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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «ПОСТРОЕНИЕ и АНАЛИЗ АЛГОРИТМОВ»

Тема: Максимальный поток

Студентка гр. 0382 _____ Кривенцова Л.С.

Преподаватель _____ Шевская Н.В.

Санкт-Петербург

2022

Цель работы.

Изучить принцип работы алгоритмов на графах (алгоритм Форда-Фалкерсона). Применить знания на практике, решив поставленную задачу.

Основные теоретические положения.

Алгоритм Форда — Фалкерсона решает задачу нахождения максимального потока в транспортной сети. Идея алгоритма 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.

Задание.

Максимальный поток.

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).

Вар. 3. Поиск в глубину. Рекурсивная реализация.

Выполнение работы.

Структура ребра

struct edge{

char r1; - вершина из которой идёт ребро.

char r2; - вершина, заканчивающая ребро.

int weight; - расстояние между вершинами.

int real_w; - фактическая величина протекающего потока.

bool direction; - направление, по которому идёт путь.

bool open_forward; - возможность пройти по ребру в его направлении.

bool open_back; - возможность пройти по ребру в обратном направлении.

};

vector<edge> current = {}; - массив текущего пути.

vector<edge> graph= {}; - массив рёбер (граф).

char start, finish; - символы начала и конца пути.

bool continiud = true; - логическая переменная, означающая нужно ли продолжать рекурсивный обход графа.

int result = 0; - величина максимального потока.

wt = 0; - наименьшая пропускная способность в рассматриваемом пути.

Функция *mysort*() сортирует массив рёбер пузырьком в лексикографическом порядке по первой вершине, потом по второй.

Функция *Collision(char node)* проверяет, есть ли вершина, в которую собираемся пойти, в уже найденном пути. Если есть, возвращает *false*, иначе *true*. Принимает на вход такую вершину в виде символа.

Функция *minimum_curr_weight()* для поиска наименьшей пропускной способности в текущем пути. Не принимает ничего в качестве аргумента, возвращает такую величину (*wt*);

Функция *into_index(char node*) принимает на вход в качестве аргумента символьное представление вершины, путь из которой нужно найти. Цель функции – поиск следующей вершины, в которую следует пойти. Приоритет отдаётся вершине, стоящей в алфавите первее всех из вершин, имеющих связь с node (входят или выходят из неё). Проверяется, чтобы найденная вершина соответствовала требованиям: чтобы её вес был положительный, вершина ещё не была включена в данный путь, была открыта (возможность идти через неё). Если вершина, в которую следует пойти, не была найдена, функция возвращает значение -1. Иначе, возвращает индекс искомой вершины в графе.

int recurs(char node) – функция рекурсивного обхода графа в глубину. В функции происходит расчёт переменной *result*, а также высчитываются фактические величины протекающего потока через вершины.

В функции *main*() производится считывание данных и вместе с этим инициализируются поля структур рёбер графа, вызываются функции сортировки и функция, реализующая рекурсивно алгоритм Форда-Фалкерсона, затем полученные данные (величина максимального потока и рёбра графа с фактической величиной протекающего потока) выводятся на экран.

Задание по вариантам.

Алгоритм реализован рекурсивным поиском в глубину.

Тестирование.

Тестирование проводилось с помощью встроенной библиотеки <*cassert*>. Результат не отображён в таблицах, так как команда *assert* предназначена для отлова ошибок, и выводит результат на экран только в случае провала теста.

Были написаны 5 проверяющий функций.

Работа assert’а заключается в следующем*:* в тесте ест создаюся новые исходные данные (в данном случае, задавая граф). И assert’у передаётся результат функции и предполагаемый (верный) результат. При несовпадении на экран выводится сообщение. Пример:

Assertion failed: find('e') == 5, file

C:/Users/Serg/CLionProjects/untitled1/main.cpp, line 39

Тест *TestMysort()* проверяет функцию *mysort*() следующим образом:

Задаются данные несортированного графа, вызывается функция *mysort(),* затем с помощью assert проверяется порядок элементов в графе.

Тест *TestMinimum_curr_weight()* проверяет нахождение минимального веса в текущем пути (*minimum_curr_weight():* задаётся массив вершин *current*, и в assert передаётся равенство результата работы функции и истинный наименьший вес из *current*.

В функции *TestCollision*() задаётся задаётся массив вершин *current* (перед этим он очищается с помощью .erase, которому передаются адреса начала и конца массива) и assert сравнивает результат работы функции *Collision(char node),* которая отвечает за проверку, не включён ли уже элемент в данный массив, и правильный ответ.

В тесте *TestInto(),* проверяемом нахождение следующей вершины (проходя в алфавитном порядке), перед тем как сравнить найденные индексы с верными ответами, аннулируются массивы графа и текущего пути, и граф задаётся новыми данными, в то время когда *current* остается пустым *(current = {}).*

Аналогично работает функция *TestRecurs(),* проверяющая основной алгоритм.

В конце каждого теста есть вывод строки – результата. Ведь если assert не вызвал ошибку, то программа выполняется дальше, и печатается строка, что определённый тест пройден.

Результат:

Function mysort: Test OK

Function minimum_curr_weight: Test OK

Function Collision: Test OK

Function into_index: Test OK

Process finished with exit code 0

Выводы.

В результате работы была написана программа, решающая поставленную задачу при использовании изученных теоретических материалов. Программа было протестирована, результаты тестов удовлетворительны.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <cassert>
using namespace std;

struct edge{
    char r1;
    char r2;
    int weight;
    int real_w;
    bool direction;
    bool open_forward;
    bool open_back;
};

vector<edge> current = {};
vector<edge> graph = {};
char start, finish;
bool continiud = true, sign = false;
int result = 0, wt = 0;

void mysort(){
    edge temp;
    for (int i = 0; i < graph.size() - 1; i++) {
        for (int j = 0; j < graph.size() - i - 1; j++) {
            if (int(graph[j].r1) > int(graph[j + 1].r1)) {
                temp = graph[j];
                graph[j] = graph[j + 1];
                graph[j + 1] = temp;
            }
            else if (int(graph[j].r1) == int(graph[j + 1].r1)){
                if (int(graph[j].r2) > int(graph[j + 1].r2)) {
                    temp = graph[j];
                    graph[j] = graph[j + 1];
                    graph[j + 1] = temp;
                }
            }
        }
    }
}
void TestMysort(){
    graph.erase(graph.begin(), graph.end());
    graph = {{'n','x'},
            {'n','g'},
             {'d', 'r'}};
    mysort();
    assert(graph[0].r1 == 'd' and graph[0].r2 == 'r');
    assert(graph[1].r1 == 'n' and graph[1].r2 == 'g');
```

```
    assert(graph[2].r2 == 'x');
    graph.push_back({'a','b'});
    graph.push_back({'y','x'});
    mysort();
    assert(graph[graph.size()-1].r1 == 'y' and graph[graph.size()-1].r2 == 'x');
    cout << endl << "Function mysort: Test OK" << endl;
}

int minimum_curr_weight(){
    int min;
    if (current[0].direction) min = current[0].weight;
    else min = current[0].real_w;
    for (int i = 1; i < current.size(); i++){
        if (current[i].direction and current[i].weight < min){
            min = current[i].weight;
        }

        if  (!current[i].direction and current[i].real_w < min){
            min = current[i].real_w;
        }
    }
    return min;
}

void TestMinimum_curr_weight(){
    current.erase(current.begin(), current.end());
   current = {{'n','x', 35, 51, true, true, true}
    ,{'n','x', 95, 81, true, true, true},
    {'m', 'w', 33, 46, true, true, true}};
    assert(minimum_curr_weight() == 33);
    current.push_back({'a','n', 4, 7, true, true, true});
    assert(minimum_curr_weight() == 4);
    cout << endl << "Function minimum_curr_weight: Test OK" << endl;
}

bool Collision(char node){
    for (int i = 0; i < current.size(); i++){
        if (current[i].r1 == node or current[i].r2 == node){
            return false;
        }
    }
    return true;
}
void TestCollision(){
    current.erase(current.begin(), current.end());
    current = {{'a', 'b'},{'b','c'},{'c','d'}};
    assert(Collision('k'));
    assert(Collision('m'));
    current.push_back({'d','m'});
    assert(!Collision('c'));
    assert(!Collision('m'));
    assert(!Collision('a'));
```

```
    cout << endl << "Function Collision: Test OK" << endl;
}

int into_index(char node){
    int minindex = -1; int t = 0, min = -1;
    for (int i = 0; i < graph.size(); i++)
        min = min + static_cast<int>(graph[i].r1) +
static_cast<int>(graph[i].r2);
    for (t = 0; t <= graph.size(); t++){
        if ((graph[t].r1 == node and Collision(graph[t].r2))
                and graph[t].open_forward and graph[t].weight > 0
                and static_cast<int>(graph[t].r2) < min){
            minindex = t;
            min = static_cast<int>(graph[t].r2);
        }
        else if (graph[t].r2 == node and Collision(graph[t].r1)
                 and graph[t].open_back
                 and graph[t].real_w > 0
                 and static_cast<int>(graph[t].r1) < min) {
            minindex = t;
            min = static_cast<int>(graph[t].r1);
        }
    }
    if (minindex < 0) return -1;
    if (graph[minindex].r2 == node) graph[minindex].direction = false;
    return minindex;
}

void TestInto(){
    graph.erase(graph.begin(), graph.end());
    current.erase(current.begin(), current.end());
    current = {};
    graph = {{'a', 'b', 3,1, true, true},
             {'b','c',2, 0, true,true,true},
             {'a','c',1,1, true,true, true}};
    assert(into_index('a') == 0);
    assert(into_index('c') == 2);
    graph.push_back({'a','a',3});
    assert(into_index('m') == -1);
    graph.push_back({'d','e',0,0});
    assert(into_index('e') == -1);
    cout << endl << "Function into_index: Test OK" << endl;
}

void recurs(char node){
    edge curredge;
    int index;
    if (node == start) continiud = true;
    else continiud = false;
    do{
        index = into_index(node);
        if (index < 0) {
```

```
            sign = true;
            return;
        }
        curredge = graph[index];
        if (curredge.weight != 0 and curredge.r1 == node
                                    or curredge.real_w != 0 and curredge.r2 ==
node){
            current.push_back(curredge);
            if (curredge.r1 == finish or curredge.r2 == finish){
                wt = minimum_curr_weight();
                result += wt;
            }
            else {
                if (curredge.r1 == node) recurs(curredge.r2);
                else if (curredge.r2 == node) {
                    recurs(curredge.r1);
                }
                if (sign) {
                    if (graph[index].direction) {graph[index].open_forward =
false;
                        curredge.open_forward = false;}
                    else {
                        graph[index].open_back = false;
                        curredge.open_back = false;
                    }
                    sign = false;
                }
            }
            if (curredge.r1 == node) {
                graph[index].weight -= wt;
                graph[index].real_w += wt;
                curredge.weight -= wt;
                curredge.real_w += wt;
            }
            else if (curredge.r2 == node) {
                graph[index].weight += wt;
                graph[index].real_w -= wt;
                curredge.weight += wt;
                curredge.real_w -= wt;
                graph[index].direction = true;
            }
            current.pop_back();
        }
        if (node == start) {
            continiud = true;
            wt = 0;
        }
    }
    while (curredge.r1 != finish and curredge.r2 != finish and !sign and
           continiud);
}
```

```
void TestRecurs(){
    graph.erase(graph.begin(), graph.end());
    current.erase(current.begin(), current.end());
    current = {};
    start = 'a', finish = 'c';
    continiud = true; sign = false;
    result = 0; wt = 0;
    graph = {{'a', 'b', 20,0, true, true, true},
             {'a','d',40, 0, true,true,true},
             {'a','e',30,0, true,true, true},
             {'e', 'd', 50,0, true, true},
             {'e','c',40, 0, true,true,true},
             {'d','b',20,0, true,true, true},
             {'d', 'c', 30,0, true, true},
             {'b','c',30, 0, true,true,true}
    };
    mysort();
    recurs(start);
    assert(result == 90);
    graph.erase(graph.begin(), graph.end());
    current.erase(current.begin(), current.end());
    current = {};
    start = 'a', finish = 'e';
    graph = {{'a', 'b', 20,0, true, true},
             {'b','a',20, 0, true,true,true},
             {'a','d',10,0, true,true, true},
             {'d', 'a', 10,0, true, true},
             {'a','c',30, 0, true,true,true},
             {'c','a',30,0, true,true, true},
             {'b', 'c', 40,0, true, true},
             {'c','b',40, 0, true,true,true},
             {'c', 'd', 20,0, true, true},
             {'d','c',20, 0, true,true,true},
             {'c','e',20,0, true,true, true},
             {'e', 'c', 20,0, true, true},
             {'b','e',30, 0, true,true,true},
             {'e','b',30,0, true,true, true},
             {'d', 'e', 10,0, true, true},
             {'e','d',10, 0, true,true,true}
    };
    result = 0; wt = 0;
    mysort();
    recurs(start);
    assert(result == 60);
    cout << endl << "Function recurs: Test OK" << endl;
}

int main() {
    int count = 0;
    std::cin >> count >> start >> finish;
    for (int i = 0; i < count; i++){
        edge a;
```

```
        std::cin >> a.r1 >> a.r2 >> a.weight;
        a.real_w = 0;
        a.direction = true;
        a.open_forward = true;
        a.open_back = true;
        graph.push_back(a);
    }
    mysort();
    recurs(start);
    cout << result << endl;
    for (int t = 0; t < graph.size();t++){
        cout << graph[t].r1 << ' ' << graph[t].r2
             << ' ' << graph[t].real_w << endl;
    }
    TestMysort();
    TestMinimum_curr_weight();
    TestCollision();
    TestInto();
    TestRecurs();
    return 0;
}
```