# 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 *mij* (переезд из города *i* в город *j*) можно сопоставить критерий выгодности маршрута (вес ребра) равный *wi* (натуральное число [1, 1000]), *mij=inf*, если *i=j*.

Если маршрут включает в себя ребро *mij*, то *xij=1*, иначе *xij=0.*

Требуется найти минимальный маршрут (минимальный гамильтонов цикл):



# 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 C) | [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 C.

МАТРИЦА/ГРАФ 20х20.

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