**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** | 7afa b 7a c 6b d 6c f 9d e 3d f 4e c 2 | 12a b 6a c 6b d 6c f 8d e 2d f 4e c 2 |
| **2** | 14aea b 20b a 20a c 30c a 30a d 10d a 10b e 30e b 30d e 10e d 10b c 40c b 40c e 20e c 20 | 60a b 20a c 30a d 10b a 0b c 0b e 30c a 0c b 10c e 20d a 0d e 10e b 0e c 0e d 0 |
| **3** | 9afa c 10c f 7a d 3d f 1d b 4d e 5e b 5b f 10c d 7 | 13a c 10a d 3b f 5c d 3c f 7d b 4d e 1d f 1e 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))