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. 8382 Мирончик П.Д.

Преподаватель Фирсов М.А.

Санкт-Петербург
2020

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 Wij - ребро графа
Vi Vj Wij - ребро графа
...

Выходные данные:
Pmax - величина максимального потока
Vi Vj Wij - ребро графа с фактической величиной протекающего потока
Vi Vj Wij - 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. 3: Поиск в глубину. Рекурсивная реализация

ОПИСАНИЕ СТРУКТУР

class Path

Stores data of paths between heads.

from_head, to_head: edges of path

total_size: maximum flow

used_size: current flow from from_head to to_head

def use(self, from_head, use_size)

Applies maximum available flow, comes from from_head. If from_head equals self.from_head, increases used_size on min(use_size, self.total_size - self.used_size), otherwise decreases used_size on min(self.used_size, use_size). So used_size is clamped between 0 and self.total_size.

:param from_head: Head, form which flow appears

:param use_size: Flow's size, that comes to this path

:return: Flow's size, that was applied

def remain_for(self, head)

Computes maximum available flow, that may be passed from [head]

:param head: Head, from which flow comes

:return: Maximum flow from [head]

def other_head(self, head)

Get head from path's edges, that not equals [head]

:param head: Head with which should compare

:return: One of from_head|to_head, that not equals to [head]

class Head

Class that stored data about heads

name: name of head

paths: list of paths, that're connected with this head (enters or exits from this head)

def add_path(self, to_head, size)

Add path between this [*self*] and [*to_head*]

:param to_head: End of path

:param size: Path's from size

def compute_out_paths_count(self)

Computes count of not empty output paths (paths, where *used_size* < *total_size*)

:return: Count of not empty output paths (paths, where *used_size* < *total_size*)

def create_sorted_paths(self)

Creates array based on self.paths from which excluded paths to heads, that are contains in [stack]

:return: Array based on self.paths from which excluded paths to heads, that are contains in [stack]

Другие функции, не входящие в состав классов:

def fill(cur_head, end_head, stream)

Main recursive function, that fills system.

:param cur_head: Head on current step

:param end_head: End of system

:param stream: Flow, comes into current head

:return: Flow's size, that comes to [end_head] through [cur_head], that was computed from recursive calls

def log(text)

Logger, that prints log messages if logging is enabled

:param text: Log's text to print

АЛГОРИТМ

При считывании данных формируется *Map* с ключами – названиями вершин и значениями – объектами *Head* соответствующих вершин. Каждая вершина содержит массив объектов *Path* путей, которые соединяют вершину с другими вершинами.

Затем вызывается функция *fill,* в которую передается начальная вершина, конечная, а также размер потока (для начальной вершины это *math.inf*).

Функция *fill* работает следующим образом:

1. Если текущая вершина равна конечной, значит весь имеющийся на данной итерации поток попадает в выход – *return stream*.

2. В противном случае итерируем по имеющимся непустым путям (путям, где возможен поток из текущей вершины – в прямом направлении или обратном, если труба направлена от текущей вершины). Пускаем через путь максимально возможный поток (поток, не превосходящий текущую максимальную пропускную способность из текущей вершины и не превосходящий поток, текущий на текущей итерации), вызывая *fill(other_head, end_head, path_ stream),* где *other_head* – вершина, противоположная текущей на выбранном пути, *end_head* совпадает с *end_head* на текущей итерации (т.к. конечная вершина неизменна), а *path_stream* равен пропущенному по пути потоку. Затем поток по данному пути уменьшается на (*path_stream – filled*), где *filled* – поток, который реально достиг конечной вершины (*end_head*). Инкрементируем *total_used* на значение *filled,* где *total_used* – общий размер потока, который начиная с текущей итерации достиг конечной вершины. На начало выполнения функции равен нулю.

3. Возвращаем *total_used*.

После завершения fill возвращает размер потока, текущего из начальной вершины в конечную и при этом заполняет поля used_size объектов Path значениями, равными потоку, проходящему через путь.

После этого выбираются и сортируются все объекты Path и затем выводятся в консоль согласно требуемому формату.

Программа поддерживает логгирование всех выполняемых шагов. Для вывода в лог (вывод в лог является обычным выводом в консоль, однако выполняется только в том случае, если переменная *IS_LOGGING* имеет значение *True*) используется функция *log*.

ТЕСТИРОВАНИЕ

Номер теста	input	output
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
2	14 a e a b 20 b a 20 a c 30 c a 30 a d 10 d a 10 b e 30 e b 30 d e 10 e d 10 b c 40 c b 40 c e 20 e c 20	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 e 20 d a 0 d e 10 e b 0 e c 0 e d 0
3	9 a f a c 10 c f 7 a d 3 d f 1 d b 4 d e 5	13 a c 10 a d 3 b f 5 c d 3 c f 7 d b 4 d e 1 d f 1

	e b 5 b f 10 c d 7	e b 1

ВЫВОД

В процессе выполнения работы была решена задача поиска максимального потока в сети с использованием рекурсивной реализации алгоритма Форда-Фалкерсона. При решении задачи использовались паттерны объектно-ориентированного программирования для упрощения разработки программы и улучшения читаемости кода.

ПРИЛОЖЕНИЕ А.
КОД ПРОГРАММЫ

```
from math import *

IS_LOGGING = True


def log(text):
    """
    Logger, that prints log messages if logging is enabled
    :param text:  Log's text to print
    """
    if IS_LOGGING:
        print(text)


class Path:
    """
    Stores data of paths between heads.

    from_head, to_head: edges of path
    total_size: maximum flow
    used_size: current flow from from_head to to_head
    """
    def __init__(self, from_head, to_head, total_size):
        self.from_head = from_head
        self.to_head = to_head
        self.total_size = total_size
        self.used_size = 0
        log("Create path {0}-{1} with size {2}".format(from_head.name,
to_head.name, total_size))

    def use(self, from_head, use_size):
        """
        Applies maximum available flow, comes from from_head. If
from_head equals self.from_head,
        increases used_size on min(use_size, self.total_size -
self.used_size), otherwise decreases
        used_size on min(self.used_size, use_size). So used_size is
clamped between 0 and self.total_size

        :param from_head: Head, form which flow appears
        :param use_size: Flow's size, that comes to this path
        :return: Flow's size, that was applied
        """
        if from_head == self.from_head:
            diff = min(use_size, self.total_size - self.used_size)
        else:
            diff = -min(self.used_size, use_size)

        self.used_size += diff
        log("Used size of {0} was changed on {1}".format(str(self),
diff))
        return abs(diff)
```

```
    def remain_for(self, head):
        """
        Computes maximum available flow, that may be passed from
[head]
        :param head: Head, from which flow comes
        :return: Maximum flow from [head]
        """
        assert head == self.from_head or head == self.to_head

        if head == self.from_head:
            return self.total_size - self.used_size
        else:
            return self.used_size

    def other_head(self, head):
        """
        Get head from path's edges, that not equals [head]
        :param head: Head with which should compare
        :return: One of from_head|to_head, that not equals to [head]
        """
        assert head == self.from_head or head == self.to_head

        if head == self.from_head:
            return self.to_head
        else:
            return self.from_head

    def __str__(self):
        return "Path from=" + self.from_head.name + ", to=" +
self.to_head.name + ", total=" + str(self.total_size) + ", used=" +
str(self.used_size)

    def __format__(self, format_spec):
        return "Path from=" + self.from_head.name + ", to=" +
self.to_head.name + ", total=" + str(self.total_size) + ", used=" +
str(self.used_size)


class Head:
    """
    Class that stored data about heads

    name: name of head
    paths: list of paths, that're connected with this head (enters or
exits from this head)
    """
    def __init__(self, name):
        self.name = name
        self.paths = []
        log("Create head {0}".format(name))

    def add_path(self, to_head, size):
        """
        Add path between this [self] and [to_head]
        :param to_head: End of path
        :param size: Path's from size
        """
```

```
        path = Path(self, to_head, size)
        self.paths.append(path)
        to_head.paths.append(path)
        log("Add path between {0}-{1} with size {2}".format(self.name,
to_head.name, size))

    def compute_out_paths_count(self):
        """
        Computes count of not empty output paths (paths, where
used_size < total_size)
        :return: Count of not empty output paths (paths, where
used_size < total_size)
        """
        count = 0

        for path in self.paths:
            if path.remain_for(self) > 0 and path.other_head(self) not
in stack:
                count += 1

        return count

    def create_sorted_paths(self):
        """
        Creates array based on self.paths from which excluded paths to
heads, that are contains in [stack]
        :return: Array based on self.paths from which excluded paths
to heads, that are contains in [stack]
        """
        sorted_paths = []

        for path in self.paths:
            if path.other_head(self) not in stack:
                sorted_paths.append(path)

        return sorted_paths

    def __str__(self):
        return "Head " + self.name

    def __format__(self, format_spec):
        return "Head " + self.name


def fill(cur_head, end_head, stream):
    """
    Main recursive function, that fills system.
    :param cur_head: Head on current step
    :param end_head: End of system
    :param stream: Flow, comes into current head
    :return: Flow's size, that comes to [end_head] through [cur_head],
that was computed from recursive calls
    """
    assert stream >= 0

    if stream == 0:
        log("Flow for head {0} is 0, returning".format(cur_head.name))
        return 0
```

```
    if cur_head == end_head:
        log("End head reached, flow {0} was added".format(stream))
        return stream

    log("Process {0} head".format(cur_head.name))
    stack.append(cur_head)
    paths = cur_head.create_sorted_paths()
    total_used = 0

    for path in paths:
        used = path.use(cur_head, stream)
        filled = fill(path.other_head(cur_head), end_head, used)
        stream -= filled
        path.use(cur_head, filled - used)
        total_used += filled

    log("Head {0} processed".format(cur_head.name))
    stack.pop()
    return total_used


# stack, where are stored current heads
stack = []

# map of heads
heads_map = {}

paths_count = int(input())
start_name = input()
end_name = input()

for i in range(paths_count):
    values = input().split()
    size = int(values[2])

    if values[0] not in heads_map:
        heads_map[values[0]] = Head(values[0])

    if values[1] not in heads_map:
        heads_map[values[1]] = Head(values[1])

    heads_map[values[0]].add_path(heads_map[values[1]], size)

print(fill(heads_map[start_name], heads_map[end_name], inf))

paths_set = set()

# here we are creating set of all paths, exists in our system
for head in heads_map.values():
    for path in head.paths:
        paths_set.add(path)

paths = []

# transform paths's set into array
for path in paths_set:
    paths.append(path)
```

```
# sorting paths with requirements
paths.sort(key=lambda p: p.to_head.name)
paths.sort(key=lambda p: p.from_head.name)

# processing case, when there is path from a to b and from b to a at
one time. If we found this situation,
# we should decrease both of paths on min(first.flow_size,
second.from_size), because it's impossible when from
# comes in both sides
for path in paths:
    for other in paths:
        if path.from_head == other.to_head and path.to_head ==
other.from_head:
            min_flow = min(path.used_size, other.used_size)
            path.used_size -= min_flow
            other.used_size -= min_flow

# print paths
for path in paths:
    print("{0} {1} {2}".format(path.from_head.name, path.to_head.name,
path.used_size))
```