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: Потоки в сети

Студент гр. 9383	____________________	Соседков К.С.
Преподаватель	____________________	Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Изучить алгоритм Форда-Фалкерсона для нахождения максимального потока в сети.

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:

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).

Задание (Вариант 7).

Поиск пути одновременно с двух сторон: от истока и от стока.

Описание работы алгоритма.

Используемые термины:

Остаточная сеть – копия оригинального графа с которой работает алгоритм.

Пропускная способность – вес ребра в графе.

Входные данные: граф, начальная вершина(исток), конечная вершина(сток).

1) Создание остаточной сети.

2) Поиск любого пути в остаточной сети от начальной вершины к конечной. Если путь не найден, алгоритм закончен.

3) Поиск минимальной пропускной способности на найденном пути.

4) Вычитание минимальной пропускной способности для кажого ребра на найденном пути.

5) Добавление минимальной пропускной способности для обратных ребер на найденном пути. Переход на шаг 2.

Отличия стандартного алгоритма Форда-Фалкерсона от индивидуального задания только в реализации шага №2.

Анализ алгоритма.

Путь в остаточной сети находится за время O(E), где E - колличество ребер в графе. На каждом шаге пропускная способность ребер на найденном пути уменьшается минимум на единицу, следовательно, алгоритм сойдется не больше чем за O(f) шагов, где f-максимальный поток в графе. Исходя из этого время работы алгоритма ограничено O(Ef).

Сложность стандартного алгоритма Форда-Фалкерсона и индивидуального задания одинакова.

Описание основных функций и переменных.

Граф реализован с помощью словаря, ключами которого являются вершины графа, а значениями — словари, содержащие смежные вершины и веса.

Пример графа:

```
graph = {„a“: {„b“: 3, „c“:4},
        {„b“: {„c“ : 1}},
        {„c“: {}}
}
```

max_flow(graph, start, end) — поиск максимального потока в графе.

find_path(graph, start, end) — поиск пути от start до end. Возвращает кортеж содержащий путь и минимальное значение пропускной способности на данном пути.

transpose_graph(graph) — транспонирование графа(смена ориентации дуг графа).

build_path(graph_map, graph_map2, vertex) — восстановление пути.

Результаты тестирования представлены в таблице 1.

Тестирование.

Для основных функций max_flow(поиск максимального потока) и find_path(поиск пути в графе) были написаны тесты. Для тестирования был написан Python-скрипт - run_tests.py. Результаты тестирования представлены на Рисунке 1.

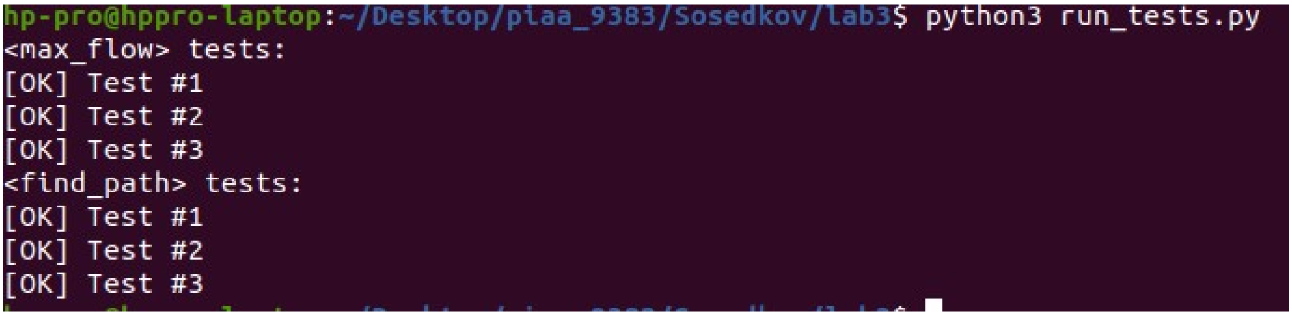

Рисунок 1: Результаты тестирования

Таблица 1. Результаты тестирования

Ввод	Вывод
7 a f a b 12 a c 32 a r 5 b c 12 c f 55 b c 90 r f 13 e f 3.0	49 a b 12 a c 32 a r 5 b c 12 c f 44 r f 5
5 a b a b 12 b a 12 b c 2 a c 4 c b 12	16 a b 12 a c 4 b a 0 b c 0 c b 4
2 a	12 a c 12

c a c 12 c a 2	c a 0

Выводы.

При выполнении работы был изучен и реализован алгоритм Форда-Фалкерсона для нахождения максимального потока в сети, а так же был реализован нестандартный двухсторонний алгоритм поиска пути в графе.

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД.

Название файла: lab3.py

```
import copy
import sys


def transpose_graph(graph):
    transposed_graph = dict.fromkeys(graph, {})
    for v in graph:
        for u in graph[v]:
            if not transposed_graph[u]:
                transposed_graph[u] = {v: graph[v][u]}
            else:
                transposed_graph[u][v] = graph[v][u]
    transposed_graph = dict(sorted(transposed_graph.items()))
    for i in transposed_graph:
        transposed_graph[i] = dict(sorted(transposed_graph[i].items(), reverse=True))
    return transposed_graph


def build_path(graph_map, graph_map2, vertex):
    min_weight=sys.maxsize
    first_path = vertex
    while graph_map[first_path[0]]:
        min_weight = min(min_weight, graph_map[first_path[0]][list(graph_map[first_path[0]].keys())[0]])
        first_path = list(graph_map[first_path[0]].keys())[0] + first_path

    second_path = vertex
    while graph_map2[second_path[-1]]:
        min_weight = min(min_weight, graph_map2[second_path[0]][list(graph_map2[second_path[0]].keys())
[0]])
        second_path = second_path + list(graph_map2[second_path[-1]].keys())[0]

    first_path = first_path[:len(first_path)-1]
    return first_path+second_path, min_weight
```

```
def find_path(flow_graph, flow_graph_transposed, start, end):
    visited_start = [start]
    visited_end = [end]

    queue_start = [start]
    queue_end = [end]

    graph_map_start = dict.fromkeys(flow_graph, {})
    graph_map_end = dict.fromkeys(flow_graph, {})

    while len(queue_start) or len(queue_start):
        if len(queue_start):
            current = queue_start.pop()
            for vertex in flow_graph[current]:
                if vertex in visited_end and flow_graph[current][vertex] > 0:
                    graph_map_start[vertex] = {current : flow_graph[current][vertex]}
                    return build_path(graph_map_start, graph_map_end, vertex)

                if vertex not in visited_start and flow_graph[current][vertex] > 0:
                    visited_start.append(vertex)
                    graph_map_start[vertex] = {current : flow_graph[current][vertex]}
                    queue_start.append(vertex)

        if len(queue_end):
            current = queue_end.pop()
            for vertex in flow_graph_transposed[current]:
                if vertex in visited_start and flow_graph_transposed[current][vertex] > 0:
                    graph_map_end[vertex] = {current : flow_graph_transposed[current][vertex]}
                    return build_path(graph_map_start, graph_map_end, vertex)

                if vertex not in visited_end and flow_graph_transposed[current][vertex] > 0:
                    visited_end.append(vertex)
                    graph_map_end[vertex] = {current : flow_graph_transposed[current][vertex]}
                    queue_end.append(vertex)

    return ('',0)
```

```
def max_flow(original_graph, start, end):
    flow_graph = copy.deepcopy(original_graph)
    while True:
        path, min_weight = find_path(flow_graph, transpose_graph(flow_graph), start, end)
        if path:
            for u,v in zip((path)[:-1], (path)[1:]):
                flow_graph[u][v] -= min_weight
                if u not in flow_graph[v]:
                    flow_graph[v][u] = min_weight
                else:
                    flow_graph[v][u] += min_weight
        else:
            break

    return sum([original_graph[start][edge]-flow_graph[start][edge] for edge in flow_graph[start]]),
flow_graph



graph = {}
number_of_edges = int(input())
start = input()
end = input()
for i in range(number_of_edges):
    v1, v2, weight = input().split()
    if v1 in graph:
        graph[v1][v2] = int(weight)
    else:
         graph[v1] = {v2: int(weight)}
    if v2 not in graph:
        graph[v2] = {}

graph = dict(sorted(graph.items()))
for i in graph:
    graph[i] = dict(sorted(graph[i].items()))
```

```
flow_value, flow_graph = max_flow(graph, start, end)
print(flow_value)

for v1 in graph:
    for v2 in graph[v1]:
        if graph[v1][v2] - flow_graph[v1][v2] > 0:
            print(v1,v2, graph[v1][v2] - flow_graph[v1][v2])
        else:
            print(v1,v2, 0)
```