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

Студент гр. 8303 _____ Колосова М.П.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы

Изучение алгоритма Форда-Фалкерсона для поиска максимального потока в сети.

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 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

Индивидуализация

Поиск не в глубину и не в ширину, а по правилу: каждый раз выполняется переход по дуге, соединяющей вершины, имена которых в алфавите ближе всего друг к другу. Если таких дуг несколько, то выбрать ту, имя конца которой в алфавите ближе к началу алфавита.

Описание алгоритма

Величина потока итеративно увеличивается посредством поиска увеличивающего пути в остаточной сети. Увеличивающий путь — это путь от истока к стоку, вдоль которого можно послать ненулевой поток. Поиск этого пути в программе реализован как модифицированный обход в глубину, переход к следующей вершине при обходе происходит по условию индивидуализации, так как вектор смежных вершин отсортирован в нужном порядке. После

остаточная сеть перестраивается, а к величине максимального потока прибавляется значение максимальной пропускной способности пути. Процесс повторяется, пока можно найти увеличивающий путь.

Сложность алгоритма

В каждой итерации происходит поиск увеличивающего пути с помощью модифицированного обхода в глубину. Смежные вершины хранятся в отсортированном виде, поэтому условие индивидуализации не усложняет алгоритм и сложность поиска пути составит o(V+E), где V - вершины, E - ребра.

Пусть F — величина максимального потока, тогда можно оценить худший случай работы. Если каждая итерация будет увеличивать искомую величину на единицу, потребуется F вызовов. Значит, общая сложность — o(F*(V+E)).

Сложность алгоритма по памяти:

Для поиска пути используется матрица смежности, ее максимальный размер — V*V, V — количество вершин. Так же в памяти хранится исходный граф как вектор ребер, его максимальный размер — E, E — число ребер. Самый объемный по памяти случай создает для каждого ребра обратное в остаточной сети, тогда наибольшая возможная длина — 2E. Таким образом, получим квадратичную зависимость от вершин и линейную от ребер - o(V*V + 3E).

Описание структур данных

static vector<pair<char, char>> graph; - вектор пар, используется для хранения исходного графа как списка ребер

static map<char, map<char, Edge>> residualNet; - двумерный ассоциативный контейнер, используется для хранения ребер остаточной сети

static map<char, vector<char>> adjacentVertexMap; - контейнер ключ-значение, используется для хранения соседей каждой вершины

class Edge; - класс для хранения остаточной пропускной способности ребра и фактического потока по вспомогательному ребру

map<char, bool> checked; - контейнер ключ-значение, используется для отметок о посещении вершин при обходе

Описание функций

void **inputGraph**(size_t N); - функция считывания входного потока данных о графе, создает начальную остаточную сеть и заполняет информацию о смежных вершинах; в качестве аргумента принимает количество ребер, которое необходимо считать

int **findPath**(int bandwidth, char cur, map<char, bool> check, char stock); - функция поиска пути от истока к стоку, реализована как модификация обхода в глубину (переход выполняется по дуге, стягивающей ближайшие вершины)
Аргументы функции:
bandwidth — минимальная пропускная способность ребер, которые уже посещены
cur — текущая вершина обхода
check — контейнер, содержащий информацию о посещенных вершинах
stock — конечная вершина пути, сток графа
Возвращает максимально возможную величину потока, которую можно пустить по увеличивающему пути

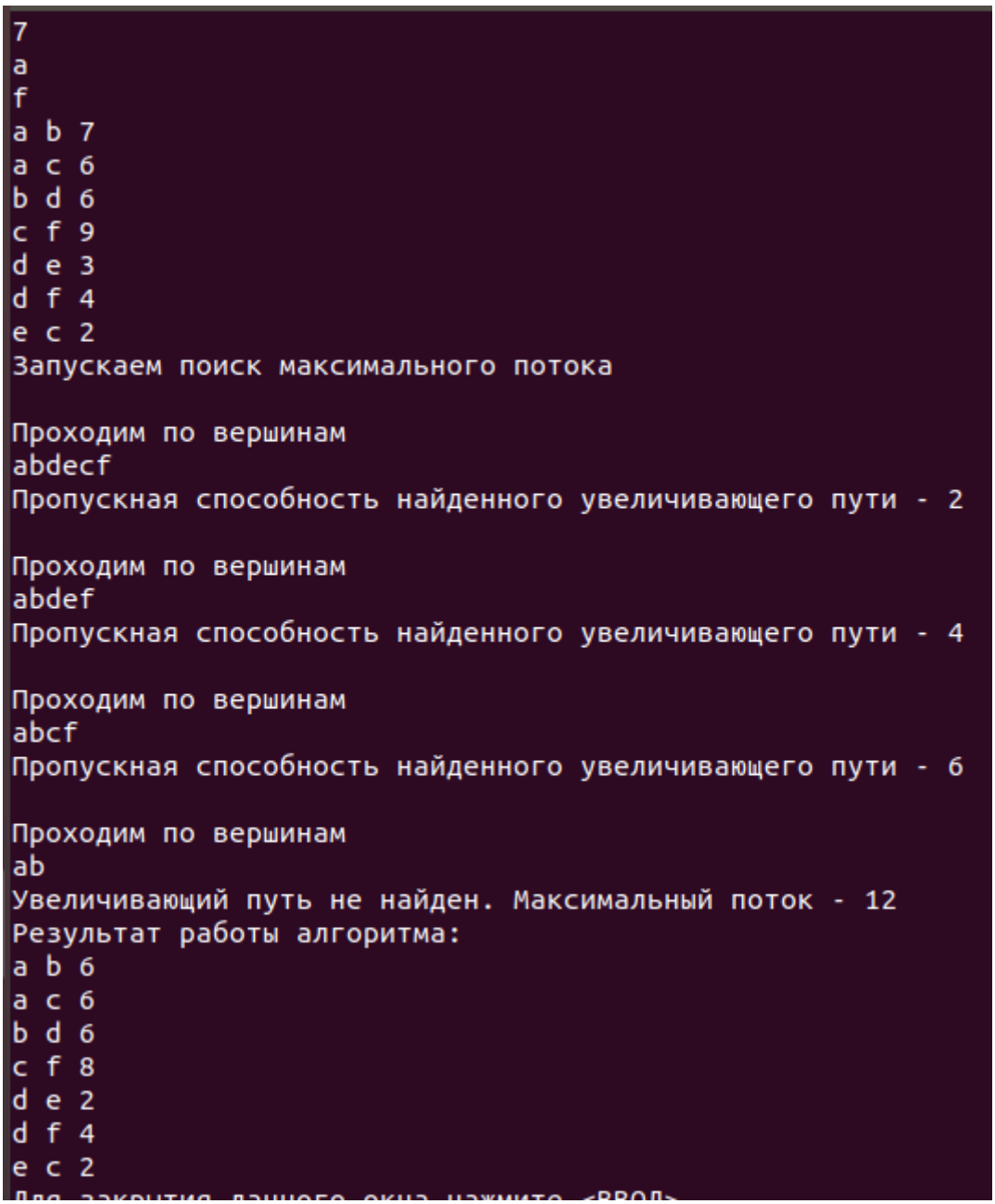
int **findMaxFlow**(char source, char stock); - функция поиска максимального потока по алгоритму Форда-Фалкерсона
Аргументы:
source — стартовая вершина обхода, исток
stock — финальная вершина обхода, сток
Возвращает нулевое значение при заверщении.

void **printGraph**(); - функция печати результата, выводит отсортированные ребра графа и фактический поток, протекающий по ним

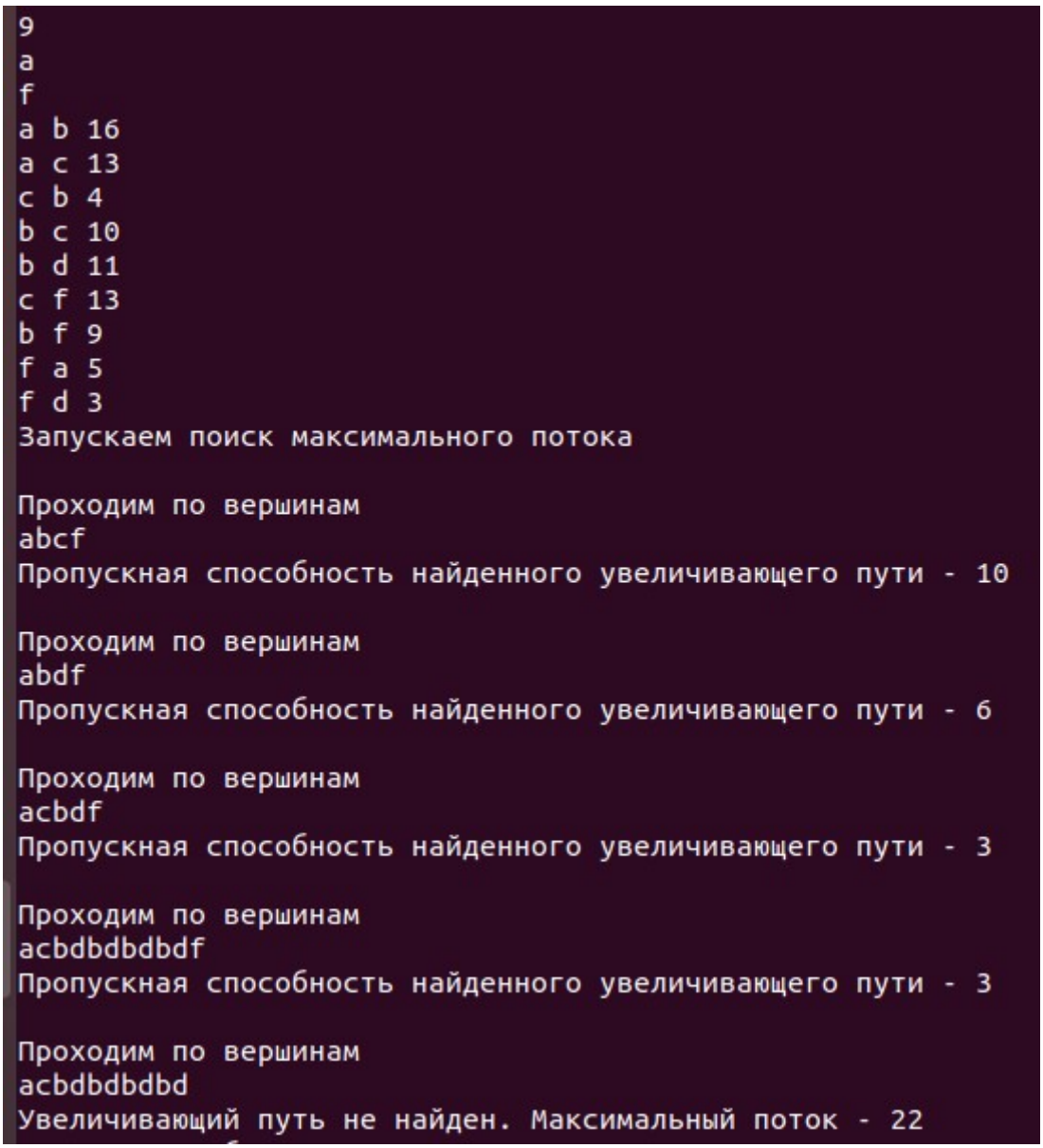
Тестирование

```
6
a
f
a b 16
a c 13
c b 4
b c 10
b d 11
b f 12
Запускаем поиск максимального потока

Проходим по вершинам
abccdf
Пропускная способность найденного увеличивающего пути - 12

Проходим по вершинам
abccdcbdbd
Увеличивающий путь не найден. Максимальный поток - 12
Результат работы алгоритма:
a b 12
a c 0
b c 0
b d 0
b f 12
c b 0
```

```
4
a
f
a b 4
b a 4
b f 4
a f 7
Запускаем поиск максимального потока

Проходим по вершинам
abf
Пропускная способность найденного увеличивающего пути - 4

Проходим по вершинам
af
Пропускная способность найденного увеличивающего пути - 7

Проходим по вершинам
a
Увеличивающий путь не найден. Максимальный поток - 11
Результат работы алгоритма:
a b 4
a f 7
b a 0
b f 4
```


7
a
f
a b 7
a c 6
b d 6
c f 9
d e 3
d f 4
e c 2
Запускаем поиск максимального потока
Проходим по вершинам
abdecf
Пропускная способность найденного увеличивающего пути - 2
Проходим по вершинам
abdef
Пропускная способность найденного увеличивающего пути - 4
Проходим по вершинам
abcf
Пропускная способность найденного увеличивающего пути - 6
Проходим по вершинам
ab
Увеличивающий путь не найден. Максимальный поток - 12
Результат работы алгоритма:
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2

```
5
a
f
a b 7
a c 3
a f 5
c b 6
b f 8
Запускаем поиск максимального потока

Проходим по вершинам
abf
Пропускная способность найденного увеличивающего пути - 7

Проходим по вершинам
acbf
Пропускная способность найденного увеличивающего пути - 1

Проходим по вершинам
acbf
Пропускная способность найденного увеличивающего пути - 5

Проходим по вершинам
acb
Увеличивающий путь не найден. Максимальный поток - 13
Результат работы алгоритма:
a b 7
a c 1
a f 5
b f 8
c b 1
```


9
a
f
a b 16
a c 13
c b 4
b c 10
b d 11
c f 13
b f 9
f a 5
f d 3
Запускаем поиск максимального потока
Проходим по вершинам
abcf
Пропускная способность найденного увеличивающего пути - 10
Проходим по вершинам
abdf
Пропускная способность найденного увеличивающего пути - 6
Проходим по вершинам
acbdf
Пропускная способность найденного увеличивающего пути - 3
Проходим по вершинам
acbdbdbdbdf
Пропускная способность найденного увеличивающего пути - 3
Проходим по вершинам
acbdbdbdbd
Увеличивающий путь не найден. Максимальный поток - 22

```
Увеличивающий путь не найден. Максимальный поток - 22
Результат работы алгоритма:
a b 16
a c 6
b c 7
b d 0
b f 9
c b 0
c f 13
f a 0
f d 0
```

Вывод

В ходе лабораторной работы была написана программа, в которой находится максимальный поток всей сети и фактический поток каждого ребра исходного графа. Был реализован особый поиск пути с выбором ребра, соединяющего ближайшие вершины. Таким образом, был изучен алгоритм Форда-Фалкерсона.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД

```
#include <iostream>
#include <vector>
#include <map>
#include<set>
#include <algorithm>
#include <climits>

using namespace std;

void inputGraph(size_t N);//чтение и запись графа и сети
int findPath(int bandwidth, char cur, map<char, bool> check, char stock);//поиск
дополняющего пути
int findMaxFlow(char source, char stock);//поиск максимального потока
void printGraph();//печать результата

class Edge{

public:
    //остаточная пропускная способность ребра
    int maxBandwidth;
    //фактический поток по обратному ребру
    int factBandwidth;

    Edge(): maxBandwidth(0), factBandwidth(0){}
    Edge(int maxBandwidth){
        this->maxBandwidth = maxBandwidth;
        this->factBandwidth = 0;
    }

};

static vector<pair<char, char>> graph;//граф
static map<char, map<char, Edge>> residualNet;//остаточная сеть
static map<char, vector<char>> adjacentVertexMap;// карта смежных вершин

int main()
{
    size_t N;
    char source, stock;
    cin>>N;
    cin>>source;
    cin>>stock;

    inputGraph(N);
    cout<<"Запускаем поиск максимального потока"<<endl;
    findMaxFlow(source, stock);
    cout<<"Результат работы алгоритма:"<<endl;
    printGraph();

    return 0;
}

void inputGraph(size_t N){
```

```
    for(size_t i = 0; i < N; i++){
        char v1, v2;
        int bandwidth;//пропускная способность
        cin>>v1>>v2>>bandwidth;

        graph.push_back(pair<char, char>(v1, v2));
        residualNet[v1][v2] = Edge(Edge(bandwidth));
        adjacentVertexMap[v1].push_back(v2);
        adjacentVertexMap[v2].push_back(v1);
    }

    //сортировка графа в лексикографическом порядке
    sort(graph.begin(), graph.end(), [](pair<char, char> a, pair<char, char> b){
             if(a.first == b.first)
                 return a.second < b.second;
             return a.first < b.first;
         });

    for(auto it : adjacentVertexMap){
        //сортируем смежные вершины по близости к начальной
        char startV = it.first;
        sort(it.second.begin(), it.second.end(), [&startV](const char &a, const
char &b) -> bool{
                 return abs(a - startV) < abs(b - startV);
             });
    }
    return;
}

int findPath(int bandwidth, char cur, map<char, bool> check, char stock){

    if(check[cur])
        return  0;
    check[cur] = true;
    cout<<cur;
    //текущая вершина - сток
    if(cur == stock)
        return bandwidth;

    //обход соседей
    for(auto &next : adjacentVertexMap[cur]){

        int factBandwidth = residualNet[cur][next].factBandwidth;
        int maxBandwidth = residualNet[cur][next].maxBandwidth;

        if(factBandwidth > 0){
            int newBandwidth = findPath(min(bandwidth, factBandwidth), next,
check, stock);
            if(newBandwidth > 0){
                //изменяем остаточную пропускную способность и увеличиваем поток
по обратному ребру
                residualNet[next][cur].maxBandwidth += newBandwidth;
                residualNet[cur][next].factBandwidth -= newBandwidth;
                //возвращаем минимальную пропускную способность пути
                return  newBandwidth;
            }
        }

        if(maxBandwidth > 0){
            int newBandwidth = findPath(min(bandwidth, maxBandwidth), next,
check, stock);
```

```
            if(newBandwidth > 0){
                //изменяем остаточную пропускную способность и увеличиваем поток
по обратному ребру
                residualNet[next][cur].factBandwidth += newBandwidth;
                residualNet[cur][next].maxBandwidth -= newBandwidth;
                //возвращаем минимальную пропускную способность пути
                return newBandwidth;
            }
        }

    }
    return 0;

}

int findMaxFlow(char source, char stock){

    int flow = 0, maxFlow = 0;

    for(;;){
        map<char, bool> checked;
        //поиск увеличивающего пути
        cout<<endl<<"Проходим по вершинам"<<endl;
        flow = findPath(INT_MAX, source, checked, stock);
        if(flow == 0 || flow == INT_MAX){//путь не найден
            cout<<endl<<"Увеличивающий путь не найден. ";
            cout<<"Максимальный поток - ";
            cout<<maxFlow<<endl;
            return 0;
        }
        //увеличение макисмального потока на данной итерации
        cout<<endl<<"Пропускная способность найденного увеличивающего пути - ";
        cout<<flow<<endl;
        maxFlow += flow;
    }

}

void printGraph(){

    for(auto const &elem : graph){
        cout<<elem.first<<" "<<elem.second<<" "<<residualNet[elem.second]
[elem.first].factBandwidth<<endl;
    }

}
```