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. 0304 _____ Мажуга Д.Р.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург,
2022

Цель работы.

Исследование способов нахождения максимального потока в сети, в частности, алгоритма Форда-Фалкерсона.

Задание.

Вариант 4.

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
```

Индивидуализация:

Вариант 4. Поиск в глубину. Итеративная реализация.

Описание алгоритма.

Для решения задачи поиска максимального потока в сети был использован алгоритм Форда-Фалкерсона. Сеть, в которой предстоит найти максимальный поток, задается в виде списка смежности. В данном списке каждой вершине

сопоставляются ребра, исходящие из этой вершины. Каждое ребро содержит информацию о том, в какую вершину данное ребро входит, далее вес данного ребра, который будет использован в качестве значения остаточной пропускной способности ребра, а также значение потока, который можно пустить по данному ребру. Предварительно, список смежности сортируется в лексикографическом порядке, а также – во время добавления новых ребер (в алгоритме – обратных ребер) для того, чтобы вывести ответ согласно требованиям в задании.

Шаги алгоритма Форда-Фалкерсона:

1. Обнуление всех значений потока на ребрах.

2. Поиск какого-либо пути из вершины-истока в вершину-сток. Если такой путь обнаружить не удается, то алгоритм заканчивает работу.

3. Для полученного пути производится поиск минимального значения остаточной пропускной способности ребра, входящего в данный путь, c_{min}.

4. Рассматриваем ребра, входящие в найденный путь. У каждого из этих ребер уменьшаем значение остаточной пропускной способности (веса) на значение c_{min}. Значение потока на этих ребрах увеличиваем на то же самое значение c_{min}.

5. Для ребер, противоположным данным, осуществляем противоположную операцию: значение остаточной пропускной способности увеличиваем на значение c_{min}, значение потока — уменьшаем на c_{min}. Если обратного ребра не было в исходной сети, добавляем его, начальные значение веса и потока у такого ребра считаем равными нулю.

6. Увеличиваем значение потока в сети на значение c_{min}.

Осуществляем операции 2-6 до тех пор, пока путь в остаточной сети существует.

Поиск из шага 2 работы алгоритма был реализован с помощью алгоритма поиска в глубину итеративным методом:

1. Был создан стек *stack*, в который первым элементом был добавлен исток. Был создан пустой список обработанных вершин *processed*, а также

словарь *transitions*, с помощью которого хранится информация, из какой вершины был осуществлен проход в данную.

2. Из стека извлекается вершина. Если данная вершина является стоком, то производим построение пути до нее. Если данная вершина конечной не является, то рассматриваем все смежные с ней вершины.

3. Обозначим текущую вершину *v*, смежную с ней — *n*. Если вершина *n* не находится в списке рассмотренных вершин и ребро *vn* имеет ненулевую остаточную пропускную способность, добавляем вершину *n* в стек, а также сохраняем информацию, что в вершину *n* можно попасть из вершины *v* (сохраняем в словаре *transitions*). Иначе, ребро *v*n не рассматривается.

4. Рассматриваемая вершина *v* добавляется в список обработанных вершин.

Шаги 2-4 повторяются до тех пор, пока не опустеет стек или не будет найден какой-либо путь до вершины стока.

Восстановление пути от истока до стока происходит путем просмотра информации о том, из какой вершины можно попасть в текущую. Текущая вершина записывается в путь, новой текущей вершиной становится та, из которой был произведен переход. Эта операция производится до тех пор, пока текущая вершина не станет равна истоку. В путь также добавляется вершина исток, после чего массив, содержащий путь, разворачивается и возвращается.

Если стек опустел, а путь не был найден, возвращается значение None, что говорит о том, что очередной путь построить не удалось.

Описание сложности алгоритма.

Обозначим количество вершин в сети как *V*, количество ребер в сети как *E*. Также обозначим значение максимального потока в сети как *f*.

Каждый шаг алгоритма увеличивает значение потока в сети как минимум на 1, если мы работаем с целыми числами, что означает, что алгоритм завершит

свое выполнение не более чем за f шагов. Таким образом, алгоритм работает за $O(f)$ операций, однако необходимо также учесть и поиск пути для увеличения потока в сети. Так как построение пути производится с помощью алгоритма поиска в глубину, данная операция занимает $O(E)$ операций. Таким образом, общая сложность по количеству операций для алгоритма Форда-Фалкерсона составляет $O(Ef)$. Если величина пропускной способности хотя бы одного из рёбер— иррациональное число, то алгоритм может работать бесконечно, даже не обязательно сходясь к правильному решению.

Для хранения графа используется список смежности. Это позволяет хранить граф за $O(E)$ памяти. Алгоритм Форда-Фалкерсона также требует хранения информации о величине потока и остаточной пропускной способности всех ребер исходного графа, а также обратных к ним, что приводит к использованию $O(2E) = O(E)$ дополнительной памяти.

Алгоритм поиска в глубину итеративным методом требует хранения обработанных вершин для избегания циклов, из чего следует сложность его по памяти $O(V)$. Итоговая сложность по памяти для алгоритма Форда-Фалкерсона составляет $O(E + V)$.

Функции и структуры данных:

Для хранения информации о ребрах был создан класс Edge, который имеет следующие поля:

destinationVertexName –вершина, в которую входит данное ребро.

capacity – пропускная способность ребра.

currentFlow –текущее значения протекающего через ребро потока.

wasRead – булево свойство, указывающее на то, является ли данное ребро исходно заданным или данное ребро появилось в графе в ходе работы алгоритма Форда-Фалкерсона.

Для хранения информации о всем графе используется словарь с названием *graph*. Данный словарь по ключу хранит название вершины, а

по значению список смежных вершин.

Перед запуском работы алгоритма Форда-Фалкерсона, введенный с клавиатуры граф сортируется: сперва каждый список сортируется по возрастанию имени вершины, в которую входит ребро из списка, а затем свойства словаря сортируются по возрастанию названия вершины. Данная сортировка позволяет корректно вывести результат работы алгоритма, в соответствии с требованиями к заданию. За сортировку графа отвечает функция def sortGraph(graph), которая принимает на вход граф graph и возвращает его отсортированную версию.

В ходе работы алгоритма нам часто необходимо получить доступ к определенному ребру. Для этого была написана функция *def getEdge(graph, source, destination)*, которая принимает на вход граф *graph*, название вершины *source*, из которой исходит искомое ребро и название вершины *destination*, в которое входит ребро. В случае наличия данного ребра в графе функция вернет данное ребро, иначе

– *None*.

Для вывода графа в консоль используется функция *def logGraph(graph*), которая принимает на вход граф *graph* и выводит его в консоль.

Ниже перечислены функции, необходимые для работы алгоритма:

def findFlow(start, end, graph) – функция, запускающая алгоритм Форда-Фалкерсона по поиску максимального потока в сети. Данная функция принимает на вход название вершины истока *start*, название вершины стока *end* и граф *graph*, а возвращает максимальный поток в данном графе.

def pathFinder(start, end, graph) – функция для поиска доступного пути в графе. Данная функция принимает на вход название вершины истока *start*, название вершины стока *end* и граф *graph*, а возвращает найденный путь в виде списка названий вершин от истока к стоку.

def pathFlow(path, graph) – функция для поиска максимального возможного

потока для пути *path* в графе *graph*.

Тестирование.

Результаты тестирования представлены в табл. 1

Таблица 1.

№ п/п	Входные данные	Выходные данные	Комментарий
1.	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	Пример корректной работы алгоритма для первого степа
2.	11 a h a b 4 a c 3 b d 5 b f 6 c e 8 c d 9 d f 7 d e 2 e g 10 f g 3 g h 1	1 a b 0 a c 1 b d 0 b f 0 c d 0 c e 1 d e 0 d f 0 e g 1 f g 0 g h 1	Ещё один пример корректной работы алгоритма для первого степа
3.	7 a z a b 7 a c 6 b d 6 c f 9 d e 3 d f 4 e c 2	0 a b 0 a c 0 b d 0 c f 0 d e 0 d f 0 e c 0	Путь между данными вершинами не был найден
4.	9 b	13 a b 0	Ещё один пример корректной работы

	e a b 1 a c 2 b d 7 b e 8 a g 2 b g 6 c e 4 d e 4 g e 1	a c 0 a g 0 b d 4 b e 8 b g 1 c e 0 d e 4 g e 1	алгоритма. В данном примере идём не из вершины a.

Выводы.

В ходе работы был рассмотрен алгоритм поиска максимального потока в сети — алгоритм Форда-Фалкерсона. Поиск пути из истока в сток был реализован с помощью алгоритма поиска в глубину итеративным методом, при этом во время поиска, вершины обрабатывались в алфавитном порядке, т.к. при формировании графа, он был отсортирован. Сложность по количеству операций алгоритма составился O(Ef), по памяти — O($E + V$).

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ.

Файл: main.py

```
demo = True

class Edge:
    def_init_(
        self,
        destinationVertexName,
        capacity,
        currentFlow=0,
        wasRead=False
    ):
        if demo:
            print(f'\n[СОЗДАНИЕ_ЭКЗЕМПЛЯРА_РЕБРА]')
            print(f'Создание экземпляра Ребра со следующими
характеристиками:')
            print(f'- Вершина, в которую входит ребро:
{destinationVertexName}')
            print(f'- Пропускная способность ребра: {capacity}')
            print(f'- Текущий поток, протекающий по ребру: {currentFlow}')
            print(f'- Принадлежит ли данное ребро исходному графу:
{wasRead}')

        self.destinationVertexName = destinationVertexName # Вершина, в
которую входит ребро
        self.capacity              = capacity               # Пропускная
способность ребра.
        self.currentFlow           = currentFlow            # Величина
текущего потока, проходящего
                                                            # через данное
ребро
        self.wasRead               = wasRead                # Было ли ребро
в исходном графе

# Функция для сортировки графа.
def sortGraph(graph):
    if demo:
        print('\n[СОРТИРОВКА]\nПроцесс сортировки словаря-графа: сперва по
символу вершины назначения, а потом - вершины исочника')

    # Сортировка списка смежности
    # - по символу вершины, в которое входит ребро
    for key in graph:
        graph[key] = sorted(graph[key], key=(lambda x:
x.destinationVertexName))
    # - по символу вершины, из которого исходит ребро.
    graph = dict(sorted(graph.items()))
    return graph

# Функция для поиска потока в графе
def findFlow(start, end, graph):
    if demo:
        print(f'[ЗАПУСК_АЛГОРИТМА_ФОРДА_ФАЛКЕРСОНА]')
        print(f'Запуск цикла для проведения алгоритма поиска макс. потока в
графе.')
```

```
totalFlow = 0

    while True:
        if demo:
            print(f'Новая итерация работы алгоритма Форда-Фалкерсона.')
        # Получение пути в остаточной сети
        path = pathFinder(start, end, graph)

        # Если путь не найден, то максимальное значение потока уже
 посчитано
        if not path:
            if demo:
                print(f'Путь не был найден. Конец алгоритма.')
            break
        else:
            if demo:
                print(f'Текущий путь: {path}')

        # Получаем максимальное возможное значение
        # протекающего потока в данном пути
        flow = pathFlow(path, graph)
        if demo:
            print(f'Полученное значение текущего потока: {flow}')

        # Увеличиваем значение потока всех ребер в пути
        totalFlow += flow
        if demo:
            print(f'Увеличиваем значение текущего потока на полученный
 поток {flow}. Весь поток теперь равен: {totalFlow}')

        # Изменяем значение потока всех ребер в пути
        pathLen = len(path)
        if demo:
            print(f'Запускаем цикл для изменения значений потока и
 пропускных способностей для ребер, входящих в состав пути.')
        for i in range(1, pathLen):
            # Изменяем значение остаточной пропускной способности
            # и потока в ребре
            edge = getEdge(graph, path[i-1], path[i])
            edge.capacity    -= flow
            edge.currentFlow += flow
            if demo:
                print(f'Для ребра {path[i-1]} -> {path[i]} (прямого
 направления) определены новая пропускная способность ({edge.capacity}) и
 текущий поток ({edge.currentFlow})')
                print(f'Обработаем ребро обратного направления.')

            # Изменяем значение остаточной пропускной способности
            # и потока в обратном ребре
            reverseEdge = getEdge(graph, path[i], path[i-1])
            if reverseEdge:
                reverseEdge.capacity   +=   flow
                reverseEdge.currentFlow -= flow
                if demo:
                    print(f'Для ребра {path[i]} -> {path[i-1]} (обратного
 направления) определены новая пропускная способность
```

```
({reverseEdge.capacity}) и текущий поток ({reverseEdge.currentFlow})')


            else:
                if demo:
                    print(f'Ребра обратного направления, т.е ребра
{path[i]} -> {path[i-1]} нет в графе (ребро не определено). Создадим его.')

                # Если обратного ребра нет, создадим его
                if path[i] not in graph:
                    graph[path[i]] = []
                newEdge = Edge(path[i-1], flow, -flow, False)

                if demo:
                    print(f'Для ребра {path[i]} -> {path[i-1]} (обратного
направления) определены новая пропускная способность ({newEdge.capacity}) и
текущий поток ({newEdge.currentFlow})')
                    print(f'Добавим это ребро в граф и отсортируем его
(т.е. граф).')
                graph[path[i]].append(newEdge)

                # И заного отсортируем, т.к. мы добавили ребро,
                # которого изначально не было в графе
                graph = sortGraph(graph)

    return totalFlow

# Поиск пути в графе
# IDDFS
def pathFinder(start, end, graph):
    if demo:
        print(f'\n[ПОИСК_ПУТИ]')
        print(f'Запуск процесса поиска пути в графе. Поиск пути проводится
по итеративному поиску в глубину.')

    processed   = [] # Список обработанных вершин
    transitions = {} # Переходы: (to, from), т.е. в обратном порядке.
    stack       = [] # Стек (для реализации Итеративного DFS)

    if demo:
        print(f'Инициализируем стек. Кладем в него имя первой вершины
(истока) - {start}')
        print(f'Начинаем цикл поиска в глубину:')

    stack.append(start)
    while stack:
        cur = stack.pop()

        if demo:
            print(f'\tНовая итерация. Достаем последний элемент стека:
{cur}')

        # Если вершина является стоком,
        # восстанавливаем путь от нее до начала
        if cur == end:
            if demo:
                print(f'\tДанная вершина является стоком. Восстанавливаем
путь от стока к истоку.')
```

```
            path = []
            while cur != start:if demo:
                    print(f'\tТекущая вершина: {cur}, ее "предок": 
{transitions[cur]}.\n\tДобавляем текущую вершину в список для хранения 
пути.')
                path.append(cur)
                cur = transitions[cur]
            path.append(start)
            if demo:
                print(f'\tДобавляем исток "{start}" в список для хранения 
пути.\n\tПолученный путь: {path}')

            # Обращаем список пути, т.к.
            # он заполнялся в обратном порядке
            path.reverse()
            if demo:
                print(f'\tПриведем данный список в надлежащий вид, а 
именно, обратим его.\n\tВ итоге получаем верный путь: {path}.')
                print(f'Поиск пути окончен.')
            return path
        else:
            if demo:
                print(f'\tТекущая вершина не совпадает со стоком "{end}". 
Строим путь дальше')

        # Просматриваем исходящие из данной вершины
        # ребра ненулевой пропускной способности
        if demo:
            print(f'\tПросматриваем, к каким вершинам есть доступные ребра 
из текущей вершины:')
        for edge in graph[cur]:

            if demo:
                print(f'\tРебро из {cur} в {edge.destinationVertexName}:')

            # Ребро уже было пройдено, либо его пропускная способность 
равна 0
            if ((edge.destinationVertexName in processed) or edge.capacity 
== 0):
                if demo:
                    print(f'\tДанное ребро уже было пройдено, либо его 
пропускная способность равна нулю (пропускная способность - 
{edge.capacity}).Переход к следующему ребру.')
                continue

            if demo:
                print(f'\tДанное ребро не было пройдено, а его пропускная 
способность  не равна нулю (пропускная способность - {edge.capacity}).')

            if (edge.destinationVertexName not in transitions):
                if demo:
                    print(f'\tДобавим о данном ребре информацию в словарь 
обратных переходов (для того, чтобы суметь из конечной точки построить путь 
к истоку).')
                transitions[edge.destinationVertexName] = cur

            if demo:
                print(f'\tДобавим назначение данного ребра в стек.')
```

```
            stack.append(edge.destinationVertexName)if demo:
            print(f'\tЗапишем, что мы обработали вершину {cur}. Это нужно
для того,\nчтобы если к данной вершине ведет несколько ребер, мы не
обрабатывали ее по-новой.')
        processed.append(cur)

    # Если путь не найден, возвращаем нулевое значение.
    if demo:
        print(f'Путь не был найден.')
    return None

# Функция для вычисление максимального потока
# для данного пути
def pathFlow(path, graph):
    if demo:
        print(f'\n[ВЫЧИСЛЕНИЕ_ПОТОКА]')
        print(f'Вычисление максимального возможного потока для пути:
{path}')
        print('Запуск цикла обхода ребер:')

    minCapacity = 0

    for i in range(1, len(path)):
        edge = getEdge(graph, path[i-1], path[i])

        # Минимальное значение найдено
        # либо данная итерация является первой
        if minCapacity == 0 or minCapacity > edge.capacity:
            minCapacity = edge.capacity

        if demo:
            print(f'\t- Шаг {i}')
            print(f'\tТекущее ребро: {path[i-1]} -> {path[i]}, его
пропускная способность: {edge.capacity}')
            print(f'\tТекущая минимальная пропусная способность:
{minCapacity}')

    if demo:
        print(f'Итоговая пропускная способность, т.е. максимальный поток
для данного пути: {minCapacity}')

    return minCapacity

# Поиск ребра
def getEdge(graph, source, destination):
    if demo:
        print(f'\n[ПОИСК_РЕБРА_В_ГРАФЕ]')

    if source in graph:
        for edge in graph[source]:
            if edge.destinationVertexName == destination:
                if demo:
                    print(f'Ребро с источником "{source}" и назначением
"{destination}" найдено в графе.')
                    print(f'({source} -> {destination}, пропускная
способность: {edge.capacity}, текущий поток: {edge.currentFlow})')
                return edge
```

```
    if demo:
        print(f'Ребро с источником "{source}" и назначением "{destination}"
не было найдено в графе.')


# Вывод графа в консоль
def logGraph(graph):
    if demo:
        print(f'\n[ЛОГИРОВАНИЕ_ИСХОДНОГО_ГРАФА_В_КОНСОЛЬ]')

    for source in graph:
        for edge in graph[source]:

            # Проверка на наличие ребра в исходном графе
            if edge.wasRead:
                flowOutput = edge.currentFlow if edge.currentFlow > 0 else
0
                print(f'{source} {edge.destinationVertexName}
{flowOutput}')

def main():
    # Ввод данных
    if demo:
        print(f'[ВВОД_ДАННЫХ]')
        print(f'Введите следующие данные:')
        print(f'- Количество ребер')
        print(f'- Название стартовой вершины')
        print(f'- Название конечной вершины')

    edges = int(input())
    start  = input()
    end    = input()

    # Список смежности
    graph = {}

    if demo:
        print(f'Введите {edges} ребро/ребер и их пропускные способности:')

    # Заполнение списка смежности
    for _ in range(edges):
        source, destination, weight = input().split()
        weight = int(weight)

        if source not in graph:
            graph[source] = []

        newEdge = Edge(destination, weight, 0, True)
        graph[source].append(newEdge)

    #  Сортировка  графа
    graph = sortGraph(graph)

    # Поиск максимального потока
    totalFlow = findFlow(start, end, graph)

    if demo:
```

```
print('[РЕЗУЛЬТАТ]')
     print(totalFlow)

     logGraph(graph)


 if_name_== "_main__":
     main()
```