**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 в сети**

|  |  |  |
| --- | --- | --- |
| Студент гр. 0303 |  | Архипов В. А. |
| Преподаватели |  | Фирсов М. А. |

Санкт-Петербург

2022

## Цель работы.

Реализовать программу для распределения максимального потока по сети на основе алгоритма Форда-Фалкерсона.

## Задание.

Вариант 3: Поиск в глубину. Рекурсивная реализация

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

- исток

- сток

- ребро графа

- ребро графа

…

Выходные данные:

- величина максимального потока

- ребро графа с фактической величиной протекающего потока

- 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

## Выполнение работы.

**Описание работы алгоритма.**

Алгоритм можно разделить на 3 части: поиск пути из истока в сток, нахождение потока, который можно пропустить через найденный путь и непосредственно само проталкивание потока по графу. Алгоритм выполняется до тех пор, пока мы находим пути из истока в сток. Как только такой путь найден не будет, алгоритм завершит свою работу.

Теперь подробнее рассмотрим каждый из этапов алгоритма:

1. Поиск пути из истока в сток.

Поиск пути в соответствии с индивидуализацией осуществляется в глубину. Рекурсивная функция поиска начинает свою работу из вершины истока. При каждом своем вызове функция смотрит на смежные с текущей вершины и определяет, можно ли перейти к ним. Осуществить переход мы можем в том случае, если эта вершина еще не была нами посещена в текущей сессии поиска и если значение потока на ребре, по которому мы собираемся перейти, не максимально (если мы хотим перейти по направлению ребра) или положительно (если мы хотим пойти против направления ребра). Из вершин, куда мы можем пойти, составляется список, и функция поиска вызывается для каждой вершины из данного списка. Если список таких вершин оказался пустым или если все вершины из этого списка тупиковые, то текущая вершина удаляется из пути, а поиск продолжается. Если же текущая рассматриваемая вершина является стоком, то функция останавливает свою работу и возвращает путь.

1. Определение потока, который будет в дальнейшем пропущен в сеть.

По очереди рассматриваем ребра, составляющие путь из истока в сток и вносим в список значения потоков, которые мы можем пустить по этим рёбрам. Если мы идем по направлению движения ребра, то размер потока, который можно будет протолкнуть, будет равен разности максимальной пропускной способности ребра и текущему значению потока. Если же мы идем против направления ребра, то потенциальный поток будет равен текущему потоку вдоль этого ребра. После того как все рёбра будут рассмотрены, находится минимальное значение из этого списка. Это и будет поток, который мы будем проталкивать в следующем шаге алгоритма.

1. Проталкивание потока.

Проходимся по рёбрам, образующим путь из истока в сток. Если мы движемся по направлению ребра, то увеличиваем значение потока на величину, полученную в результате работы второго пункта алгоритма, если движемся против направления ребра, то уменьшаем значение потока на эту величину.

Оценки сложности работы алгоритма:

- Сложность по памяти O(E), где E – число рёбер

- Сложность по числу операций O(), где f – значение максимального потока, V – число вершин графа, E – число рёбер графа, d – коэффициент ветвления (среднее число смежных вершин)

1. Вывод информации.

Когда найти путь из истока в сток уже нельзя, алгоритм заканчивает свою работу и выводит её результат на экран. Перед этим происходит «разгрузка» двунаправленных рёбер и сортировка всех рёбер графа по алфавиту.

**Описание функций и структур данных.**

Для хранения графа была использована структура данных словарь Python.

Ключами словаря являются вершины, из которых выходит ребро. Данными словаря являются словари, ключами которых являются вершины, в которые приходят ребра, исходящие из первоначальной вершины-ключа, а значениями является список вида [пропускная способность этого ребра, текущий поток, проходящий через него]. Иными словами данные организованы по типу {вершина: {смежная вершина\_1: [максимальная пропускная способность ребра, текущий поток по направлению этого ребра], смежная вершина\_2: [максимальная пропускная способность ребра, текущий поток по направлению этого ребра]…}.

Для хранения сети был реализован класс Graph, имеющий следующие поля:

- self.graph – словарь, описанный выше

- self.start – вершина-исток

- self.finish – вершина-сток

- self.visited – список, который хранит в себе посещенные вершины (перед началом каждого нового прогона алгоритма обнуляется)

- self.way – список, хранящий в себе путь от истока к сток. Формируется в процессе работы рекурсивной функции поиска. Перед каждым новым прогоном обнуляется

- self.flows – список, хранящий в себе значения, на который можно увеличить поток рёбер, составляющих путь из истока в сток. Перед каждым новым прогоном алгоритма обнуляется

- self.debug\_way – путь от истока к стоку с учетом направления движения (необходим для отладки)

- self.debug\_way\_ext – путь от истока к стоку с учетом направления и весов рёбер (необходим для отладки)

Методы класса Graph:

* get\_related(self, node) – позволяет найти все вершины, смежные с данной

Аргументы:

node – вершина, для которой мы хотим найти все смежные с ней вершины

Возвращаемое значение: список, содержащий вершины, смежные с node

* get\_parents(self, node) - позволяет получить список вершин, чьи рёбра входят в node (список вершин-родителей)

Аргументы:

node – вершина, для которой мы хотим найти родителей

Возвращаемое значение: список родителей вершины node

* get\_children(self, node) – позволяет получить список детей вершины node

Аргументы:

node – вершина, для которой мы хотим найти дочерние вершины

Возвращаемое значение: список детей вершины node

* find\_way(self, node, depth = 1) – рекурсивный поиск пути от истока к стоку

Аргументы:

node – вершина, с которой начинается поиск

depth – глубина рекурсии (по умолчанию равна 1)

Возвращаемого значения нет. Найденный путь запишется в поле self.way

- find\_min\_flow(self) – поиск минимального потока, который можно пропустить через найденный путь.

Аргументов нет.

Возвращаемое значение: число, равное значению потока, которое можно протолкнуть через найденный путь. Во время работы функция изменяет поле self.flows

- def relax(self, min\_flow) – выполняет проталкивание потока в сети.

Аргументы:

min\_flow – значение потока, которое нужно протолкнуть в графе

Возвращаемого значения нет. Во время работы функция изменяет поле self.graph

* def post\_processing(self) – функция обработки графа. Если есть двунаправленные рёбра, то значение потока на этих ребрах будет пересчитано (сокращено на минимальное), иными словами двунаправленные рёбра очищаются от перегрузки.

Аргументов нет.

Возвращаемого значения нет. Функция изменяет поле self.graph

Разработанный программный код см. в приложении А.

## Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | 17  a  i  a b 7  a c 4  a d 6  b a 7  b e 6  c j 5  d j 6  e c 3  e f 9  e h 8  g e 9  f i 8  j f 3  j i 5  h i 6  c e 3  c b 2 | 16  a b 6  a c 4  a d 6  b a 0  b e 6  c b 0  c e 2  c j 2  d j 6  e c 0  e f 5  e h 3  f i 8  g e 0  h i 3  j f 3  j i 5 | Ответ верный. |
| 2. | 5  a  d  a b 20  b c 20  c d 20  a c 1  b d 1 | 21  a b 20  a c 1  b c 19  b d 1  c d 20 | Ответ верный. |
| 3. | 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 | Ответ верный. |
| 4. | 11  s  t  s a 6  a s 1  a b 1  a c 5  s b 4  b c 3  b t 2  c b 3  c t 8  t c 1  t b 1 | 10  a b 1  a c 5  a s 0  b c 3  b t 2  c b 0  c t 8  s a 6  s b 4  t b 0  t c 0 | Ответ верный. |
| 5. | 5  s  t  s a 4  s b 5  a b 2  a t 3  b t 7 | 9  a b 2  a t 2  b t 7  s a 4  s b 5 | Ответ верный. |
| 6. | 11  s  t  s a 7  s b 9  s c 6  b d 14  c d 9  a e 8  a f 7  b f 7  d t 15  e t 4  f t 11 | 22  a e 4  a f 3  b d 9  b f 0  c d 6  d t 15  e t 4  f t 3  s a 7  s b 9  s c 6 | Ответ верный. |
| 7. | 7  s  t  s a 4  s b 5  a b 1  b a 1  a t 3  t b 1  b t 6 | 9  a b 1  a t 3  b a 0  b t 6  s a 4  s b 5  t b 0 | Ответ верный. |
| 8. | 11  s  t  s a 4  s b 3  a c 3  a d 2  b d 2  c d 3  c e 2  d e 2  d f 2  e t 4  f t 3 | 6  a c 3  a d 1  b d 2  c d 1  c e 2  d e 2  d f 2  e t 4  f t 2  s a 4  s b 2 | Ответ верный. |
| 9. | 9  s  t  s m 5  s n 3  s r 3  m n 6  n r 4  r k 5  n k 2  n t 4  k t 8 | 11  k t 7  m n 5  n k 2  n r 2  n t 4  r k 5  s m 5  s n 3  s r 3 | Ответ верный. |
| 10. | 11  s  t  s a 10  s b 3  s d 1  a b 1  d b 3  b c 6  d c 3  a c 2  c t 5  a t 5  d t 10 | 11  a b 1  a c 2  a t 5  b c 3  c t 5  d b 0  d c 0  d t 1  s a 8  s b 2  s d 1 | Ответ верный. |

## Выводы.

В ходе выполнения лабораторной работы была реализована программа решающая задачу на основе алгоритма Форда-Фалкерсона, использующая обход графа в глубину.

**ПРИЛОЖЕНИЕ А**  
**РАЗРАБОТАННЫЙ ПРОГРАММНЫЙ КОД**

Файл task1.py:

import logging

import sys

LEVEL = logging.DEBUG

logging.basicConfig(stream=sys.stdout, level=LEVEL)

class Graph:

    def \_\_init\_\_(self, dictionary, start, finish):

        # Словарь для хранения графа

        self.graph = dictionary

        # Вершина истока

        self.start = start

        # Вершина стока

        self.finish = finish

        # Посещенные вершины

        self.visited = []

        # Путь, по которому мы прошли

        self.way = []

        # Значения потоков на этом пути

        self.flows = []

        # Отладочный вывод пути

        self.debug\_way = []

        # Отладочный вывод пути с весами рёбер

        self.debug\_way\_ext = []

    # Получить всех смежных вершин с вершиной node

    def get\_related(self, node):

        res = []

        # Получение списка детей вершины node

        # Ребенок - вершина, в которую входит ребро из node

        res.extend(list(self.graph[node].keys()))

        # Получение списка родителей node

        # Вершина родитель - вершина, из которой выходит ребро в node

        for elem in self.graph.keys():

            if node in self.graph[elem]:

                res.append(elem)

        return res

    # Получение всех родителей вершины node

    def get\_parents(self, node):

        res = []

        for elem in self.graph.keys():

            if node in self.graph[elem]:

                res.append(elem)

        return res

    # Поиск стоковой вершины в глубину

    def find\_way(self, node, depth=1):

        logging.debug('  ' \* depth + f'Пришли в вершину \'{node}\'')

        # Добавление просмотренной вершины в путь

        if node not in self.way:

            self.way.append(node)

            self.debug\_way.append(node)

            self.debug\_way\_ext.append(node)

        # Добавление вершины в список посещённых

        logging.debug('  ' \* depth + 'Путь на данный момент: ' + ''.join(self.debug\_way))

        self.visited.append(node)

        # Условие выхода из рекурсии - пришли в стоковую вершину

        if node == finish:

            logging.debug('  ' \* depth + f'Пришли в стоковую вершину')

            logging.debug('  ' \* depth + 'Поиск пути завершен')

            logging.debug('  ' \* depth + 'Найденный путь ' + ''.join(self.debug\_way))

            return self.way

        # Список родителей текущей рассматриваемой вершины

        temp = [parent for parent in self.get\_parents(node) if

                self.graph[parent][node][1] > 0 and parent not in self.visited]

        # Список детей текущей рассматриваемой вершины

        watch = [elem for elem in self.get\_children(node) if

                 self.graph[node][elem][1] < self.graph[node][elem][0] and elem not in self.visited]

        # Список вершин, в которые мы можем пойти на данном этапе

        watch.extend(temp)

        # Просматриваем каждую вершину из списка и пытаемся построить путь до стока из неё

        if watch:

            logging.debug(

                '  ' \* depth + f'Вершины, в которые можно пойти из \'{node}\' и где мы еще не были: ' + ','.join(

                    list(set(watch))))

        else:

            logging.debug('  ' \* depth + f'Из вершины \'{node}\' некуда идти дальше')

        for child in watch:

            if child in self.get\_parents(node) and child in self.get\_children(node):

                if self.graph[node][child][0] == self.graph[node][child][1]:

                    self.debug\_way.append('<--')

                    self.debug\_way\_ext.append(f'<--{self.graph[child][node][0]}\({self.graph[child][node][1]})--')

                else:

                    self.debug\_way.append('-->')

                    self.debug\_way\_ext.append(f'--{self.graph[node][child][0]}\({self.graph[node][child][1]})-->')

            elif child not in temp:

                self.debug\_way.append('-->')

                self.debug\_way\_ext.append(f'--{self.graph[node][child][0]}\({self.graph[node][child][1]})-->')

            else:

                self.debug\_way.append('<--')

                self.debug\_way\_ext.append(f'<--{self.graph[child][node][0]}\({self.graph[child][node][1]})--')

            return self.find\_way(child, depth + 1)

        # Если из этой вершины некуда пойти, то удаляем её из нашего пути

        self.way.remove(node)

        self.debug\_way.remove(node)

        self.debug\_way\_ext.remove(node)

        logging.debug('  ' \* depth + f'Возвращаемся из вершины \'{node}\'')

        # Если после удаления текущей рассматриваемой вершины путь пустой, то прийти в сток мы не можем

        # Нет пути

        if self.way == []:

            logging.debug('Просмотрены все рёбра, путей в сток не обнаружено')

            return None

        self.debug\_way.pop()

        self.debug\_way\_ext.pop()

        # Запуск поиска из последней вершины пути

        return self.find\_way(self.way[-1], depth - 1)

    # Получить детей node

    def get\_children(self, node):

        return list(self.graph[node].keys())

    # Найти поток, который можно протолкнуть через этот путь

    def find\_min\_flow(self):

        deb\_str = []

        logging.debug('\n')

        logging.debug('Поиск значения потока для пути ' + ''.join(self.debug\_way\_ext))

        # Обнуляем список потоков, которые можем пропустить через путь

        self.flows = []

        # Проходимся по пути и записываем в список потоков значения, которые можем пропихнуть через сеть

        for i in range(len(self.way) - 1):

            prev = self.way[i]

            cur = self.way[i + 1]

            # Считаем потоки для ребер, по направлению которых мы идем

            if cur in self.get\_children(prev):

                # Если можем пройти по направлению ребра, то записываем в поток разность текущего потока и пропускной способности

                if self.graph[prev][cur][0] - self.graph[prev][cur][1] != 0:

                    self.flows.append(self.graph[prev][cur][0] - self.graph[prev][cur][1])

                    logging.debug(f'По ребру {prev}-->{cur} можно пустить: '

                                  f'{self.graph[prev][cur][0]} - {self.graph[prev][cur][1]} = {self.flows[-1]}')

                    deb\_str.append(f'{prev}--{self.flows[-1]}-->')

                # В противном случае записываем поток, идущий против направления ребра

                else:

                    self.flows.append(self.graph[cur][prev][1])

                    logging.debug(f'По ребру {prev}-->{cur} нельзя ничего пропустить, '

                                  f'поэтому мы рассмотрим поток на ребре {cur}-->{prev} = {self.graph[cur][prev][1]}')

                    deb\_str.append(f'{prev}<--{self.flows[-1]}--')

            # Считаем потоки на ребрах, против направления которых мы идем

            else:

                self.flows.append(self.graph[cur][prev][1])

                logging.debug(f'По ребру {prev}-->{cur} нельзя ничего пропустить, '

                              f'поэтому мы рассмотрим поток на ребре {cur}-->{prev} = {self.graph[cur][prev][1]}')

                deb\_str.append(f'{prev}<--{self.flows[-1]}--')

            prev = cur

            cur = self.way[i]

        # Возвращаем минимальное значение потока

        deb\_str.append(finish)

        logging.debug('Итого имеем ' + ''.join(deb\_str))

        logging.debug(f'Минимальное значение потока = {min(self.flows)}\n')

        return min(self.flows)

    # Перерасчет весов ребер

    def relax(self, min\_flow):

        deb\_str = []

        deb\_str\_pre = []

        # Проходимся по найденному пути и пересчитываем значение потока на рёбрах

        logging.debug('Релаксация')

        logging.debug('До релаксации ' + ''.join(self.debug\_way\_ext))

        for i in range(len(self.way) - 1):

            prev = self.way[i]

            cur = self.way[i + 1]

            if cur in self.get\_children(prev):

                # Если идем по направлению ребра, увеличиваем значение потока

                if self.graph[prev][cur][1] + min\_flow <= self.graph[prev][cur][0]:

                    deb\_str\_pre.append(f'{prev}--{self.graph[prev][cur][0]}\({self.graph[prev][cur][1]})-->')

                    self.graph[prev][cur][1] += min\_flow

                    logging.debug(f'Увеличиваем поток на ребре {prev}-->{cur} :\t '

                                  f'{self.graph[prev][cur][1] - min\_flow} + {min\_flow} = {self.graph[prev][cur][1]}')

                    deb\_str.append(f'{prev}--{self.graph[prev][cur][0]}\({self.graph[prev][cur][1]})-->')

                else:

                    # Если идем против направления ребра, то уменьшаем значение потока

                    deb\_str\_pre.append(f'{prev}<--{self.graph[cur][prev][0]}({self.graph[cur][prev][1]})--')

                    self.graph[cur][prev][1] -= min\_flow

                    logging.debug(f'Уменьшаем поток на ребре {cur}-->{prev} :\t '

                                  f'{self.graph[cur][prev][1] + min\_flow} - {min\_flow} = {self.graph[cur][prev][1]}')

                    deb\_str.append(f'{prev}<--{self.graph[cur][prev][0]}\({self.graph[cur][prev][1]})--')

            else:

                deb\_str\_pre.append(f'{prev}<--{self.graph[cur][prev][0]}\({self.graph[cur][prev][1]})--')

                self.graph[cur][prev][1] -= min\_flow

                logging.debug(f'Уменьшаем поток на ребре {cur}-->{prev} :\t '

                              f'{self.graph[cur][prev][1] + min\_flow} - {min\_flow} = {self.graph[cur][prev][1]}')

                deb\_str.append(f'{prev}<--{self.graph[cur][prev][0]}\({self.graph[cur][prev][1]})--')

            prev = cur

            cur = self.way[i]

        deb\_str\_pre.append(self.finish)

        deb\_str.append(self.finish)

        # logging.debug('До релаксации    ' + ''.join(deb\_str\_pre))

        logging.debug('После релаксации ' + ''.join(deb\_str))

        logging.debug('\n')

    # Сокращение значений потока на двунаправленных рёбрах

    def post\_processing(self):

        # Поиск двунаправленных рёбер

        for key in self.graph.keys():

            for value in self.graph[key].keys():

                if self.graph.get(key) and self.graph.get(value):

                    if self.graph.get(key).get(value) and self.graph.get(value).get(key):

                        # Сокращение значений потоков на двунаправленных ребрах

                        minimal = min(self.graph[key][value][1], self.graph[value][key][1])

                        self.graph[key][value][1] -= minimal

                        self.graph[value][key][1] -= minimal

k = int(input())

start = input()

finish = input()

graph = {}

for \_ in range(k):

    n = input()

    st, fin, w = n.split(' ')

    graph.setdefault(st, dict())

    # [Пропускная способность ребра, поток по ребру]

    graph[st][fin] = [int(w), 0]

graph = Graph(graph, start, finish)

way = graph.find\_way(graph.start)

# Пока есть путь из истока в сток, насыщаем ребра

while way:

    graph.relax(graph.find\_min\_flow())

    logging.debug('++++++'\*10)

    graph.way = []

    graph.debug\_way = []

    graph.debug\_way\_ext = []

    graph.visited = []

    graph.flows = []

    way = graph.find\_way(graph.start)

res = 0

# Подсчет суммы значений потоков на рёбрах, идущих из стока

for child in graph.get\_children(graph.start):

    res += graph.graph[start][child][1]

print(res)

# Вывод ответа

graph.post\_processing()

for key in dict(sorted(graph.graph.items())):

    for value in dict(sorted(dict(sorted(graph.graph.items()))[key].items())):

        print(key, value, graph.graph[key][value][1])