

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

Студентка гр. 9383

Лапина А.А.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Изучить алгоритм Форда-Фалкерсона поиска максимального потока в сети, а также реализовать данный алгоритм на языке программирования C++.

Основные теоретические положения.

Сеть – ориентированный взвешенный граф, имеющий один исток и один сток.

Исток – вершина, из которой рёбра только выходят.

Сток – вершина, в которую рёбра только входят.

Поток – абстрактное понятие, показывающее движение по графу.

Величина потока – числовая характеристика движения по графу (сколько всего выходит из стока = сколько всего входит в сток).

Пропускная способность – свойство ребра, показывающее, какая максимальная величина потока может пройти через это ребро.

Максимальный поток (максимальная величина потока) – максимальная величина, которая может быть выпущена из стока, которая может пройти через все рёбра графа, не вызывая переполнения ни в одном ребре.

Фактическая величина потока в ребре – значение, показывающее, сколько величины потока проходит через это ребро.

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).

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

Вариант 4. Поиск в глубину. Итеративная реализация.

Выполнение работы:

В программе реализован класс edge для хранения информации о ребре:

С переменными:

char v — куда идет ребро;

float metrika — вес (метрика) ребра;

int view — отметка является ли ребро просмотренным (0 — нет, 1 - да)

С методами:

char name() - для получения v;

float len() - для получения metrika;

void inc(int min) — увеличивает metrika (длину ребра) на заданное число

min;

void dec(int min) — уменьшает metrika (длину ребра) на заданное число

min;

В программе реализован класс answ для хранения ответа:

С переменными:

char source - исток;

char dest - сток;

int metrika - длина;

В программе были использованы следующие функции:

* void dec_metrika(Graph &G, std::string path, int min) — уменьшает вес у
нынешних ребер;

*int Search_min(std::string path, Graph G) — ищет минимальный вес в пути
path;

*void Read(Graph &G) — считывает ребра и заполняет Graph;
* void create_reverse_edges(Graph &G, std::string path, int min) — делает обратные ребра;
*std::string Search_path(Graph &G, char cur, char finish, std::string path) — ищет путь в графе;
*bool Check_v(Graph G, char start) — проверяет условие основного цикла, можно ли еще пройти по графу, возвращает 0, если нет ребер, с метрикой больше 0 из стартовой вершины;
* std::vector <answ> for_answer(Graph G, Graph G2) — составляет список пройденных ребер для ответа;
* int Algoritm(Graph &G, char start, char finish) — реализует алгоритм Форда-Фалкерсона:

Описание алгоритма работы программы:

Программа получает на вход N — количество ориентированных рёбер графа, 2 вершины start и finish, затем ребра - <исходящая вершина> <входящая вершина> <вес> до окончания ввода (ctrl+D), считывание происходит в функции Read() в Graph — словарь, где first — исходящая вершина, а second — вектор из структур (на рисунке 1 изображена структура для входных данных из задания лабораторной).

Затем с помощью функции Algoritm ищем максимальный поток в графе итеративным способом: ищем путь в графе, затем ищем максимальный поток в этом пути (минимальный вес ребра), прибавляем найденное значение, затем на это значение уменьшаем вес пройденных ребер и делаем обратные, затем ищем новый путь пока он есть (для проверки реализована функция Check_v), если путь не найден — функция заканчивает свою работу — возвращая при этом максимальный поток. После, смотрим исходные ребра, анализируем, и в вектор result записываем результат, выводим ответ.

first	second	
a	v = b; metrika = 3.0	v = d; metrika = 5.0
b	v = c; metrika = 1.0	
c	v = d; metrika = 1.0	
d	v = e; metrika = 1.0	

Рисунок 1 — изображение словаря Graph для примера из задания

Тестирование

1) Входные данные:

7

a

f

a b 7

a c 6

b d 6

c f 9

d e 3

d f 4

e c 2

Выходные данные:

12

a b 6

a c 6

b d 6

c f 8

d e 2

d f 4

e c 2

2) Входные данные:

3

a

c

a b 5

b c 7

a c 10

15

Выходные данные:

15

a b 5

a c 10

b c 5

3) Входные данные:

4

a

c

a b 15

b c 13

a x 5

x o 60

Выходные данные:

13

a b 13

a x 0

b c 13

x o 0

4) Входные данные:

8

a

c

a b 15

b a 15

b c 10

c b 10

a x 8

x a 8

x c 15

c x 15

Выходные данные:

18

a b 10

a x 8

b a 0

b c 10

c b 0

c x 0

x a 0

x c 8

Также были реализованы тесты:

1) Для проверки работы функции `dec_metrika` — задается граф, применяется функция `dec_metrika`, после проверяем значение `metrika` всех вершин графа (должно уменьшиться на передаваемое число).

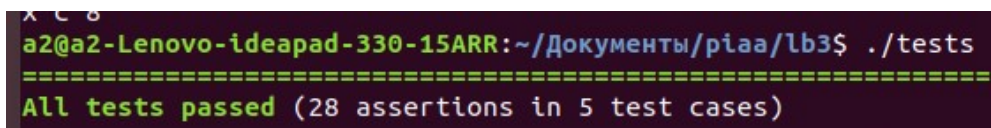
2) Для проверки функции `Search_path`, проверяем правильный ли путь находит от вершины `start` к `finish`.

3) Реализован тест для функции Search_min — которая ищет минимальный вес ребра в передаваемом пути.

4) Для проверки корректной работы функции Check_v — задается граф, с положительным весом ребер, исходящих из стартовой вершины, вызываем тестируемую функцию, ответ должен быть равен TRUE, затем, изменяем вес ребер, исходящих из стартовой вершины на 0 и тогда, при вызове функции значение должно быть равно FALSE.

5) Реализован тест для функции Algorithm — задаем граф и ищем максимальный поток.

Результаты работы написанных тестов предоставлены на рисунке 2



```
x c 8
a2@a2-Lenovo-ideapad-330-15ARR:~/Документы/p1aa/lb3$ ./tests
=====
All tests passed (28 assertions in 5 test cases)
```

Рисунок 2 — Результаты тестов программы

Разработанный программный код см. в приложении А.

Вывод.

Был изучен алгоритм Форда-Фалкерсона поиска максимального потока в сети, а также реализован и протестирован данный алгоритм на языке программирования C++.

Приложение А

Исходный код программы

Название файла: main.cpp

```
#include "lb3.hpp"

int main(){
    char start, finish;
    int N;
    std::cin>>N;
    std::cin>>start;
    std::cin>>finish;
    Graph G, G2;
    Read(G);
    G2 = G; //для хранения исходных данных
    int max_potok = Algoritm(G, start, finish);
    std::cout << max_potok << "\n";
    std::vector<answ> result = for_answer(G, G2);
    for (int i = 0; i!=result.size(); i++){
        std::cout << result[i].source << " " << result[i].dest << "
" << result[i].metrika << "\n";
    }
    return 0;
}
```

Название файла: lb3.cpp

```
#include "lb3.hpp"

void Read(Graph &G){
    char source , dest;
    float metr;
    while (std::cin >> source >> dest >> metr){
        edge g(dest, metr);
        //проверяем есть ли уже вектор с такой вершиной
        if (G.count(source) == 0) {
            std::vector<edge> d = {g};
            G[source] = d;
        } else {
            G[source].push_back(g);

            std::sort(G[source].begin(), G[source].end(), [] (edge
&p1, edge &p2) { //по возрастанию, берем с конца наименьший
                if (p1.name() < p2.name()){
                    return true;
                }
                return false;
            });
        }
    }
}

void dec_metrika(Graph &G, std::string path, int min){
```

```

//Уменьшаем вес у нынешних
for (int i = 0; i != path.size() - 1; i++){
    char source = path[i];
    char dest = path[i+1];

    edge g(dest, min);
    for (int j = 0; j != G[source].size(); j++){
        if (G[source][j].name()==dest)
            G[source][j].dec(min);
    }
}

}

int Search_min(std::string path, Graph G){
    //Ищем минимальный поток
    int min = 10000;
    for (int i = 0; i != path.size() - 1; i++){
        char source = path[i];
        char dest = path[i+1];
        for (int j = 0; j != G[source].size(); j++){
            if (G[source][j].name() == dest){
                if (min>G[source][j].len())
                    min = G[source][j].len();
            }
        }
    }
    return min;
}

void create_reverse_edges(Graph &G, std::string path, int min){
    //Делаем обратные рёбра

    std::reverse(path.begin(), path.end()); //перевернули

    for (int i = 0; i != path.size() - 1; i++){
        char source = path[i];
        char dest = path[i+1];

        edge g(dest, min);
        //проверяем есть ли уже вектор с такой вершиной
        if (G.count(source) == 0) { //если нет
            std::vector<edge> d = {g};
            G[source] = d;
        }
        else { //если есть
            int flag = 0;
            for (int j = 0; j != G[source].size(); j++){
                if (G[source][j].name() == dest){ //если есть
                    G[source][j].inc(min);
                    flag = 1;
                }
            }

            if (flag==0){ //если нет
                G[source].push_back(g);
            }
        }
    }
}

```

такое ребро

```

        std::sort(G[source].begin(), G[source].end(), []
(edge &p1, edge &p2) { //по возрастанию, берем с конца наименьший
        if (p1.name() < p2.name()){
            return true;
        }
        return false;
    });
    }

    }

}

std::string Search_path(Graph &G, char cur, char finish,
std::string path){
    while (cur!=finish){
        char now = cur;
        for (int it=0; it!= G[cur].size(); it++){

            int f4 = 1;
            for(int i =0; i!= path.size();i++){ //проверка, что
вершина не встречалась ранее
                if (G[cur][it].name() == path[i]){
                    f4 = 0;
                }

            }

            if (f4==0){
                continue;
            }

            if ((!G[cur][it].name().empty() && G[cur][it].view
== 0 && G[cur][it].len()>0) || (G[cur][it].name()==finish && G[cur]
[it].len()>0)){ //если взяли вершину
                G[cur][it].view = 1; //отмечаем как просмотренную
                cur = G[cur][it].name();
                path.push_back(cur);
                break;
            }

            else{
                G[cur][it].view = 1; //отмечаем как просмотренную
                if (it == (G[cur].size() - 1)){
                    path.pop_back();
                    cur = path.back();
                    break;
                }
            }
        }
        if (cur==now){
            if (path.size()<2){
                return path;
            }
            else{
                path.pop_back();
            }
        }
    }
}

```

```

        cur = path.back();
    }
}
return path;
}

bool Check_v(Graph G, char start){ //возвращает 0, если нет ребер,
с метрикой больше 0 из стартовой вершины
    for (int i = 0; i != G[start].size(); i++){
        if (G[start][i].len() > 0){
            return 1;
        }
    }
    return 0;
}

std::vector <answ> for_answer(Graph G, Graph G2){
    std::vector <answ> result;
    for(auto it = G2.begin(); it != G2.end(); it++){
        int l = 0;
        int l2 = 0;

        for(auto k = it->second.begin(); k!=it->second.end(); k++){
            char source = it->first;
            char dest = k->name();
            int l2 = k->len();
            int l = 0;
            for (int j = 0; j != G[source].size(); j++){
                if (G[source][j].name() == dest)
                    l = l2 - G[source][j].len();
            }
            if (l<0)
                l = 0;
            answ g(source, dest, l);
            result.push_back(g);
        }
    }

    std::sort(result.begin(), result.end(), [] (answ &p1, answ &p2)
{ //по возрастанию, берем с конца наименьший
    if (p1.source < p2.source){
        return true;
    }
    if (p1.source == p2.source){
        if (p1.dest < p2.dest)
            return true;
        else
            return false;
    }
    return false;
});
    return result;
}

int Algoritm(Graph &G, char start, char finish){

```

```

int max_potok = 0;
bool check = Check_v(G, start);
char cur = start;
std::string path;
path.push_back(cur);
while(check){
    path = Search_path(G, cur, finish, path);    //ищем путь
    if (path.empty())
        break;
    int min = Search_min(path, G); //Ищем ребро с минимальным
весом - максимальный поток пути
    max_potok = max_potok + min;
    dec_metrika(G, path, min); //Уменьшаем вес у нынешних
    create_reverse_edges(G, path, min); //Делаем обратные рёбра

    if (path.size()>1){
        path.pop_back();
        cur = path.back();
        check = Check_v(G, start);
    }

    if (path.size()==1)
        continue;

    std::string new_path;
    char source = path[0];
    char dest = path[1];
    new_path.push_back(source);
    int f3 = 1;
    for (int j = 0; j!= path.size()-1; j++){
        for(int k = 0; k!=G[source].size(); k++){
            if(G[source][k].name() == dest){
                if (G[source][k].len()>0){
                    new_path.push_back(G[source][k].name());
                    source = G[source][k].name();
                    dest = path[j+2];
                    cur = dest;
                    break;
                }
                else{
                    cur = new_path[new_path.size()-1];
                    f3 = 0;
                    break;
                }
            }
        }
        if (f3==0)
            break;
    }

    for(int z = new_path.size(); z != path.size(); z++){
        source = path[z];
        dest = path[z+1];
        for (int y = 0; y!=G[source].size(); y++){
            if (G[source][y].name() == dest){
                G[source][y].view = 0;
            }
        }
    }
}

```

```

    }
    path = new_path;
    if (path.size() >= 1){
        check = Check_v(G, start);
    }
    else{
        check = 0;
        break;
    }
}
return max_potok;
}

```

Название файла: lb3.hpp

```

#include <iostream>
#include <map>
#include <tuple>
#include <vector>
#include <algorithm>

class edge {
public:
    edge(char v, float metrika){
        this->v = v;
        this->metrika = metrika;
        this->view = 0;
    }

    char name()const{
        return this->v;
    }

    float len()const{
        return this->metrika;
    }
    void inc(int min){
        this->metrika = this->metrika+min;
    }

    void dec(int min){
        this->metrika = this->metrika-min;
    }

private:
    char v;
    float metrika;
public:
    int view;
};

class answ{
public:
    answ(char source, char dest, int metrika){
        this->source = source;
        this->dest = dest;
    }
};

```

```

        this->metrika = metrika;
    }

    public:
        char source;
        char dest;
        int metrika;
};

using Graph = std::map<char, std::vector<edge>>;

int Search_min(std::string path, Graph G);
void Read(Graph &G);
void dec_metrika(Graph &G, std::string path, int min);
void create_reverse_edges(Graph &G, std::string path, int min);
std::string Search_path(Graph &G, char cur, char finish,
std::string path);
bool Check_v(Graph G, char start);
std::vector<answ> for_answer(Graph G, Graph G2);
int Algoritm(Graph &G, char start, char finish);

```

Название файла: tests_programm.cpp

```

#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include "../source/lb3.hpp"

TEST_CASE("Тест для функции dec_metrika") {

    SECTION("1 случай") {
        char start = 'a';
        char finish = 'd';
        Graph G = {{'a', {{'b', 3}, {'c', 5}}}, {'b', {{'d', 2}}}};

        dec_metrika(G, "abd", 2);
        CHECK(G['a'][0].len() == 1);
        CHECK(G['a'][1].len() == 5);
        CHECK(G['b'][0].len() == 0);
    }

    SECTION("2 случай") {
        char start = 'a';
        char finish = 'd';
        Graph G = {{'a', {{'b', 3}, {'c', 5}}}, {'b', {{'d', 2}}},
{'c', {{'e', 7}}}, {'e', {{'d', 20}}}};

        dec_metrika(G, "aced", 5);
        CHECK(G['a'][0].len() == 3);
        CHECK(G['a'][1].len() == 0);
        CHECK(G['b'][0].len() == 2);
        CHECK(G['c'][0].len() == 2);
        CHECK(G['e'][0].len() == 15);
    }

    SECTION("3 случай") {
        char start = '1';
    }
}

```



```

        char finish = '3';
        Graph G = {{'1', {'2', 10}, {'4', 5}}, {'2', {'3', 15}},
{'4', {'5', 7}}, {'5', {'6', 2}}};

        dec_metrika(G, "123", 10);
        CHECK(G['1'][0].len() == 0);
        CHECK(G['1'][1].len() == 5);
        CHECK(G['2'][0].len() == 5);
        CHECK(G['4'][0].len() == 7);
        CHECK(G['5'][0].len() == 2);
    }
}

```

```

TEST_CASE("Тест для функции Search_path") {

```

```

    SECTION("1 случай") {

```

```

        char start = 'a';
        char finish = 'd';
        Graph G = {'a', {'b', 3}, {'c', 5}}, {'b', {'d', 2}}};

        CHECK(Search_path(G, start, finish, "a") == "abd");
    }

```

```

    SECTION("2 случай") {

```

```

        char start = 'a';
        char finish = 'd';
        Graph G = {'a', {'b', 3}, {'c', 5}}, {'b', {'d', 2}},
{'c', {'e', 7}}, {'e', {'d', 20}}};

        CHECK(Search_path(G, start, finish, "a") == "abd");
    }

```

```

    SECTION("3 случай") {

```

```

        char start = '1';
        char finish = '6';
        Graph G = {{'1', {'2', 10}, {'4', 5}}, {'2', {'3', 15}},
{'4', {'5', 7}}, {'5', {'6', 2}}};

        CHECK(Search_path(G, start, finish, "1") == "1456");
    }

```

```

}

```

```

TEST_CASE("Тест для функции Search_min") {

```

```

    SECTION("1 случай") {

```

```

        char start = 'a';
        char finish = 'd';
        Graph G = {'a', {'b', 3}, {'c', 5}}, {'b', {'d', 2}}};

        CHECK(Search_min("abd", G) == 2);
    }

```

```

    SECTION("2 случай") {

```

```

        char start = 'a';
        char finish = 'd';

```

```

        Graph G = {{'a', {'b', 3}, {'c', 5}}, {'b', {'d', 2}},
{'c', {'e', 7}}, {'e', {'d', 20}}};

        CHECK(Search_min("aced", G) == 5);
        CHECK(Search_min("abd", G) == 2);
    }

    SECTION("3 случай") {
        char start = '1';
        char finish = '3';
        Graph G = {{'1', {'2', 10}, {'4', 5}}, {'2', {'3', 15}},
{'4', {'5', 7}}, {'5', {'6', 2}}};

        CHECK(Search_min("123", G) == 10);
        CHECK(Search_min("1456", G) == 2);
    }

}

TEST_CASE("Тест для функции Check_v") {

    SECTION("1 случай") {
        char start = 'a';
        char finish = 'd';
        Graph G = {{'a', {'b', 3}, {'c', 5}}, {'b', {'d', 2}}};

        CHECK(Check_v(G, 'a') == 1);
        G['a'][0].dec(3);    //делаем длину ребер равной 0
        G['a'][1].dec(5);    //делаем длину ребер равной 0
        CHECK(Check_v(G, 'a') == 0);
    }

    SECTION("2 случай") {
        char start = 'a';
        char finish = 'd';
        Graph G = {{'a', {'b', 20}, {'c', 7}}, {'b', {'d', 2}},
{'c', {'e', 7}}, {'e', {'d', 20}}};

        CHECK(Check_v(G, 'a') == 1);
        G['a'][0].dec(20);   //делаем длину ребер равной 0
        G['a'][1].dec(7);    //делаем длину ребер равной 0
        CHECK(Check_v(G, 'a') == 0);
    }

}

TEST_CASE("Тест для функции Algoritm") {

    SECTION("1 случай") {
        char start = 'a';
        char finish = 'd';
        Graph G = {{'a', {'b', 3}, {'c', 5}}, {'b', {'d', 2}}};

        CHECK(Algoritm(G, start, finish) == 2);
    }

}

```

```

SECTION("2 случай") {
    char start = 'a';
    char finish = 'd';
    Graph G = {{'a', {'b', 20}, {'c', 7}}, {'b', {'d', 2}},
{'c', {'e', 7}}, {'e', {'d', 20}}};

    CHECK(Algoritm(G, start, finish) == 9);

}
SECTION("3 случай") {
    char start = 'a';
    char finish = 'f';
    Graph G = {{'a', {'b', 7}, {'c', 6}}, {'b', {'d', 6}},
{'c', {'f', 9}}, {'d', {'e', 3}, {'f', 4}}, {'e', {'c', 2} }};

    CHECK(Algoritm(G, start, finish) == 12);

}
}

```