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

Студент гр. 9382 _____ Герасев Г.А.

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:
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
...

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).

Sample Input:

7
a
f
a b 7
a c 6
b d 6
c f 9
d e 3
d f 4

e c 2

Sample Output:

12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2

Вар 4. Поиск в глубину. Итеративная реализация.

Описание структуры данных, используемой для представления графа.

Для представления графа используется встроенная структура данных – dict. В качестве ключа выступает вершина, а в качестве значения выступает dict ребер, доступных из этой вершины. Для каждого ключа — ребер дается значение, поток который можно пропустить, и поток, который уже был пропущен.

Описание алгоритма.

Считываем входные данные, заполняем словарь, тем самым формируя граф. Вызывается метод с соответствующими значениями, который возвращает пару (пропущенный поток по сети, получившаяся сеть). Более детальное описание данного метода:

1) На вход принимаются вершины и величина потока, которую мы хотим пропустить в данный момент.
2) Вызывается метод, который возвращает путь, по которому можно провести поток. Если его нет, то алгоритм останавливает работу.

Данный метод работает простым обходом в ширину со стеком – в стек добавляются вершины, до которых может «дотянуться» алгоритм (вершины в которые есть подходящие пути из рассматриваемого узла). После чего узел помечается как обработанный. Если из стека была получена обработанная вершина, то она удаляется из него. Алгоритм заканчивает работу, если была добавлена конечная вершина в стек, или если стек опустел.

3) Найденный путь используется – соответствующим образом меняется пропускная способность графа. Для этого используется специальный метод. Данный метод расшифровывает путь и делегирует задачу по обработке каждой отдельной пары вершин другому методу.
4) Когда невозможно более провести поток получившийся граф передается вместе с полученным потоком.

Оценка сложности по памяти

Для каждой вершины необходимо хранить вершины, до которых есть путь, поэтому верхняя оценка сложности равна O(N^2).

Оценка сложности по времени

На каждом шаге алгоритм добавляет поток увеличивающего пути к уже имеющемуся потоку. На каждом шаге алгоритм увеличивает поток хотя бы на единицу. Значит, что алгоритм завершится 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 В.

Выводы.

Был изучен поиск потока в ориентированном графе алгоритмом Форда-Фалкерсона и написана программа, которая его реализует.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lab3.py

```
from typing import Union



import copy
from operator import itemgetter

from typing import List, Optional
import typing
import sys


class Graph:
    def __init__(self, inputGraph = {}):
        self.graph = inputGraph

    def copy(self):2
a b 0
a c 0
a d 0
b c 2
c d 2
        # copy method
        return Graph(copy.deepcopy(self.graph))

    def print(self):
        # std output, stepick way
        list = []
        for i in self.graph:
            for j in self.graph[i]:
                res = int(self.graph[i][j][1]) # [1] because negative
flows == used flows in the algorithm
                list.append("{} {} {}".format(i,j, res))
                # list.append("{} {} {}".format(i,j,self.graph[i][j]))
        res = sorted(list)
```

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

    def addEdge(self, root, leaf, value):
        print("Adding edge from ", root, "to", leaf, "with value ",
value)
        # Add edge to the graph, if there is no one already
        if root not in self.graph:
            self.graph[root] = {}
        if leaf not in self.graph[root]:
          self.graph[root][leaf] = [value, 0] # Forward and backward
flow



    def useFlow(self, a, b, flow):
        print("Adding flow from ", a, "to", b, "equal ", flow)
        # Add flow from a to b nodes on the result map
        if a not in self.graph or b not in self.graph:
            return
        if self.graph[b][a][1] > 0:
            dif = flow - self.graph[b][a][1]
            self.graph[b][a][1] = 0
            self.graph[b][a][0] += flow - dif

            self.graph[a][b][0] -= dif
            self.graph[a][b][1] += dif


        else:
            self.graph[a][b][0] -= flow
            self.graph[a][b][1] += flow

    def givePath(self, start, end):
        print("Looking for path from ", start, "to", end)
        # returns path with positive flow from start to finish
        currentPath = []
        stack = []
        handeledNodes = {}
```

```
        if start not in self.graph or end not in self.graph:
            return currentPath
        stack.append(start)
        while stack != []:
            currentNode = stack[-1]
            if currentNode not in handeledNodes:
                currentPath.append(currentNode)
                for i in self.graph[currentNode]:
                    print("Handeling path", currentNode, "->", i)
                    if (self.graph[currentNode][i][0] > 0 or
self.graph[i][currentNode][1] > 0) and i not in handeledNodes:
                        print("Found good path")
                        stack.append(i)
                        if i == end:
                            currentPath.append(i)
                            return currentPath
                handeledNodes.update({currentNode : 0})
            else:
                stack.pop()
                if currentPath != []:
                    currentPath.pop()
        return currentPath

    def usePath(self, path):
        print("Using path ", path)
        # uses created path to change flows in the system
        values = []
        for i in range(len(path)-1):
            first = path[i]
            second = path[i+1]
            res = max(self.graph[first][second][0], self.graph[second]
[first][1])
            values.append(res)
        flow = min(values)
        if flow > 0 :
            for i in range(len(path)-1):
                first = path[i]
```

```
                second = path[i+1]
                self.useFlow(first, second, flow)

        return flow

def fordsAlg(graph : Graph, source, runoff):
    # The main algoritm
    totalFlow = 0
    graphCopy = graph.copy()
    for i in graph.graph:
        for j in graph.graph[i]:
            graphCopy.addEdge(j, i, 0)

    path = graphCopy.givePath(source, runoff)
    while path != []:
        print("Handeling new path")
        flow = graphCopy.usePath(path)
        totalFlow += flow
        print("Now total flow is ", totalFlow)
        path = graphCopy.givePath(source, runoff)

    return (graphCopy, totalFlow)

def inputHandler(inputList):
    # Handeling raw input
    graph = Graph()
    source = inputList[0]
    runoff = inputList[1]
    inputList = inputList[2:]

    for line in inputList:
        line = line.split(" ")
        graph.addEdge(line[0], line[1], float(line[2]))

    return (graph, source, runoff)
```

```
if __name__ == '__main__':
    lines = list(iter(input, ''))
    inputRaw = inputHandler(lines[1:])
    source = inputRaw[1]
    runoff = inputRaw[2]
    graph = inputRaw[0]

    res = fordsAlg(graph, source, runoff)


    for i in graph.graph:
        for j in graph.graph[i]:
            graph.graph[i][j] = res[0].graph[i][j]
    print(int(res[1]))
```

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	 Adding edge from a to c with value 8.0 Adding edge from a to d with value 2.0 Adding edge from c to d with value 16.0 Adding edge from c to f with value 4.0 Adding edge from d to g with value 6.0 Adding edge from g to f with value 18.0 Adding edge from g to h with value 5.0 Adding edge from f to h with value 5.0 Adding edge from c to a with value 0 Adding edge from d to a with value 0 Adding edge from d to c with value 0 Adding edge from f to c with value 0 Adding edge from g to d with value 0 Adding edge from f to g with value 0 Adding edge from h to g with value 0 Adding edge from h to f with value 0 Looking for path from a to h Handeling path a -> c Found good path Handeling path a -> d Found good path Handeling path d -> g Found good path Handeling path d -> a Handeling path d -> c Handeling path g -> f Found good path Handeling path g -> h

	Found good path Handeling new path Using path ['a', 'd', 'g', 'h'] Adding flow from a to d equal 2.0 Adding flow from d to g equal 2.0 Adding flow from g to h equal 2.0 Now total flow is 2.0 Looking for path from a to h Handeling path a -> c Found good path Handeling path a -> d Handeling path c -> d Found good path Handeling path c -> f Found good path Handeling path c -> a Handeling path f -> h Found good path Handeling new path Using path ['a', 'c', 'f', 'h'] Adding flow from a to c equal 4.0 Adding flow from c to f equal 4.0 Adding flow from f to h equal 4.0 Now total flow is 6.0 Looking for path from a to h Handeling path a -> c Found good path Handeling path a -> d Handeling path c -> d Found good path Handeling path c -> f Handeling path c -> a Handeling path d -> g

	Found good path Handeling path d -> a Handeling path d -> c Handeling path g -> f Found good path Handeling path g -> h Found good path Handeling new path Using path ['a', 'c', 'd', 'g', 'h'] Adding flow from a to c equal 3.0 Adding flow from c to d equal 3.0 Adding flow from d to g equal 3.0 Adding flow from g to h equal 3.0 Now total flow is 9.0 Looking for path from a to h Handeling path a -> c Found good path Handeling path a -> d Handeling path c -> d Found good path Handeling path c -> f Handeling path c -> a Handeling path d -> g Found good path Handeling path d -> a Handeling path d -> c Handeling path g -> f Found good path Handeling path g -> h Handeling path g -> d Handeling path f -> h Found good path Handeling new path

	Using path ['a', 'c', 'd', 'g', 'f', 'h'] Adding flow from a to c equal 1.0 Adding flow from c to d equal 1.0 Adding flow from d to g equal 1.0 Adding flow from g to f equal 1.0 Adding flow from f to h equal 1.0 Now total flow is 10.0 Looking for path from a to h Handeling path a -> c Handeling path a -> d 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	9 a b 5 a e 4

a b 8 b c 10 b e 3 a e 4 c e 2	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 a d 4	2 a b 0 a c 0 a d 0 b c 2 c d 2