МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский технический университет связи и информатики»

Кафедра

«Математическая кибернетика и информационные технологии»

Лабораторная работа №5

«Рекурсия. Фракталы.»

по дисциплине

«Структуры и алгоритмы обработки данных»

Выполнил:

студент гр. БСТ2004

Шадюк М.Р.

Вариант №22

Москва, 2022 г.

Оглавление

[Задание 3](#_Toc102110031)

[Код 3](#_Toc102110032)

[Выполнение 4](#_Toc102110033)

# Задание

1.  Реализовать алгоритм поиска кратчайшего расстояния между двумя вершинами ориентированного взвешенного графа в соответствии с вариантом.

2.  Предусмотреть задание графа в виде матрицы смежности/инцидентности, читаемой из файла, либо графически с помощью пользовательского интерфейса.

3.  Разработать графический интерфейс пользователя с визуализацией графа и отображением кратчайшего расстояния между задаваемыми пользователем вершинами.

4. По результатам работы проанализировать временную сложность работы заданного алгоритма в зависимости от числа узлов и ребер графа.

Данные представить в виде таблицы.

# Выполнение

## Код

# Алгоритм Джонсона

import math

import copy

import networkx as nx

import matplotlib.pyplot as plt

import timeit

def matric(rez,alf,path):#создание списка кортеджей длин наименьшего пути

    path\_m = []

    for i in range(len(rez)-1):

        for ii in path:

            if ii[0] == alf[rez[i]].upper() and ii[1] == alf[rez[i+1]].upper():

                path\_m.append((alf[rez[i]].upper(),alf[rez[i+1]].upper(),ii[2]))

    return path\_m

def graf(path,rezMatrix):# отрисовка графа

    G = nx.DiGraph()

    M = nx.DiGraph()

    E = path

    Er=rezMatrix

    G.add\_weighted\_edges\_from(E)

    M.add\_weighted\_edges\_from(Er)

    pos=nx.spring\_layout(G)

    nx.draw(G, pos, with\_labels=True, font\_weight='bold',font\_color='white',node\_color='black')

    nx.draw(M, pos, with\_labels=True, font\_weight='bold',font\_color='white',node\_color='red',edge\_color='red')

    edge\_weight = nx.get\_edge\_attributes(G,'weight')

    nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels = edge\_weight)

    plt.show()

def mat(z,alf):#создание списка кортеджей которая всебе хранит растоянния от вершин

    v = []

    for k in z:

        v.append(k.split())

    n = len(v)

    path = []

    for i in range(n):

        for ii in range(n):

            if v[i][ii] =="0" or v[i][ii] == '-':

                pass

            else:

                k = i

                kk = ii

                while k>=len(alf):

                    k-=len(alf)

                while kk>=len(alf):

                    kk-=len(alf)

                path.append((alf[k].upper(),alf[kk].upper(),v[i][ii]))

    return path

def read\_file(f,alf):

    V = []

    z = []

    for i in f:

        z.append(i)

    path = mat(z,alf)

    for i in z:

        k = i.split()

        o = []

        for ii in k:

            if ii == "-":

                o.append(math.inf)

            else:

                o.append(int(ii))

        V.append(o)

        o = []

    return V,path

def d\_get\_path(path, start, end):

    result = [end]

    while end != start:

        end = path[end]

        result.append(end)

    return result[::-1]

def dijkstra\_path(matrix, start, end):

    dist = [matrix[start][i] for i in range(len(matrix))]

    prev = [start for i in range(len(matrix))]

    checked\_nodes = set()

    checked\_nodes.add(start)

    for i in range(len(matrix)):

        node = 0

        for i in range(len(matrix)):

            if dist[i] < math.inf and i not in checked\_nodes:

                node = i

        if node:

            checked\_nodes.add(node)

            for i in range(len(matrix)):

                if dist[i] > matrix[node][i] + dist[node]:

                    dist[i] = matrix[node][i] + dist[node]

                    prev[i] = node

    if dist[end] == math.inf:

        return [], dist

    else:

        return d\_get\_path(prev, start, end), dist

def bellman\_ford\_path(matrix, start, end,n):

    edges = []

    for i in range(n):

        for j in range(n):

            if i != j and matrix[i][j] < math.inf:

                edges.append((i,j,matrix[i][j]))

    from\_pos = 0

    to\_pos = 1

    edge\_length = 2

    d = [math.inf] \* n

    d[start] = 0

    p = [-1] \* n

    while True:

        flag = False

        for j in range(len(edges)):

            if d[edges[j][from\_pos]] < math.inf:

                if d[edges[j][to\_pos]] > d[edges[j][from\_pos]] + edges[j][edge\_length]:

                    d[edges[j][to\_pos]] = d[edges[j][from\_pos]] + edges[j][edge\_length]

                    p[edges[j][to\_pos]] =  edges[j][from\_pos]

                    flag = True

        if not flag:

            break

    if d[end] == math.inf:

        return [], d

    else:

        path = []

        cur = end

        while cur != -1:

            path.append(cur)

            cur=p[cur]

        return path[::-1], d

def johnsons\_path(matrix, start, end,n):

    bf\_path, bf\_dist = bellman\_ford\_path(copy.deepcopy(matrix), start, end,n)

    if bf\_dist == []:

        return []

    for i in range(n):

        for j in range(n):

            if matrix[i][j] != math.inf and bf\_dist[i] != math.inf and bf\_dist[j] != math.inf:

                matrix[i][j] += bf\_dist[i] - bf\_dist[j]

    paths = []

    dist = []

    for i in range(n):

        p, d = dijkstra\_path(matrix, i, end)

        paths.append(p)

        dist.append(d)

    for i in range(n):

        for j in range(n):

            if dist[i][j] != math.inf and bf\_dist[i] != math.inf and bf\_dist[j] != math.inf:

                dist[i][j] +=  bf\_dist[j] - bf\_dist[i]

    return paths[start]

alf = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

f = open("mas.txt", "r",encoding='utf-8')

V, path = read\_file(f,alf)

f.close()

n = len(V)#число вершин в графе

s = "A"

e = "H"

put=""

start = alf.index(s.lower())# начальная вершина

end = alf.index(e.lower())#конечная вершина

start\_time = timeit.default\_timer()

rez= johnsons\_path(V, start, end,n)

tmJ = timeit.default\_timer() - start\_time

#time\_matrix\_Jon = "{0} ms".format(round((time.time() - start\_time)\*1000))

print(f'посторение пути из точки {alf[start].upper()} в точку {alf[end].upper()}:')

for i in rez:

    put += alf[i].upper()+"->"

print(put[:-2])

rezMatrix = matric(rez,alf,path)

graf(path,rezMatrix)

f = open("mas\_100.txt", "r",encoding='utf-8')

V, path = read\_file(f,alf)

f.close()

n = len(V)#число вершин в графе

start\_time = timeit.default\_timer()

rez= johnsons\_path(V, start, end,n)

tmJ100 = timeit.default\_timer() - start\_time

print(tmJ, tmJ100, sep='\n' )

## Результат

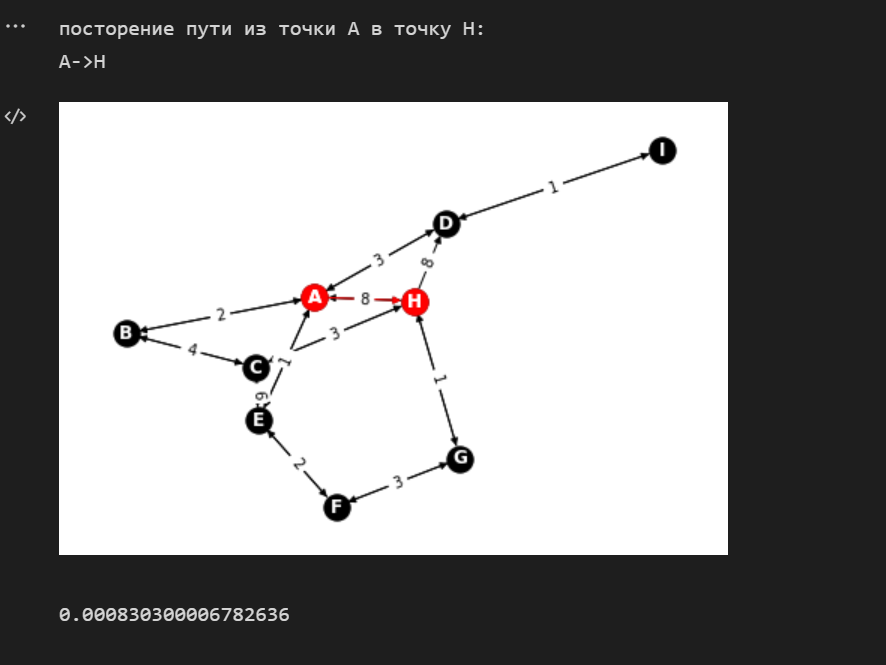


Рисунок 1 – Выполнение программы

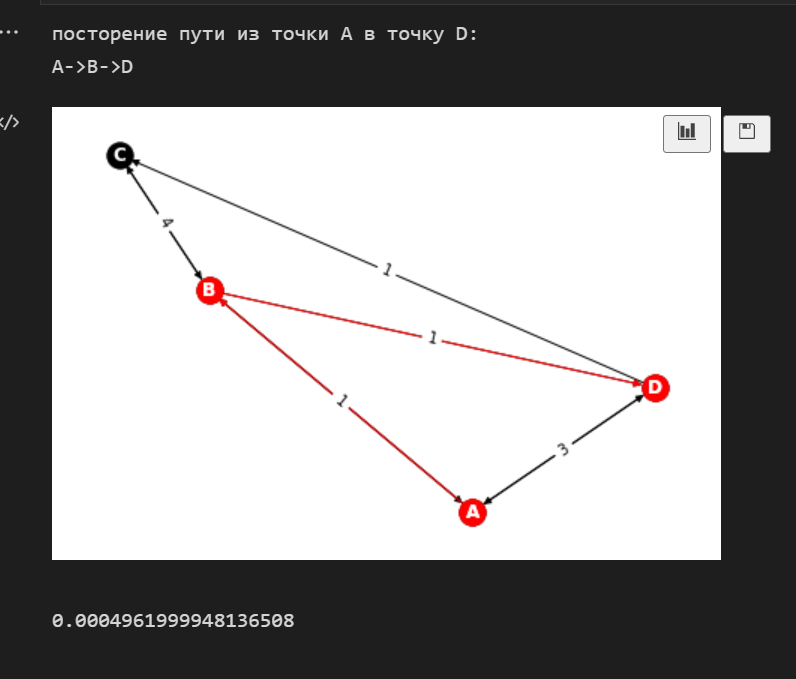


Рисунок 2 – Выполнение программы

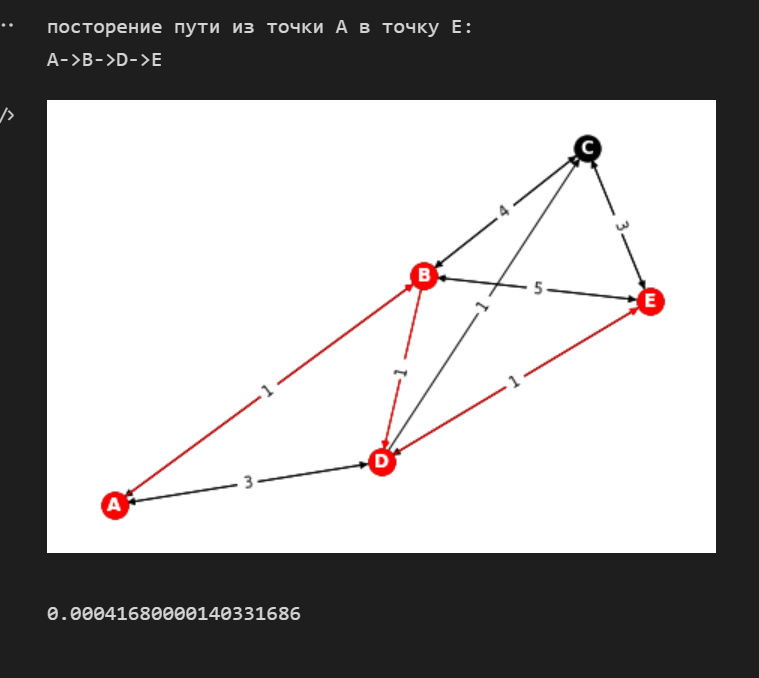


Рисунок 3 – Выполнение программы

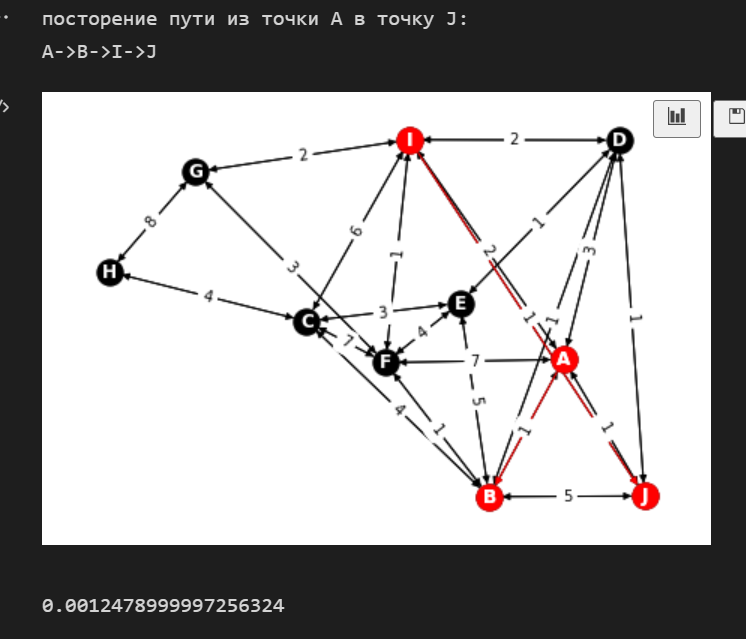


Рисунок 4 – Выполнение программы

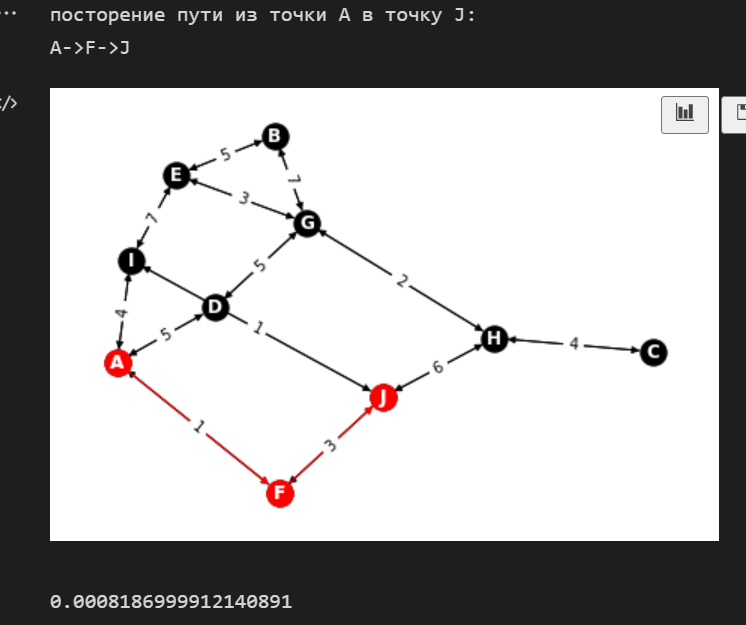


Рисунок 5 – Выполнение программы

# Таблица

|  |  |
| --- | --- |
| Граф(кол-во узлов/кол-во ребер) | Результат |
| 4/5 | 0.0004961999948136508 |
| 5/8 | 0.00041680000140331686 |
| 10/27 | 0.0012478999997256324 |
| 10/13 | 0.0008186999912140891 |