**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: Алгоритм Кнута-Морриса-Пратта

|  |  |  |
| --- | --- | --- |
| Студент гр. 1384 |  | Усачева Д.В. |
| Преподаватель |  | Шевелева А. М. |

Санкт-Петербург

2023

## Цель работы.

Изучить возможные реализации построения минимального гамильтонова цикла – решения задачи о коммивояжере, а также их оптимизации.

## 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]), 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):

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 20х20 не дольше 3 минут в среднем.

## Выполнение работы.

Данный код решает задачу коммивояжера при помощи алгоритма Беллмана-Хелда-Карпа.

Алгоритм Беллмана-Хелда-Карпа - это алгоритм поиска кратчайшего пути в графе, позволяющий обойти заданные вершины по определённому маршруту единожды. Суть алгоритма заключается в последовательном рассмотрении всех возможных маршрутов, проходящих через каждую из вершин в заданном порядке.

Класс TSP (Traveling Salesman Problem) содержит функции:

* init – конструктор класса для инициализации переменных
* input\_graph – метод для ввода графа из файла и создания

начальных параметров для min\_way\_table, set\_of\_values, key\_addition

* min\_set – метод для поиска минимального пути из подмножеств

к вершинам

* find\_min\_hamiltonian\_path – метод для поиска минимального

гамильтонова пути

Он принимает на вход файл с графом, который описывает матрицу расстояний между каждой парой вершин в графе. Файл должен содержать матрицу в виде квадратной таблицы, где каждый элемент матрицы - это расстояние между соответствующими вершинами.

Далее происходит заполнение массива matrix\_graph из графа файла, если между вершинами нет прямого пути, то записывается значение `math.inf` – бесконечность.

Затем создаются начальные значения для min\_path\_table, set\_of\_values, и key\_addition. Значения min\_path\_table — это словарь, который хранит минимальные пути для текущего шага, где ключ — это множество, из которого ведется поиск минимального пути к вершинам. set\_of\_values представляет собой множество ключей графа для текущего шага. key\_addition — массив дополнений для множества ключей.

Для поиска минимального пути из подмножества к вершинам используется метод min\_set. В ней подмножество разбивается на подмножества меньшей длины и дополняющие их вершины. Затем выбирается минимальный путь к каждой из целевых вершин из всех разбитых на части подмножеств с помощью таблицы из предыдущего шага.

Для поиска минимального гамильтонова цикла используется метод find\_min\_hamiltonian\_path. Его поиск разделен на шаги, номер шага соответствует длине рассматриваемого на этом шаге множества. Сначала мы сохраняем данные таблицы путей и массива ее ключей из предыдущего шага, далее заполняем значения массива ключей для текущего шага. После этого идет заполнение таблицы для текущего шага при помощи функции min\_set. На каждом шаге хранится информация только о текущей и предыдущей итерации, остальные данные не являются нужными, поэтому удаляются. Также в методе рассматриваются случаи матрицы размера 1 и отсутствия прямого гамильтонова цикла.

В итоге, получается минимальная замкнутая гамильтонова цепь в графе, проходящая через каждую вершину только один раз.

## Тестирование.

Результаты тестирования представлены на рисунках 1 — 5.

1. Проверка работы программы на произвольном полном графе на 5-и

вершинах (см. рисунок 1)

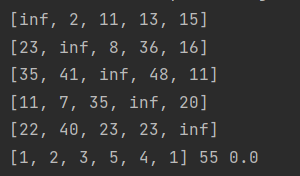


Рисунок 1 – Тестирование матрицы 5x5

1. Проверка работы программы на произвольном полном графе на 20-и

вершинах (см. рисунок 2)

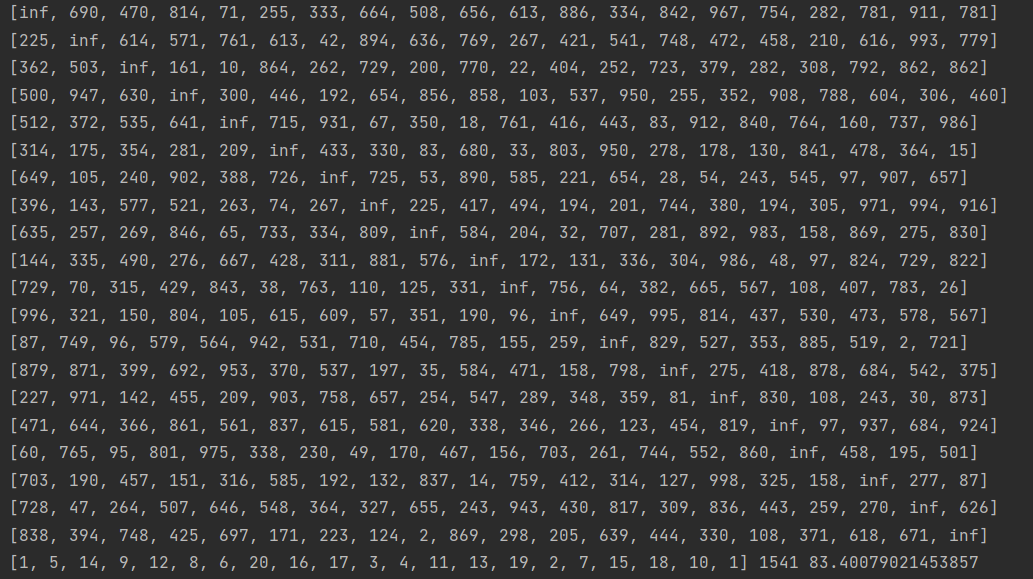


Рисунок 2 – Тестирование матрицы 20x20

1. Проверка работы программы на произвольном полном графе на 20-и

вершинах (см. рисунок 3)

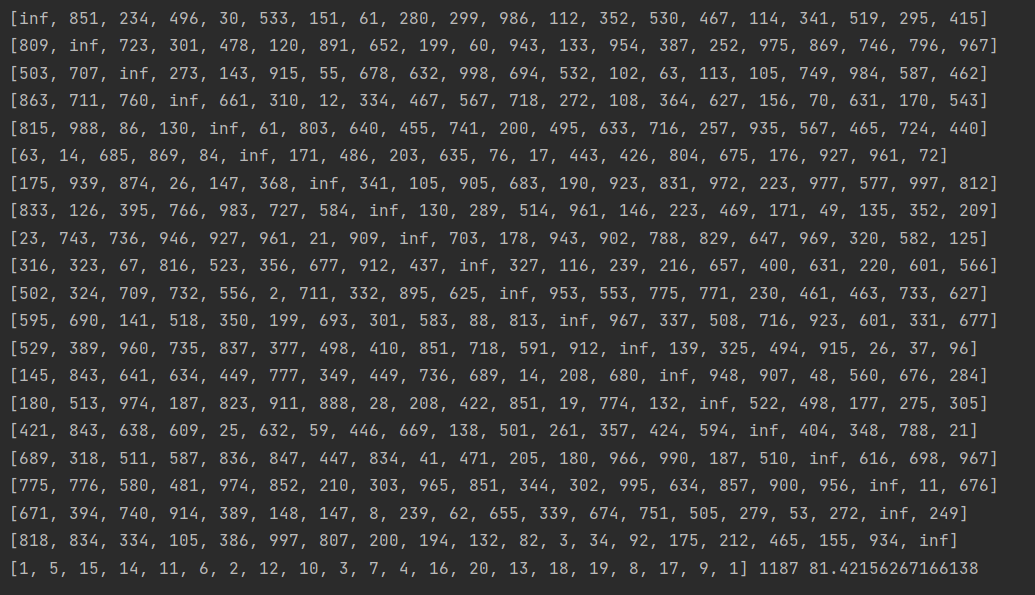


Рисунок 3 – Тестирование матрицы 20x20

1. Проверка работы программы на слабо связном графе без

гамильтонова цикла (см. рисунок 4).

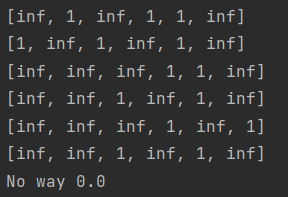


Рисунок 4 – Тестирование на слабо связном графе без гамильтонова цикла

1. Проверка работы программы на графе с одинаковым весом всех,

кроме одного, ребер (см. рисунок 5).

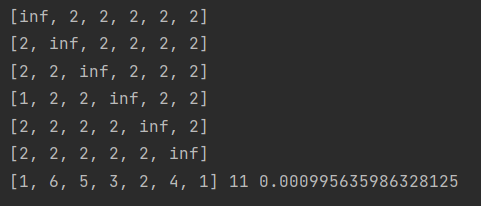


Рисунок 5 – Тестирование матрицы с одинаковым весом ребер, кроме одного

## Выводы.

В рамках данной лабораторной работы был изучен алгоритм, который решает задачу коммивояжера - поиск минимального гамильтонова цикла в графе. Полный перебор работает очень долго, метод ветвей и границ не стабилен, поэтому в качестве алгоритма для решения задачи коммивояжера был выбран алгоритм алгоритма Беллмана-Хелда-Карпа.

В качестве решения поставленной задачи была написана программа, которая с помощью алгоритма Беллмана-Хелда-Карпа решает задачу коммивояжера для графов размера 20 на 20 менее, чем за две минуты.

Были рассмотрены особые условия (граф несвязен, все ребра кроме 1 равны по весу и т.д.), и при всех таких условиях программа работала корректно.

## ПРИЛОЖЕНИЕ А

## ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл: lb3.py

import math

import time

class TSP:

'''

Коструктор класса

matrix\_graph - граф, введенный из файла

key\_addition - массив дополнений для множества ключей, состоит из чисел от 2 до count (тип значений - tuple)

min\_path\_table - словарь, для хранения минимальных путей для текущего шага, где ключ это множество, из которого

ведется поиск минимального пути к вершинам, значение - массив кортежей длины 3 (вершина

в которую идем, минимальный путь к этой вершине, длина этого пути)

previous\_min\_path\_table - словарь, для хранения таблицы для пердыдущего шага

set\_of\_values - множество ключей графа, для текущего шага

count - количество строк/столбцов графа

'''

def \_\_init\_\_(self):

self.matrix\_graph = []

self.key\_addition = []

self.min\_path\_table = dict()

self.previous\_min\_path\_table = dict()

self.set\_of\_values = []

self.count = 0

'''

Функция для ввода графика из файла, также в ней задаются начальные парметры для

min\_way\_table, set\_of\_values, key\_addition

'''

def input\_graph(self, file\_name):

file = open(file\_name)

for line in file:

line\_matrix = []

for i in line.split():

if i == '-1' or i == 'inf' or i == '-' or int(i) >= 100000:

line\_matrix.append(math.inf)

else:

line\_matrix.append(int(i))

self.matrix\_graph.append(line\_matrix)

file.close()

self.count = len(self.matrix\_graph[0])

for i in range(2, self.count + 1):

self.min\_path\_table[tuple([i])] = []

for j in range(2, self.count + 1):

if i != j:

self.min\_path\_table[tuple([i])].append(

(j, [1, i], self.matrix\_graph[0][i - 1] + self.matrix\_graph[i - 1][j - 1]))

self.set\_of\_values.append(tuple([i]))

self.key\_addition = self.set\_of\_values.copy()

'''

Функия для поиска минимального пути из подмножества к вершинам

'''

def min\_set(self, set):

subsets = []

complementary\_vertex = []

final\_vertex = []

min\_path\_set = []

set = list(set)

for i in range(2, self.count + 1):

if i in set:

complementary\_vertex.append(i)

new\_set = set.copy()

new\_set.remove(i)

subsets.append(tuple(new\_set))

else:

final\_vertex.append(i)

if not final\_vertex:

final\_vertex.append(1)

for v in final\_vertex:

path\_length = []

path = []

for i in range(len(subsets)):

for j in self.previous\_min\_path\_table[subsets[i]]:

if j[0] == complementary\_vertex[i]:

path\_length.append(j[2] + self.matrix\_graph[complementary\_vertex[i] - 1][v - 1])

path.append(j)

best\_id = path\_length.index(min(path\_length))

min\_way = path[best\_id][1].copy()

min\_way.append(path[best\_id][0])

min\_path\_set.append((v, min\_way, min(path\_length)))

return min\_path\_set

'''

Фунция для поиска минимального гамильтонова пути

'''

def find\_min\_hamiltonian\_path(self):

if self.count == 1:

print("[1, 1]", self.matrix\_graph[0][0])

return 0

start = time.time()

for step in range(2, self.count):

previous\_set\_of\_values = self.set\_of\_values.copy()

self.previous\_min\_path\_table = self.min\_path\_table.copy()

self.min\_path\_table.clear()

self.set\_of\_values.clear()

for prev\_set in previous\_set\_of\_values:

for key\_addition in self.key\_addition:

current\_set = set(prev\_set + key\_addition)

current\_set = list(current\_set)

current\_set.sort()

current\_set = tuple(current\_set)

if len(current\_set) == step:

self.set\_of\_values.append(current\_set)

self.set\_of\_values = set(self.set\_of\_values)

self.set\_of\_values = list(self.set\_of\_values)

for i in self.set\_of\_values:

self.min\_path\_table[i] = self.min\_set(i)

end = time.time() - start

if self.min\_path\_table[self.set\_of\_values[0]][0][2] == math.inf:

print("No way", end)

else:

self.min\_path\_table[self.set\_of\_values[0]][0][1].append(1)

print(self.min\_path\_table[self.set\_of\_values[0]][0][1], self.min\_path\_table[self.set\_of\_values[0]][0][2],

end)

ts = TSP()

ts.input\_graph('test1.txt')

ts.find\_min\_hamiltonian\_path()