной оценкой(ее координаты - *start_city, finish_city*), текущая матрица ветвится на две: ветвь решения, где мы включаем в маршрут выбранный отрезок пути *self.start_city - self.finish_city*, и ветвь решения, где не включаем. Для следующего ветвления выбирается матрица, которая еще не ветвилась и которая имеет минимальную нижнюю локальную границу. Так как матрица хранит матрицу-родителя, то после окончания алгоритма, последовательность восстанавливается снизу вверх по дереву ветвления. В случае невозможности построения гамильтонового цикла программа выводит сообщение об его отсутствии.

Программа прошла все тесты из приложения Б

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: *main.py*

```
from copy import deepcopy
import time

class Matrix:
    def __init__(self, matrix, local_border, not_branch_array,
parent_matrix, start_city, finish_city, map):
        self.inf = 20000
        self.matrix = matrix
        self.width = len(matrix)
        self.height = len(matrix[0])
        self.local_border = local_border
        self.not_branch_array = not_branch_array
        self.parent_matrix = parent_matrix
        self.start_city = start_city
        self.finish_city = finish_city
        self.map = map

    def replace_no_way(self):
        for i in range(self.width):
            for j in range(self.height):
                if self.matrix[i][j] == '-' or self.matrix[i][j] == 'inf':
                    self.matrix[i][j] = self.inf
                else:
                    self.matrix[i][j] = int(self.matrix[i][j])
        self.map = deepcopy(self.matrix)

    def row_reduction(self):
        arr_min_row_ways = []
        for i in range(self.width):
            inf = [self.inf]
            row_ways_without_inf = [x for x in self.matrix[i] if x not in
inf]
            if row_ways_without_inf:
                min_way = min(row_ways_without_inf)
            else:
                min_way = 0
            arr_min_row_ways.append(min_way)
            for j in range(self.height):
                if self.matrix[i][j] != self.inf:
                    self.matrix[i][j] = self.matrix[i][j] - min_way
        return arr_min_row_ways

    def column_reduction(self):
        arr_min_column_ways = []
        for i in range(self.height):
            min_way = self.inf
            for j in range(self.width):
                if self.matrix[j][i] < min_way:
                    min_way = self.matrix[j][i]
            if min_way == self.inf:
```

```
                min_way = 0
            arr_min_column_ways.append(min_way)
            for j in range(self.width):
                if self.matrix[j][i] != self.inf:
                    self.matrix[j][i] = self.matrix[j][i] - min_way
        return arr_min_column_ways

    def change_boarder(self):
        self.local_border += sum(self.row_reduction()) +
sum(self.column_reduction())

    def get_boarder(self):
        return self.local_border

    def null_evaluate(self, w, h):
        min_way_row = self.inf
        for i in range(self.height):
            if i == h:
                continue
            if self.matrix[w][i] < min_way_row:
                min_way_row = self.matrix[w][i]
        min_way_column = self.inf
        for i in range(self.width):
            if i == w:
                continue
            if self.matrix[i][h] < min_way_column:
                min_way_column = self.matrix[i][h]
        return min_way_column + min_way_row

    def if_equale_null_evaluate(self, null_row_array):
        array = [x[0] for x in null_row_array]
        if len(array) > 1 and array and array.count(null_row_array[0][0])
== len(null_row_array):
            inf_rows = 0
            for i in range(self.width):
                if self.matrix[i][0] >= self.inf:
                    inf_rows += 1
            if self.width - inf_rows > 1:
                new_null_row_array = [x for x in null_row_array if x[2] !=
0]
                return new_null_row_array
        return null_row_array

    def max_null_evaluate(self):
        null_evaluate_array = []
        for i in range(self.width):
            null_row_array = []
            for j in range(self.height):
                if self.matrix[i][j] == 0:
                    null_evaluate = self.null_evaluate(i, j)
                    null_row_array.append([null_evaluate, i, j])

null_evaluate_array.extend(self.if_equale_null_evaluate(null_row_array))
        max_null_evaluate, max_w_evaluate, max_h_evaluate = -1, 0, 0
        for i in range(len(null_evaluate_array)):
            if null_evaluate_array[i][0] > max_null_evaluate:
                max_null_evaluate, max_w_evaluate, max_h_evaluate =
null_evaluate_array[i][0], null_evaluate_array[i][1],
```

```
null_evaluate_array[i][2]
        return [max_null_evaluate, max_w_evaluate, max_h_evaluate]

    def get_parent_matrix(self):
        return self.parent_matrix

    def is_positive_city(self):
        return True if self.start_city >= 0 else False

    def get_evaluate_position(self):
        return [self.start_city, self.finish_city]

    def get_cost(self, start_city, finish_city):
        return self.map[start_city][finish_city]

    def do_sequence(self, edges):
        unique_vertices = list(set([v for e in edges for v in e]))
        adjacency_list = {v: [] for v in unique_vertices}
        for edge in edges:
            adjacency_list[edge[0]].append(edge[1])
        start_vertex = unique_vertices[0]
        path = [start_vertex]
        visited = {start_vertex}
        while len(path) < len(unique_vertices):
            current_vertex = path[-1]
            for neighbor in adjacency_list[current_vertex]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    path.append(neighbor)
                    break
        path.append(start_vertex)
        path = [x+1 for x in path]
        return path

    def everything_is_inf(self):
        flag = 0
        for i in range(self.width):
            for j in range(self.height):
                if self.matrix[i][j] < self.inf:
                    flag += 1
                    if flag > 1:
                        return False
        return True

    def solve(self):
        evaluate, start_city, finish_city = self.max_null_evaluate()
        if self.everything_is_inf():
            length = 0
            result = [[start_city, finish_city],
self.get_evaluate_position()]
            length += self.get_cost(start_city, finish_city)
            length += self.get_cost(self.get_evaluate_position()[0],
self.get_evaluate_position()[1])
            current_matrix = self.parent_matrix
            while current_matrix:
                if current_matrix.is_positive_city():
                    result.append(current_matrix.get_evaluate_position())
                    length +=
```

```
current_matrix.get_cost(current_matrix.get_evaluate_position()[0],
current_matrix.get_evaluate_position()[1])
                current_matrix = current_matrix.get_parent_matrix()
            if length >= self.inf or len(result) < len(self.map):
                print("Гамильтонова цикла не существует", end=', ')
                return
            print(self.do_sequence(result), end=', ')
            print(length, end=', ')
            return

        matrix_include_way = deepcopy(self.matrix)
        matrix_include_way[finish_city][start_city] = self.inf
        for i in range(len(matrix_include_way)):
            matrix_include_way[i][finish_city] = self.inf
        for i in range(len(matrix_include_way[start_city])):
            matrix_include_way[start_city][i] = self.inf
        positive_matrix = Matrix(matrix_include_way, self.local_border,
self.not_branch_array, self, start_city, finish_city, self.map)
        positive_matrix.change_boarder()
        local_border_positive = positive_matrix.get_boarder()
        local_border_negative = self.local_border + evaluate

        matrix_not_include_way = deepcopy(self.matrix)
        matrix_not_include_way[start_city][finish_city] = self.inf
        negative_matrix = Matrix(matrix_not_include_way, self.local_border,
self.not_branch_array, self, -start_city, -finish_city, self.map)
        negative_matrix.change_boarder()

        self.not_branch_array.append([local_border_positive, start_city,
finish_city, positive_matrix])
        self.not_branch_array.append([local_border_negative, -start_city,
-finish_city, negative_matrix])
        min_not_branch_elem = self.not_branch_array[0]
        for not_branch_elem in self.not_branch_array:
            if not_branch_elem[0] < min_not_branch_elem[0]:
                min_not_branch_elem = not_branch_elem
        if min_not_branch_elem[1] >= 0:
            self.not_branch_array.remove(min_not_branch_elem)
            min_not_branch_elem[3].solve()
        else:
            self.not_branch_array.remove(min_not_branch_elem)
            min_not_branch_elem[3].solve()

    def solve_(self):
        start_time = time.time()
        self.replace_no_way()
        self.change_boarder()
        self.solve()
        end_time = time.time()
        time_ms = (end_time - start_time) * 1000
        print(f"{time_ms:.0f}мс")


def inputs():
    map = []
    with open('test.txt') as file:
        for line in file:
            map.append(line.strip().split())
```

```
    return map


if __name__ == '__main__':
    start_map = inputs()
    matrix = Matrix(start_map, 0, [], None, -1, -1, None)
    matrix.solve_()
```

ПРИЛОЖЕНИЕ Б.

ТЕСТИРОВАНИЕ ПРОГРАММЫ.

Таблица 1 - Результаты тестирования

№ теста	Входные данные	Результат	Комментарий
1	inf 1 2 2 - inf 1 2 - 1 inf 1 1 1 - inf	[1, 2, 3, 4, 1], 4, 1мс	Тест условия задачи
2	inf 2 2 2 2 2 2 inf 2 2 2 2 2 2 inf 2 2 2 2 2 2 inf 2 2 2 2 2 2 inf 2 2 2 2 2 1 inf	[1, 2, 3, 4, 6, 5, 1], 11, 4мс	Полный граф 6*6, в котором один путь короче всех остальных
3	inf 1 - 1 1 - - inf 1 - - - - 1 inf 1 1 - - - 1 inf 1 - - - - 1 inf 1 - 1 - - 1 inf	Гамильтонова цикла не существует, 1мс	Граф, в котором нет гамильтонового цикла
4	граф 20x20 (представлен в приложении С)	[1, 7, 14, 15, 13, 18, 20, 19, 12, 3, 8, 6, 4, 16, 5, 10, 17, 2, 11, 9, 1], 201, 27мс	Граф большого размера

ПРИЛОЖЕНИЕ С.

МАТРИЦА/ГРАФ 20x20.

inf - 65 - 18 - 21 67 48 17 91 74 - 21 35 - 85 87 21 43

35 inf 88 - 24 43 46 75 - - 3 27 87 55 50 - 65 - 10 68

3 68 inf 24 44 28 19 17 13 66 43 93 38 - 42 34 58 - 91 36

12 50 75 inf 87 62 89 21 - 41 45 89 68 35 32 9 16 88 23 75

84 19 89 90 inf 93 69 52 - 3 62 62 23 - 77 93 68 24 20 38

17 77 48 19 70 inf - 43 - - 43 - 10 - 91 - 89 79 35 50

- - 93 19 94 - inf 20 - 79 43 82 - 7 61 - 49 - - -

7 1 76 - 64 20 1 inf 12 4 42 - 75 - 34 - 9 35 69 79

7 41 90 38 88 68 - 49 inf 91 87 50 58 81 - 47 48 - - -

21 20 72 97 90 - - - 50 inf - 47 - - 72 59 11 - - 41

- - 98 97 34 45 7 55 1 47 inf - 47 38 35 97 - 53 61 95

64 51 21 64 55 92 64 41 68 66 56 inf 70 - 77 84 55 87 82 48

95 23 49 54 88 34 - 97 18 76 43 40 inf 54 46 - 77 1 84 42

50 - 93 4 73 53 79 66 73 17 95 10 - inf 1 27 - 11 85 -

69 80 81 11 76 68 83 28 67 16 45 74 1 84 inf 74 81 - - -

20 54 97 47 16 - 56 80 42 84 20 83 76 62 61 inf 84 - 74 64

27 12 61 96 41 46 12 83 96 37 34 - 46 53 36 11 inf 13 87 49

94 70 50 4 75 58 96 - 24 9 - 76 10 61 16 98 - inf - 4

- 85 47 77 49 32 4 - 16 50 82 11 76 - - 92 70 - inf -

91 - 72 - 36 43 55 - 95 - 87 52 - 40 - - - 41 16 inf