МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Построение и анализ алгоритмов»
Тема: "Потоки в сети".

Студентка гр. 8383 _____ Аверина О.С.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Научиться реализовывать алгоритм поиска максимального потока в графе с использованием алгоритма Форда-Фалкерсона.

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

v_0 - исток

v_n - сток

$v_i\ v_j\ \omega_{ij}$ - ребро графа

$v_i\ v_j\ \omega_{ij}$ - ребро графа

…

Выходные данные:

P_{max} - величина максимального потока

$v_i\ v_j\ \omega_{ij}$ - ребро графа с фактической величиной протекающего потока

$v_i\ v_j\ \omega_{ij}$ - 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).

Индивидуализации для лаб. работы № 3:

Вариант 5

Поиск не в глубину и не в ширину, а по правилу: каждый раз выполняется переход по дуге, имеющей максимальную остаточную пропускную способность. Если таких дуг несколько, то выбрать ту, которая была обнаружена раньше в текущем поиске пути.

Основные теоретические положения.

В теории оптимизации и теории графов, **задача о максимальном потоке** заключается в нахождении такого потока по транспортной сети, что сумма потоков из истока, или, что то же самое, сумма потоков в сток максимальна. Идея алгоритма Алгоритм Форда-Фалкерсона заключается в следующем. Изначально величине потока присваивается значение 0: $f(u,v) = 0$ для всех u, v, принадлежащих V. Затем величина потока итеративно увеличивается посредством поиска увеличивающего пути (путь от источника s к стоку t, вдоль которого можно послать больший поток). Процесс повторяется, пока можно найти увеличивающий путь.

Алгоритм:

1. Обнуляем все потоки. Остаточная сеть изначально совпадает с исходной сетью.
2. В остаточной сети находим любой путь из источника в сток. Если такого пути нет, останавливаемся.
3. Пускаем через найденный путь (он называется увеличивающим путём или увеличивающей цепью) максимально возможный поток:
    3.1. На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью c_{min}.

3.2. Для каждого ребра на найденном пути увеличиваем поток на c_{min}, а в противоположном ему — уменьшаем на c_{min}.

3.3. Модифицируем остаточную сеть. Для всех рёбер на найденном пути, а также для противоположных им рёбер, вычисляем новую пропускную способность. Если она стала ненулевой, добавляем ребро к остаточной сети, а если обнулилась, стираем его.

4. Возвращаемся на шаг 2.

Важно, что алгоритм не конкретизирует, какой именно путь мы ищем на шаге 2 или как мы это делаем. По этой причине алгоритм гарантированно сходится только для целых пропускных способностей, но даже для них при больших значениях пропускных способностей он может работать очень долго. Если пропускные способности вещественны, алгоритм может работать бесконечно долго, не сходясь к оптимальному решению.

Реализация алгоритма Форда-Фалкерсона

Алгоритм:

Алгоритм заключается в поиске всех возможных потоков сети.

1. Считываем граф, заносим его вершины в списки смежности исходных ребер и инвертированных им.
2. Вызываем функцию поиска максимального потока. Пока не будет найден пустой путь, выполняем пункты 3-6.
3. В цикле ищем путь от истока до стока, выбирая ребра с наибольшими пропускными способностями, помечаем каждую пройденную вершину просмотренной, добавляем их в вектор *way*.
    a. Если в текущей вершине нет исходящих нерассмотренных ребер, помечаем ее просмотренной, удаляем из списка вершин и возвращаемся к предыдущей.
    b. Если после удаления вершины список оказался пуст, значит максимальный поток найден, переходим к пункту 7.

4. После нахождения пути, по списку вершин найденного потока составляется остаточная сеть: каждое ребро потока уменьшается на минимальную пропускную способность потока, а обратное ему - увеличивается.
5. Максимальный поток увеличивается на минимальную пропускную способность найденного пути.
6. Очищается список вершин потока, переходим в пункт 3.
7. Если был найден пустой путь, алгоритм завершает работу и возвращает максимальный поток графа.
8. С помощью списка смежности инвертированных ребер восстанавливаем список ребер и их фактические величины протекающего потока. Сортируем лексикографически и выводим в консоль.

Код программы содержит следующие элементы:

Класс flowSearch, который содержит

1. Структуры:

1.1. Структура *point* - хранит символ вершины и путь до нее.

1.2. unordered_map<char, vector<flow>> network – хранит список смежности входного графа, где ключ – символ вершины, а значение – это вектор пар из смежных вершин и длин путей до них.

1.3. unordered_map<char, vector<flow>> invertNetwork – хранит список смежности обратных ребер входного графа, где ключ – символ вершины, а значение – это вектор пар из смежных вершин и длин путей до них.

1.4. *unordered_map<char*, *bool> checked* – хранит флаги, просмотрена или нет данная вершина.

1.5. Структура *edge* - хранит начальную, конечную вершину ребра и его длину. Нужно при составлении списка ребер для вывода.

2. Методы:

2.1. *void inputGraph*() – метод для ввода списка ребер графа и преобразования его в список смежности.

2.2. *Void network*() – функция для поиска максимального потока в графе.

2.3. *void searchPath*(*vector<pair<char, bool>> *way*) – функция для итеративного поиска потока в графе.

Входное значение:

*vector<pair<char, bool>> *way* - указатель на список вершин потока.

Возвращаемое значение:

minResFlow - минимальная пропускная способность найденного потока.

2.4. *void sortOutput*(*int flowValue*) - функция для составления списка ребер и его вывода в консоль.

Входное значение:

int flowValue - максимальный поток графа.

Оценка сложности алгоритма.

Сложность алгоритма по времени: O(f * V^2), где f – максимальный пропускной поток графа, V – количество вершин графа. Сложность получается из того, что поиск пути имеет сложность V^2(при выборе вершины каждый раз ищется с ребром с максимальным потоком), и в худшем случае будет найден f раз, если каждый раз минимальная пропускная способность будет равна 1.

Сложность по памяти: $O(|V| + |E|)$, где $|V|$ - количество вершин, $|E|$ - количество вершин в графе, т.к. $|V| + |E|$ - это память, затрачиваемая на хранение списка смежности ребер.

Тестирование.

№	Входное значение	Результат	№	Входное значение	Результат
1	11 a d a b 7 a c 3 a f 5 c b 4 c d 5 b d 6 b f 3 b e 4 f b 7 f e 8 e d 10	15 a b 7 a c 3 a f 5 b d 6 b e 1 b f 0 c b 0 c d 3 e d 6 f b 0 f e 5	2	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
3	10 a f a b 16 a c 13 c b 4 b c 10 b d 12 c e 14 d c 9 d f 20 e d 7 e f 4	15 a b 7 a c 3 a f 5 b d 6 b e 1 b f 0 c b 0 c d 3 e d 6 f b 0 f e 5	4	6 a a a c 10 c d 10 c b 1 b c 1 a b 10 b d 10	0 a b 0 a c 0 b c 0 b d 0 c b 0 c d 0

Подробный тест.

```
Test 4:   Tests/Test2.txt
6
a
e
a b 1
a c 2
c b 2
c d 3
d e 3
b e 3

Program start:

Vertex: a
Vertex: c
Vertex: d

Flow is finding:
a c d e

From a to c
a - c: 2 - 2 = 0
c - a: 0 + 2 = 2


From c to d
c - d: 3 - 2 = 1
d - c: 0 + 2 = 2


From d to e
d - e: 3 - 2 = 1
e - d: 0 + 2 = 2

Vertex: a
Vertex: b

Flow is finding:
a b e

From a to b
a - b: 1 - 1 = 0
b - a: 0 + 1 = 1


From b to e
b - e: 3 - 1 = 2
e - b: 0 + 1 = 1

Vertex: a
There are no more paths from the source!
```

Вывод.

Было получено теоретическое представление об алгоритме Форда-Фалкерсона. Была реализована программа для поиска максимального потока в графе.

ПРИЛОЖЕНИЕ А

```
#include <vector>
#include <stack>
#include <iostream>
#include <unordered_map>
#include <fstream>
#include <algorithm>

#define DEBUG

// Вар. 5. Поиск не в глубину и не в ширину, а по правилу: каждый раз выполняется переход по дуге,
// имеющей максимальную остаточную пропускную способность. Если таких дуг несколько, то выбрать ту,
// которая была обнаружена раньше в текущем поиске пути.

using namespace std;

class flowSearch {

    struct point {
        char vertex; int way;
    };
    struct edge {
        char from, to;
        int flow;
    };

    unordered_map<char, vector<point>>  network;  // остаточная сеть
    unordered_map<char, vector<point>> invertNetwork;

    char start, finish;

    int searchPath(vector<pair<char, bool>> *way){

        char currentPoint = start, newPoint = ' ';
        int maxFlow, minResFlow = 10000;
        bool isInvertEdge = false;

        unordered_map<char, bool> checked;
        checked[start] = true;

        while(currentPoint != finish){

            std::cout <<"Vertex: " <<currentPoint << "\n";
            maxFlow = -1;
            for (auto it : network[currentPoint]) {
                if (!checked[it.vertex] && it.way > 0 && maxFlow < it.way) {
//                if (!checked[it.vertex] && it.way > 0) {
                  newPoint = it.vertex;
                  maxFlow = it.way;
                  isInvertEdge = false;
                }
```

```
            }
            for (auto it : invertNetwork[currentPoint]) {
                if (!checked[it.vertex] && it.way > 0 && maxFlow <
it.way) {
//                if (!checked[it.vertex] && it.way > 0) {
                  newPoint = it.vertex;
                  maxFlow = it.way;
                  isInvertEdge = true;
                }
            }
            currentPoint = newPoint;
            checked[currentPoint] = true;

            if (maxFlow == -1) {
                way->pop_back();
                if(way->empty())  {
                  cout << "There are no more paths from the source!\n\n";
                  return 0;
                }
                currentPoint = way->back().first;
                continue;
            }

            way->push_back({currentPoint, isInvertEdge});

            if (maxFlow < minResFlow)
                minResFlow = maxFlow;
        }
        return minResFlow;
    }

    static int comp1(edge x, edge y) {
        return x.from < y.from;
    }
    static int comp2(edge x, edge y) {
        return x.to < y.to;
    }

public:
    explicit flowSearch(): start(' '), finish(' ') {}

    void inputGraphConsole() {
        char from, to;
        int way_, n;
        cin >> n >> start >> finish;

        for (int i = 0; i < n; ++i) {
            cin >> from >> to >> way_; // from - из, to - в

            network[from].push_back({to, way_});
            invertNetwork[to].push_back({from, 0});
        }
    }

    int inputGraph() {
        char from, to;
        int way_, n;
```

```
        ifstream file;
        file.open("/home/olyaave/CLionProjects/PAA_LAB3_2/input.txt");
        if (file.is_open()) {

            file >> n >> start >> finish;
            for (int i = 0; i < n; ++i) {
                file >> from >> to >> way_; // from - из, to - в

                network[from].push_back({to, way_});
                invertNetwork[to].push_back({from, 0});
            }
            file.close();
        } else {
            cout << "File isn't open!";
            return -1;
        }
        return 0;
    }

    void networks() {

        int minResFlow;
        int flowValue = 0;
        vector<pair<char, bool>> way;   // найденный временный путь(его
вершины)

        way.push_back({start, false});

        while (!way.empty()) {

            minResFlow = searchPath(&way);  // поиск потока
            if(way.size() > 1) {

                cout << "\nFlow is finding: \n";
                for (auto it : way) {
                    cout << it.first << " ";
                }

                vector<point> *list;
                vector<point> *invertList;
                for (auto u = way.begin(); u != (way.end() - 1); ++u)//
вершина u - "из"
                {
                    cout << "\n\nFrom " << u->first << " to " << (u + 1)-
>first << endl;
                    if (!(u + 1)->second) {
                        list = &network[u->first];
                        invertList = &invertNetwork[(u + 1)->first];
                    } else {
//                        cout << "Invert ->>>>>\n";
                        list = &invertNetwork[u->first];
                        invertList = &network[(u + 1)->first];
                    }
                    for (auto &it: *list) {
                        if (it.vertex == (u + 1)->first) {
                            cout <<u->first<< " - "<<(u + 1)->first<<":
"<<it.way<<" - "<<minResFlow<<" = ";
```

```
                            it.way -= minResFlow;
                            cout << it.way << endl;
                        }
                    }
                    for (auto &it: *invertList) {
                        if (it.vertex == u->first) {
                            cout <<(u + 1)->first<<" - "<<u->first<<":
"<<it.way<<" + "<<minResFlow<<" = ";
                            it.way += minResFlow;
                            cout << it.way << endl;
                        }
                    }
                }
                way.clear();
                way.push_back({start, 0});
                cout << "\n";
                flowValue += minResFlow;
            } else
                way.clear();
        }
        sortOutput(flowValue);
    }


    void sortOutput(int flowValue) {
        cout << "Answer: " << endl;
        cout << "\n" << flowValue << endl;

        vector<edge> answer;
        for (auto u: invertNetwork)
        {
            for (auto it: invertNetwork[u.first]) {
                answer.push_back({it.vertex, u.first, it.way});
            }
        }
        sort(answer.begin(), answer.end(), comp2);
        sort(answer.begin(), answer.end(), comp1);
        for (auto it: answer) {
            cout << it.from << " " << it.to << " " << it.flow << endl;
        }
    }
};

int main() {
    cout << "Program start:\n\n";
    flowSearch *obj = new flowSearch();
    obj->inputGraphConsole();
    obj->networks();
    return 0;
}
```