МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Проектирование и анализ алгоритмов»

Тема: Потоки в сети

Студент гр. 8382 _____ Нечепуренко Н.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Разработать программу для поиска максимального потока в графе между двумя заданными вершинами, использую алгоритм Форда-Фалкерсона.

Постановка задачи.

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

Индивидуальный вариант 1: Поиск в ширину. Поочерёдная обработка вершин текущего фронта, перебор вершин в алфавитном порядке.

Описание алгоритма.

Для нахождения максимального потока в графе в данной работе используется алгоритм Форда-Фалкерсона. На каждой итерации алгоритма происходит поиск пути из истока в сток, увеличивающего значение текущего потока. В результате для каждого ребра пути обновляется поток и строятся, если их нет, обратные ребра, чтобы выполнялось условие f(u, v) = -f(v, u). Алгоритм работает до тех пор, пока можно найти путь, увеличивающий текущий поток. Исследователи алгоритма доказали, что найденный поток будет максимальным.

Особенности реализации алгоритма.

Различные реализации алгоритма различаются, по существу, только способом поиска очередного пути от истока к стоку. Например, можно искать

путь поиском в глубину или поиском в ширину, притом поиск в ширину, как правило, быстрей сходится. Классический пример приведен на рисунке 1.

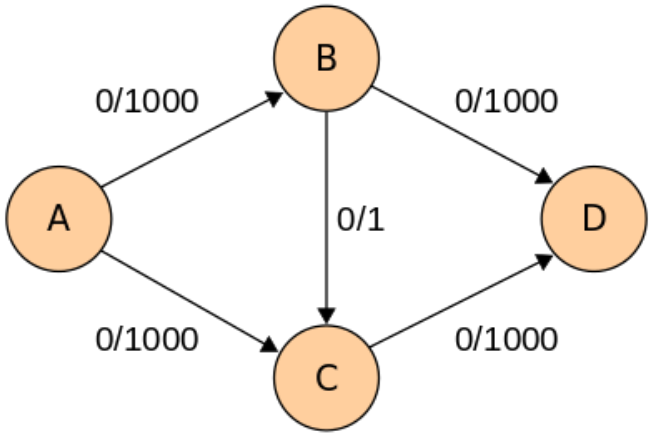

Рисунок 1 – Пример несбалансированного графа

Для такого графа поиску в ширину потребуется 2 итерации, поиску в глубину примерно в 1000 раз больше итераций. Некоторые модификации алгоритма, как алгоритм Эдмондса-Карпа или Диница, выполняют поиск кратчайшего пути. В данной работе используется поиск в ширину, притом для текущей вершины её смежные обрабатываются в лексикографическом порядке.

Поиск пути происходит в функции get_path, принимающей граф. Описание и реализация структур данных приведена в п. Функции и структуры данных, полных исходный код с исчерпывающими комментариями приведен в Приложении А.

Функции и структуры данных.

Для представления графа в программе используется класс полуребра, т.е. структуры, в которую входят: вершина, в которую ведет ребро, её пропускная способность, текущий поток, и флаг, были ли это ребро в изначальной сети. Сам граф представлен встроенной структурой данных dict, реализующий маппинг из вершины в массив полурёбер (исходный код см. в Прил. А).

В функции get_path используется словарь path_from для восстановления полученного поиском пути, словарь delta для отслеживания изменения потока,

сет visited для просмотра каждой вершины единожды. Очередь для поиска в ширину реализована с помощью интерфейса стандартной структуры list.

Сортировки производятся с помощью стандартных средств языка.

Оценка сложности алгоритма.

Для стандартной реализации алгоритма Форда-Фалкерсона имеет место оценка O(|E|F), где |E| – количество рёбер, F – максимальный поток. В худшем случае (см. рис. 1) необходимо F раз пройтись по рёбрам, их |E|, откуда и получается оценка. Более практически значимые оценки можно получить для реализаций Эдмондса-Карпа и Диница, $O(|V||E|^2)$ и $O(|V|^2|E|)$ соответственно. Стоит подчеркнуть вопрос сходимости алгоритма. В данной работе пропускные способности представлены целыми числами, что не создаёт проблем для сходимости, но в случае с иррациональными пропускными способностями существуют примеры, когда данный алгоритм не сходится.

Тестирование.

Проведём тестирование программы с выводом отладочной информации. Результат тестирования приведён в таблице 1

Таблица 1 – Тестирование программы

Входные данные	Выходные данные
3 a c a b 2 a c 2 b c 1	Считанный граф: {'a': [-> b, capacity: 2, current_flow: 0, -> c, capacity: 2, current_flow: 0], 'b': [-> c, capacity: 1, current_flow: 0]} Рассматриваем вершину a Добавляем b в очередь Добавляем c в очередь Рассматриваем вершину c ----------

	Получен путь: ['a', 'c'] Изменение потока на 2 Добавлено обратное ребро (c,a) Граф после итерации: {'a': [-> b, capacity: 2, current_flow: 0, -> c, capacity: 2, current_flow: 2], 'b': [-> c, capacity: 1, current_flow: 0], 'c': [-> a, capacity: 2, current_flow: -2]} ---------- Рассматриваем вершину a Добавляем b в очередь Рассматриваем вершину b Добавляем c в очередь Рассматриваем вершину c ---------- Получен путь: ['a', 'b', 'c'] Изменение потока на 1 Добавлено обратное ребро (b,a) Добавлено обратное ребро (c,b) Граф после итерации: {'a': [-> b, capacity: 2, current_flow: 1, -> c, capacity: 2, current_flow: 2], 'b': [-> c, capacity: 1, current_flow: 1, -> a, capacity: 2, current_flow: -1], 'c': [-> a, capacity: 2, current_flow: -2, -> b, capacity: 1, current_flow: -1]} ---------- Рассматриваем вершину a Добавляем b в очередь

	Рассматриваем вершину b ---------- Получен путь: ['c'] Изменение потока на 0 Граф после итерации: {'a': [-> b, capacity: 2, current_flow: 1, -> c, capacity: 2, current_flow: 2], 'b': [-> c, capacity: 1, current_flow: 1, -> a, capacity: 2, current_flow: -1], 'c': [-> a, capacity: 2, current_flow: -2, -> b, capacity: 1, current_flow: -1]} ---------- Максимальный поток 3 Список ребер с потоками: a b 1 a c 2 b c 1

Ввиду большого объёма выходных данных предлагается выполнить дальнейшее тестирование на рабочей машине с выводом в текстовый файл. Тесты приложены к данному отчёту.

Вывод.

В результате выполнения работы была получена реализация алгоритма Форда-Фалкерсона с использованием поиска в ширину на языке Python. Была исследована асимптотика алгоритма и особенности реализации.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ.

```
"""
Вариант 1 -- Поиск в ширину.
Поочерёдная обработка вершин текущего фронта, перебор вершин в алфавитном порядке.
"""

class SemiEdge:
    """
    Структура для хранения полуребра, т.е. вершину, в которую ведет ребро, её
    пропускную способность, текущий поток, и флаг, были ли это ребро в изначальной сети
    """

    to: str
    cost: int
    flow: int
    orig: bool
    def __init__(self, to_, cost_, flow_=0, orig_=False):
        self.to = to_
        self.cost = cost_
        self.flow = flow_
        self.orig = orig_

    def __str__(self):
        return f"-> {self.to}, capacity: {self.cost}, current_flow: {self.flow}"

    def __repr__(self):
        return self.__str__()

    """
    Переопределение сравнений для сортировки, чтобы получить корректный формат вывода
    """
    def __lt__(self, rhs):
        return self.to < rhs.to

    def __gt__(self, rhs):
```

```
            return self.to > rhs.to

        def __le__(self, rhs):
            return not self > rhs

        def __eq__(self, rhs):
            return self.to == rhs

    #Считывание данных
    edge_count = int(input())
    source = input().strip()
    drain = input().strip()
    graph = dict()

    """
    Будем хранить отображения из вершины в массив полуребер, т.е. u ->
[SemiEdge(v, ...), SemiEdge(w, ...), ...]
    для представления графа
    """

    for _ in range(edge_count):
        from_, to_, cost_ = input().split()
        graph[from_] = graph.get(from_, []) + [SemiEdge(to_, int(cost_), 0,
True)]

    print("Считанный граф: ", graph)

    def get_path(graph):
        """
        Функция нахождения пути от истока к стоку, согласно варианту,
использует поиск в ширину
        """

        path_from = dict() # мапа, для восстановления пути
        visited = set()
        queue = [source] # очередь для поиска в ширину
        delta = {source: 1e9} # для каждой вершины в пути необходимо знать
изменение потока

        while len(queue):
            """
            Поиск в ширину
```

```
            """

            cur = queue.pop()
            print(f"Рассматриваем вершину {cur}")
            visited.add(cur)
            if cur == drain: # дошли до стока
                break
            if not graph.get(cur): # у вершины нет исходящих рёбер
                continue
            for semi_edge in sorted(graph[cur]): # перебираем соседние вершины, отсортированные в лекс. граф. порядке
                to_, cost_, flow_ = semi_edge.to, semi_edge.cost, semi_edge.flow
                if to_ not in visited and flow_ < cost_: # еще не были в вершине и можно увеличить поток
                    queue.append(to_)
                    print(f"Добавляем {to_} в очередь")
                    path_from[to_] = cur
                    delta[to_] = min(delta[cur], cost_ - flow_)

        """
        Восстановление пути
        """

        result_path = [drain]; cur = drain
        while path_from.get(cur):
            result_path.append(path_from[cur])
            cur = path_from[cur]

        return list(reversed(result_path)), delta.get(drain, 0) # возвращаем дельту, чтобы знать, когда закончить алгоритм

    flow = 0
    flow_delta = 1
    while flow_delta:
        path, flow_delta = get_path(graph)
        print("-" * 10)
        print("Получен путь: ", path)
        print(f"Изменение потока на {flow_delta}")
        for cur in range(len(path)-1): # пройдем по пути и обновим потоки, добавим обратные ребра, если необходимо
            for semi_edge_idx, semi_edge in enumerate(graph[path[cur]]):
```

```
                if semi_edge.to == path[cur+1]:
                    graph[path[cur]][semi_edge_idx].flow += flow_delta # увеличим поток
                    if not graph.get(semi_edge.to):
                        graph[semi_edge.to] = [SemiEdge(path[cur], graph[path[cur]][semi_edge_idx].cost, -flow_delta)] # добавим обратное ребро
                        print(f"Добавлено обратное ребро ({semi_edge.to},{path[cur]})")
                    else:
                        """ Ищем обратное ребро, если оно есть, либо опять же добавляем новое """

                        try:
                            to = graph[semi_edge.to].index(path[cur])
                            graph[semi_edge.to][to].flow -= flow_delta
                        except:
                            graph[semi_edge.to].append(SemiEdge(path[cur], graph[path[cur]][semi_edge_idx].cost, -flow_delta))
                            print(f"Добавлено обратное ребро ({semi_edge.to},{path[cur]})")
            flow += flow_delta # обновляем макс. поток
            print("Граф после итерации:", graph)
            print("-" * 10)

    print(f"Максимальный поток {flow}")
    edges = []

    """
    Решаем проблему двойных ребер, формируем ответ
    """
    for node in graph:
        for semi_edge_idx, semi_edge in enumerate(graph[node]):
            if semi_edge.orig:
                try:
                    idx = graph[semi_edge.to].index(node)
                    if graph[semi_edge.to][idx].orig: # есть (u,v) и (v,u) принадлежащие исходной сети.
                        if semi_edge.flow > 0 and graph[semi_edge.to][idx].flow > 0: # если оба положительные, то в одно разность, в другое 0
                            if semi_edge.flow > graph[semi_edge.to][idx].flow:
```

```
                                    semi_edge.flow                                  -=
graph[semi_edge.to][idx].flow
                                    graph[semi_edge.to][idx].flow = 0
                                else:
                                    graph[semi_edge.to][idx].flow                   -=
semi_edge.flow
                                    semi_edge.flow = 0
                            else: # иначе зануляем отрицательное
                                if semi_edge.flow > graph[semi_edge.to][idx].flow:
                                    graph[semi_edge.to][idx].flow = 0
                                else:
                                    semi_edge.flow = 0
                    except:
                        pass
                    edges.append((node, semi_edge.to, semi_edge.flow)) # добавляем
ребро в ответ
        print("Список ребер с потоками:")
        import operator # для сортировки по обеим ребрам
        edges.sort(key=operator.itemgetter(0, 1))
        for edge in edges:
            print(" ".join(map(str, edge))) # вывод списка ребер
```