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. 8304 ____________________ Мухин А. М.

Преподаватель ____________________ Размочаева Т. В.

Санкт-Петербург

Цель работы.

Изучить алгоритм Форда-Фалкерсона и решить задачу с его помощью.

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

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).

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).

Входные данные:

N - количество ориентированных рёбер графа

v_0 - исток

v_n - сток

v_i v_j ω_{ij} - ребро графа

v_i v_j ω_{ij} - ребро графа

...

Выходные данные:

P_{max} - величина максимального потока

v_i v_j ω_{ij} - ребро графа с фактической величиной протекающего потока

v_i v_j ω_{ij} - ребро графа с фактической величиной протекающего потока

...

Выполнение работы.

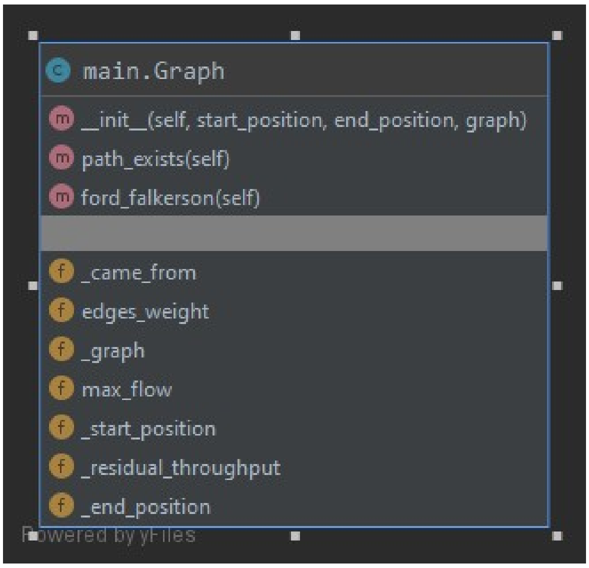
Основную работу выполняют два метода в классе Graph.

Первый - это path_exists, проверяет на наличие пути от начальной вершины к конечной, с помощью поиска в ширину, с учётом лексикографического порядка соседей текущей вершины.

Второй – это ford_falkerson, который реализует пересчёт путей в том случае, если путь ещё существует.

Также есть конструктор, в котором содержатся стартовая и конечные вершины, первоначальный граф, и его копия, которая и будет изменяться в процессе работы алгоритма. Сохранения первоначального графа необходимо для того, чтобы в конце посчитать поток, проходящий по каждому ребру в графе. Также класс содержит переменную, отвечающую за максимальный поток в графе, словарь, необходимый для восстановления пути в обратном порядке и список вершин с их весом для конечного вывода в необходимом виде.

UML диаграмма этого класса представлена ниже.



Сложность алгоритма по операциям: O (E * F), E – число ребер в графе, F – максимальный поток.

Сложность алгоритма по памяти: O (N+E), N – количество вершин, E – количество ребер.

Тестирование.

Входные данные:

7

a

f

a b 7

a c 6

b d 6

c f 9

d e 3

d f 4

e c 2

Выходные данные:

current path: acf

old width a --> c: 6

new width a --> c: 0

old width a <-- c: 0

new width a <-- c: 6

old width c --> f: 9

new width c --> f: 3

old width c <-- f: 0

new width c <-- f: 6

current path: abdf

old width a --> b: 7

new width a --> b: 3

old width a <-- b: 0

new width a <-- b: 4

old width b --> d: 6

new width b --> d: 2

old width b <-- d: 0

new width b <-- d: 4

old width d --> f: 4

new width d --> f: 0

old width d <-- f: 0

new width d <-- f: 4

current path: abdecf

old width a --> b: 3

new width a --> b: 1

old width a <-- b: 4

new width a <-- b: 6

old width b --> d: 2

new width b --> d: 0

old width b <-- d: 4

new width b <-- d: 6

old width d --> e: 3

new width d --> e: 1

old width d <-- e: 0

new width d <-- e: 2

old width e --> c: 2

new width e --> c: 0

old width e <-- c: 0

new width e <-- c: 2

old width c --> f: 3

new width c --> f: 1

old width c <-- f: 6

new width c <-- f: 8

a b 6

a c 6

b d 6

c f 8

d e 2

d f 4

e c 2

Входные данные:

a

e

a b 20

a d 10

a c 30

b a 20

b c 40

b e 30

d a 10

d c 10

d e 10

c a 30

c b 40

c d 10

c e 20

e c 20

e b 30

e d 10

Выходные данные:

current path: abe

old width a --> b: 20

new width a --> b: 0

old width a <-- b: 20

new width a <-- b: 40

old width b --> e: 30

new width b --> e: 10

old width b <-- e: 30

new width b <-- e: 50

current path: ace

old width a --> c: 30

new width a --> c: 10

old width a <-- c: 30

new width a <-- c: 50

old width c --> e: 20

new width c --> e: 0

old width c <-- e: 20

new width c <-- e: 40

current path: ade

old width a --> d: 10

new width a --> d: 0

old width a <-- d: 10

new width a <-- d: 20

old width d --> e: 10

new width d --> e: 0

old width d <-- e: 10

new width d <-- e: 20

current path: acbe

old width a --> c: 10

new width a --> c: 0

old width a <-- c: 50

new width a <-- c: 60

old width c --> b: 40

new width c --> b: 30

old width c <-- b: 40

new width c <-- b: 50

old width b --> e: 10

new width b --> e: 0

old width b <-- e: 50

new width b <-- e: 60

60

a b 20

a c 30

a d 10

b a 0

b c 0

b e 30

c a 0

c b 10

c d 0

c e 20

d a 0

d c 0

d e 10

e b 0

e c 0

e

d

Выводы.

В ходе данной лабораторной работы мы узнали, для чего нужен и как работает алгоритм Форда-Фалкерсона. А также написали программу на языке Python, которая по заданным входным значениям рассчитывала максимальный поток в сети, а также на каждом ребре.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

main.py

```
import copy
from queue import Queue
from operator import itemgetter


class Graph:
    def __init__(self, start_position, end_position, graph):
        self._start_position = start_position
        self._end_position = end_position
        self._graph = graph

        self._residual_throughput = copy.deepcopy(self._graph)
        for vertex in list(self._graph.keys()):
            for value in self._residual_throughput[vertex]:
                if value in self._residual_throughput:
                    if vertex in self._residual_throughput[value]:
                        continue
                    else:
                        
self._residual_throughput[value].update({vertex: 0})
                else:
                    self._residual_throughput[value] = {vertex: 0}

        self.max_flow = 0
        self._came_from = {}
        self.edges_weight = []

    def path_exists(self):
        queue = Queue()
        queue.put(self._start_position)
        visited = {self._start_position: True}

        while not queue.empty():
            current_elem = queue.get()
            if current_elem == self._end_position:
                return True

            for                    neighbour                    in
sorted(list(self._residual_throughput[current_elem].keys())):
                if
self._residual_throughput[current_elem][neighbour] > 0 and neighbour not
in visited:
                    queue.put(neighbour)
                    visited[neighbour] = True
                    self._came_from[neighbour] = current_elem

        return False

    def ford_falkerson(self):
        while self.path_exists():
            path = self._end_position
            while path[0] != self._start_position:
```

```
                    path = self._came_from[path[0]] + path

                print(f"current path: {path}")

                min_flow = float('inf')
                for i in range(len(path) - 1):
                    min_flow              =              min(min_flow,
self._residual_throughput[path[i]][path[i + 1]])

                for i in range(len(path) - 1):
                    print(f"\told  width  {path[i]}  -->  {path[i+1]}:
{self._residual_throughput[path[i]][path[i + 1]]}")
                    self._residual_throughput[path[i]][path[i  +  1]]  -=
min_flow
                    print(f"\tnew  width  {path[i]}  -->  {path[i  +  1]}:
{self._residual_throughput[path[i]][path[i + 1]]}")
                    print(f"\told  width  {path[i]}  <--  {path[i  +  1]}:
{self._residual_throughput[path[i + 1]][path[i]]}")
                    self._residual_throughput[path[i  +  1]][path[i]]  +=
min_flow
                    print(f"\tnew  width  {path[i]}  <--  {path[i  +  1]}:
{self._residual_throughput[path[i + 1]][path[i]]}")
                print()
                self.max_flow += min_flow

            for vertex in self._graph:
                for dest_vertex in self._graph[vertex]:
                    if       self._graph[vertex][dest_vertex]        -
self._residual_throughput[vertex][dest_vertex] < 0:
                        self.edges_weight.append((vertex,   dest_vertex,
0))
                    else:
                        self.edges_weight.append((vertex, dest_vertex,

self._graph[vertex][dest_vertex] - self._residual_throughput[vertex][
                                                          dest_vertex]))

            self.edges_weight.sort(key=itemgetter(0, 1))


    if __name__ == "__main__":
        N = int(input())
        start_position = input()
        end_position = input()
        graph = {}

        for i in range(N):
            vertex1, vertex2, weight = input().split()
            if vertex1 in graph:
                graph[vertex1].update({vertex2: int(weight)})
            else:
                graph[vertex1] = {vertex2: int(weight)}

        graph = Graph(start_position, end_position, graph)
        graph.ford_falkerson()

        print(graph.max_flow)
```

```
for i in graph.edges_weight:
```