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: Максимальный поток

Студентка гр. 9382 _____ Голубева В.П.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Научиться находить величину потока в ориентированном графе, изучить и реализовать алгоритм Форда-Фалкерсона.

Задание.

Найти максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

Сеть (ориентированный взвешенный граф) представляется в виде триплета из имён вершин и целого неотрицательного числа - пропускной способности (веса).

Входные данные:

Входные данные:

N - количество ориентированных рёбер графа

v0 - исток

vn - сток

vi vj ωij - ребро графа

vi vj ωij - ребро графа

...

Выходные данные:

Pmax - величина максимального потока

vi vj ωij - ребро графа с фактической величиной протекающего потока

vi vj ωij- ребро графа с фактической величиной протекающего потока

...

В ответе выходные рёбра отсортируйте в лексикографическом порядке по первой вершине, потом по второй (в ответе должны присутствовать все указанные входные рёбра, даже если поток в них равен 0).

Вариант 1. Поиск в ширину. Поочерёдная обработка вершин текущего фронта, перебор вершин в алфавитном порядке.

Описание структур данных

Класс Graph. Принимает на вход граф в виде матрицы смежностей

Метод searchBFS(self, source, outlet, parent) класса Graph получает на вход вершину источник, вершину сток и список для поиска родителей(используем для хранения пути от источника до стока, если он есть).

Метод findFlowSize(self, source, outlet, graph, parent) - получает на вход вершину источник, вершину сток, граф и путь по нему, возвращает величину потока, которую можно пустить через этот путь.

Метод algoFordFulkerson(self, source, outlet, flow_graph) получает на вход вершину источник, вершину сток и граф для записи результата нахождения потока.

Описание алгоритма

Вводим данные, формируем матрицу смежностей. Передаём её в функцию algoFordFulkerson для нахождения максимального потока. Пока можем найти путь, ищем его функцией searchBFS - в поиске в ширину проходим подряд по соседям вершин, извлеченным из очереди вершин,

начиная от источника. Если находим не посещённую вершину, в которую можно пойти, то заносим её в очередь, а в список родителей заносим извлеченную вершину. Затем если мы обошли все вершины и нашли путь, возвращаем из функции true. Находим величину потока через найденный путь, добавляем величину к величине максимального потока, обновляем значения в исходном графе и в графе для потока.

Оценка сложности по памяти

По памяти мы должны хранить все ребра графа, количество которых равно N, и их веса. Также мы должны сформировать граф для потока, максимальный размер которого будет равен исходному графу. То есть сложность равна O(4*N).

Оценка сложности по времени

На каждом шаге алгоритм добавляет поток увеличивающего пути к уже имеющемуся потоку. Если пропускные способности всех рёбер — целые числа, легко доказать по индукции, что и потоки через все рёбра всегда будут целыми. Следовательно, на каждом шаге алгоритм увеличивает поток по крайней мере на единицу, следовательно, он сойдётся не более чем за O(f) шагов, где f — максимальный поток в графе. Можно выполнить каждый шаг за время O(E), где E — число рёбер в графе, тогда общее время работы алгоритма ограничено O(Ef).

Тестирование

Результаты тестирования программы можно посмотреть в приложении B.

Выводы.

Был изучен поиск потока в ориентированном графе и написана программа, которая его реализует.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: l3.py

```
class Graph:

    def __init__(self, graph):
        self.graph = graph
        self.row = len(graph)

    def searchBFS(self, source, outlet, parent):

        visited = self.row * [False] #list
        queue = [source]

        visited[source] = True

        while queue:
            vertex = queue.pop(0)

            for ind, value in enumerate(self.graph[vertex]):
                if visited[ind] == False and value > 0: #check vertice
for find flow
                    queue.append(ind)
                    visited[ind] = True
                    parent[ind] = vertex
        if visited[outlet]:
           return True
        return False


    def findFlowSize(self, source, outlet, graph, parent):

        flow_size = 10000
        vertex = outlet
        while vertex != source:
            if self.graph[parent[vertex]][vertex] < flow_size:
                flow_size = self.graph[parent[vertex]][vertex]
```

```
            vertex = parent[vertex]

        return flow_size

    def algoFordFulkerson(self, source, outlet, flow_graph):

        parent = self.row * [-1]
        max_flow = 0

        while self.searchBFS(source, outlet, parent):

            #find flow size to this path
            flow_size = self.findFlowSize(source, outlet, graph,
parent)

            max_flow += flow_size

            vertex = outlet
            #update graph
            while vertex != source:
                u = parent[vertex]
                self.graph[u][vertex] -= flow_size
                self.graph[vertex][u] += flow_size
                flow_graph[u][vertex] += flow_size
                flow_graph[vertex][u] -= flow_size
                vertex = parent[vertex]
            print("\nCanging graphes...")
            print ("Current adjacency matrix")
            self.printGraph(graph)
            print("\nCurrent flow graph adjacency matrix")
            self.printGraph(flow_graph)

        return max_flow

    def printGraph(self, graph):
        for i in range(self.row):
            print(graph[i])
```

```
count = int(input())
source = input()
outlet = input()
input_list = []
dict_ver = {}
list_of_vertice = []
size = 0

for i in range(count): # compute a count of defferent vertex
    input_list.append(input())
    b = input_list[i].split(" ")
    if b[0] not in list_of_vertice:
        list_of_vertice.append(b[0])
        size +=1
    if b[1] not in list_of_vertice:
        list_of_vertice.append(b[1])
        size += 1

list_of_vertice.sort()
for i in range(size):
   dict_ver[list_of_vertice[i]] = i

graph = [[0 for x in range(size)] for y in range(size)] #incoming
adjecency graph
flow_graph = [[0 for x in range(size)] for y in range(size)] #
outcoming adjecency graph

for i in range(count):
    b = input_list[i].split(" ")
    graph[dict_ver[b[0]]][dict_ver[b[1]]] = int(b[2])


res_graph = Graph(graph)

flow = res_graph.algoFordFulkerson(dict_ver[source], dict_ver[outlet],
flow_graph)
```

```
print("\nResult flow: \n")
print(flow)

for ind, value in enumerate(input_list):
    # compute flow in this edge
    v = value.split(" ")
    a = int(flow_graph[dict_ver[v[0]]][dict_ver[v[1]]])
    if a < 0:
        a = 0
    new_str = "{} {} {}".format(value[0], value[2], a)
    input_list[ind] = new_str


input_list.sort()

for i in input_list:
    print(i)
```

ПРИЛОЖЕНИЕ В

ТЕСТИРОВАНИЕ

Входные данные	Выходные данные
8 a h a c 8 a d 2 c d 16 c f 4 d g 6 g f 18 g h 5 f h 5	Canging graphes... Current adjacency matrix [0, 0, 4, 2, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [4, 0, 0, 16, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 6, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 4, 0, 0, 0, 0, 1] [0, 0, 0, 0, 0, 18, 0, 5] [0, 0, 0, 0, 0, 4, 0, 0] Current flow graph [0, 0, 4, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [-4, 0, 0, 0, 0, 4, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, -4, 0, 0, 0, 0, 4] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, -4, 0, 0] Canging graphes... Current adjacency matrix [0, 0, 4, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [4, 0, 0, 16, 0, 0, 0, 0] [2, 0, 0, 0, 0, 0, 4, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 4, 0, 0, 0, 0, 1]

	[0, 0, 0, 2, 0, 18, 0, 3] [0, 0, 0, 0, 0, 4, 2, 0] Current flow graph [0, 0, 4, 2, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [-4, 0, 0, 0, 0, 4, 0, 0] [-2, 0, 0, 0, 0, 0, 2, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, -4, 0, 0, 0, 0, 4] [0, 0, 0, -2, 0, 0, 0, 2] [0, 0, 0, 0, 0, -4, -2, 0] Canging graphes... Current adjacency matrix [0, 0, 1, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [7, 0, 0, 13, 0, 0, 0, 0] [2, 0, 3, 0, 0, 0, 1, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 4, 0, 0, 0, 0, 1] [0, 0, 0, 5, 0, 18, 0, 0] [0, 0, 0, 0, 0, 4, 5, 0] Current flow graph [0, 0, 7, 2, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [-7, 0, 0, 3, 0, 4, 0, 0] [-2, 0, -3, 0, 0, 0, 5, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, -4, 0, 0, 0, 0, 4] [0, 0, 0, -5, 0, 0, 0, 5] [0, 0, 0, 0, 0, -4, -5, 0]

	Canging graphes... Current adjacency matrix [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [8, 0, 0, 12, 0, 0, 0, 0] [2, 0, 4, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 4, 0, 0, 0, 1, 0] [0, 0, 0, 6, 0, 17, 0, 0] [0, 0, 0, 0, 0, 5, 5, 0] Current flow graph [0, 0, 8, 2, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [-8, 0, 0, 4, 0, 4, 0, 0] [-2, 0, -4, 0, 0, 0, 6, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, -4, 0, 0, 0, -1, 5] [0, 0, 0, -6, 0, 1, 0, 5] [0, 0, 0, 0, 0, -5, -5, 0] Result flow: 10 a c 8 a d 2 c d 4 c f 4 d g 6 f h 5 g f 1 g h 5

7 a f a b 7 a c 6 b d 6 c f 9 d e 3 d f 4 e c 2	… 12 a b 6 a c 6 b d 6 c f 8 d e 2 d f 4 e c 2
5 a e a b 8 b c 10 b e 3 a e 4 c e 2	… 9 a b 5 a e 4 b c 2 b e 3 c e 2
4 a c a b 2 b c 1 c d 1 c a 1	... 1 a b 1 b c 1 c a 0 c d 0
5 b d a c 5 a b 6 c d 3 b c 2	… 2 a b 0 a c 0 a d 0 b c 2 c d 2

a d 4	