

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
Вариант 6

Студентка гр. 8303

Потураева М.Ю.

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

$v_i v_j w_{ij}$ – ребро графа

$v_i v_j w_{ij}$ – ребро графа

...

Выходные данные:

P_{max} – величина максимального потока

$v_i v_j w_{ij}$ – ребро графа с фактической величиной протекающего потока

$v_i v_j w_{ij}$ – 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).

Пример входных данных

7

a

f

a b 7

a c 6

b d 6

c f 9

d e 3

d f 4

e c 2

Пример выходных данных

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

Изначально заполняются структуры для хранения графа. После чего производится поиск соседних вершин по правилу индивидуализации.

Поиск соседних вершин начинается с истока. Переход в соседнюю вершину может быть осуществлен, только если она ранее не была просмотрена и остаточная пропускная способность до неё больше нуля. К тому же выбирается соседняя вершина, у которой разность символов этой и рассматриваемой вершины минимальная. В случае равенства разностей выбирается вершина, имя которой находится ближе к началу алфавита.

При нахождении пути до стока, находится ребро с минимальной пропускной способностью и происходит пересчет пропускных способностей пути. Пересчет заключается в вычитании минимальной пропускной способности из пропускных способностей всех рёбер пути, начиная с истока, а в обратном направлении она прибавляется. Также увеличивается величина итогового максимального потока.

Если на какой-то итерации цикла соседние вершины найдены не будут, и алгоритм вернётся к истоку, следовательно, сквозных путей в графе больше построить нельзя. Если же соседних вершин нет, но текущая вершина не исток, то производится откат к предыдущей вершине.

При выводе итогового результат высчитывается фактический поток через каждое ребро, который равен разности первоначальной и конечной пропускных способностей. Если фактический поток отрицательный, и ребро было задано изначально, то принимаем его значение равным нулю.

Сложность алгоритма по операциям: $O(E * F)$, где E – число ребер в графе, F – максимальный поток.

Сложность алгоритма по памяти: $O(N + E)$, где N – количество вершин, E – количество ребер.

Описание структур данных.

1.

```
struct Neighbour {  
    char symbol; //вершина  
    int bandwidth; //введенная пропускная способность  
    int cur_bandwidth; //промежуточная пропускная способность  
    int final_bandwidth; //итоговая пропускная способность
```

```

        Neighbour(char symbol, int bandwidth) : symbol(symbol),
        bandwidth(bandwidth), cur_bandwidth(bandwidth) {}
        Neighbour(char symbol) : symbol(symbol) {}
};

```

Структура используется для хранения информации о соседних вершинах.

Neighbour(char symbol, int bandwidth)- конструктор, который используется для инициализации полей symbol, bandwidth, cur_bandwidth.

Neighbour(char symbol))- конструктор, который используется для инициализации поля symbol.

2.

```

struct Top {
    char symbol; //вершина
    char prevTop; //предыдущая вершина
    vector <Neighbour> neighbours; //вектор соседей
    Top() {}
    Top(char symbol) : symbol(symbol) {}
};

```

Структура для хранения информации о вершине и ее соседях.

Top()- конструктор без инициализации полей.

Top(char symbol)- конструктор, который используется для инициализации поля symbol.

3. vector <Top> tops – вектор для хранения вершин графа.

4. vector <Top> viewingTop – вектор для хранения просмотренных вершин.

Описание функций

1) bool cmp_top(const Top& first, const Top& second) – компаратор для сравнения вершин графа.

Функция возвращает true или false, в зависимости от того, какая вершина больше.

first-первая вершина

second- вторая вершина

2) `bool cmp_neigh(const Neighbour& first, const Neighbour& second)` – компаратор для сравнения соседних вершин.

Функция возвращает true или false, в зависимости от того, какая из соседних вершин больше.

first-первая соседняя вершина

second- вторая соседняя вершина

3) `bool isExist(vector<Top> vector, char tmp)` – функция для проверки вхождения вершины в рассматриваемый вектор.

Функция возвращает true или false, в зависимости от того, найдена вершина или нет.

vector- вектор вершин

tmp- вершина

4) `int findIndexTop(vector<Top> vector, char tmp)`- функция, которая находит индекс переданной вершины.

Функция возвращает индекс нужной вершины, или же -1, если такая вершина отсутствует.

vector-вектор вершин, в котором производится поиск

tmp-вершина, которую необходимо найти

5) `int findIndexNeigh(Top top, char tmp)`- функция, которая находит индекс вершины среди соседей другой вершины.

Функция возвращает индекс нужной соседней вершины, или же -1, если такая вершина отсутствует.

top-вершина, по соседям которой производится поиск

tmp-вершина, которую необходимо найти

6) `void init_Graph(size_t N, vector <Top>& tops)`-инициализация графа.

N-количество ребер в графе

Tops-вектор для хранения вершин графа

7) `int maxFlowCount(const vector <Top>& tops, Top estr)`- функция для нахождения минимальной пропускной способности.

Функция возвращает минимальную пропускную способность одного из ребер графа.

tops-вектор вершин графа

estr-сток

8) void recountFlow(vector <Top>& tops, Top estr, int maxFlow)- функция для пересчета пропускных способностей ребер графа.

tops-вектор вершин графа

estr-сток

maxFlow-минимальная пропускная способность в сети

tops-вектор для хранения вершин

9) void Priority(int& index, Top cur ,const vector <Top>& viewingTop) – функция, для выбора вершин согласно приоритету индивидуализации.

index-индекс вершины с наилучшим приоритетом

cur-рассматриваемая вершина

viewingTop-вектор просмотренных вершин

10) int FordFalk(char source, char estr, vector <Top>& tops, vector <Top>& viewingTop) –функция, реализующая алгоритма Форда-Фалкерсона.

Функция возвращает значение максимального потока в сети.

source-исток

estr-сток

tops-вектор для хранения вершин

viewingTop-вектор просмотренных вершин

ТЕСТИРОВАНИЕ.

1.

```
Enter information about the graph:
```

```
7
```

```
a
```

```
f
```

```
a b 7
```

```
a c 6
```

```
b d 6
```

```
c f 9
```

```
d e 3
```

```
d f 4
```

```
e c 2
```

```
Beginning of the algorithm:
```

```
Viewing Tops:a
```

```
Priority vertex:b
```

```
i=0
```

```
i=1
```

```
Viewing tops:a b
```

```
Priority vertex:d
```

```
i=0
```

```
i=1
```

```
i=2
```

```
Viewing tops:a b d
```

```
Priority vertex:e
```

```
i=0
```

```
i=1
```

```
i=2
```

```
i=3
```

```
Viewing tops:a b d e
```

```
Priority vertex:c
```

```
i=0
```

```
i=1
```

```
i=2
```

```
i=3
```

```
i=4
```

```
Viewing tops:a b d e c
```

```
Priority vertex:f
```



```

i=0
i=1
i=2
i=3
i=4
i=5
Viewing tops:a b d e c f
The current vertex is a estuary.

Calculating the maximum flow.
Current min:10000
Current flow:c->f 9
New min:9
Current min:9
Current flow:e->c 2
New min:2
Current min:2
Current flow:d->e 3
Current min:2
Current flow:b->d 6
Current min:2
Current flow:a->b 7
The end of the calculating the maximum flow.

Start of recalculation flow:
Current top:f
The path from f to c was not found, creating an edge.
New bandwidth previous top=9-2=7
Current top:c
The path from c to e was not found, creating an edge.
New bandwidth previous top=2-2=0
Current top:e
The path from e to d was not found, creating an edge.
New bandwidth previous top=3-2=1
Current top:d
The path from d to b was not found, creating an edge.
New bandwidth previous top=6-2=4
Current top:b
The path from b to a was not found, creating an edge.
New bandwidth previous top=7-2=5
Recount final maximum flow:0+2=2
Viewing tops is cleared.
Viewing Tops:a
Priority vertex:b

```

```

i=0
i=1
Viewing tops:a b
Priority vertex:d

i=0
i=1
i=2
Viewing tops:a b d
Priority vertex:e

i=0
i=1
i=2
i=3
Viewing tops:a b d e
Priority vertex was not found
Priority vertex:f

i=0
i=1
i=2
i=3
i=4
Viewing tops:a b d e f
The current vertex is a estuary.

Calculating the maximum flow.
Current min:10000
Current flow:d->f 4
New min:4
Current min:4
Current flow:b->d 4
Current min:4
Current flow:a->b 5
The end of the calculating the maximum flow.

Start of recalculation flow:
Current top:f
The path from f to d was not found, creating an edge.
New bandwidth previous top=4-4=0
Current top:d
The path from d to b was found.
New bandwidth current top=2+4=6
New bandwidth previous top=4-4=0
Current top:b
The path from b to a was found.
New bandwidth current top=2+4=6

```

```

Viewing tops is cleared.
Viewing Tops:a
Priority vertex:b

i=0
i=1
Viewing tops:a b
Priority vertex was not found
Priority vertex:c

i=0
i=1
i=2
Viewing tops:a b c
Priority vertex:e

i=0
i=1
i=2
i=3
Viewing tops:a b c e
Priority vertex:d

i=0
i=1
i=2
i=3
i=4
Viewing tops:a b c e d
Priority vertex was not found
Priority vertex was not found
Priority vertex:f

i=0
i=1
i=2
i=3
i=4
i=5
Viewing tops:a b c e d f
The current vertex is a estuary.

Calculating the maximum flow.
Current min:10000
Current flow:c->f 7
New min:7
Current min:7
Current flow:a->c 6
New min:6

```

```

Priority vertex:d
Viewing Tops:a b c e d
Priority vertex was not found
Priority vertex was not found
Priority vertex:f
Viewing Tops:a b c e d f
The current vertex is a estuary.
Calculating the maximum flow.
Current min:10000
Current flow:c->f 7
New min:7
Current min:7
Current flow:a->c 6
New min:6
The end of the calculating the maximum flow.
Start of recalculation flow:
Current top:f
The path from f to c was found.
New bandwidth current top=2+6=8
New bandwidth previous top=7-6=1
Current top:c
The path from c to a was not found, creating an edge.
New bandwidth previous top=6-6=0
Recount final maximum flow:6+6=12
Viewing tops is cleared.
Viewing Tops:a
Priority vertex:b
Viewing Tops:a b
Priority vertex was not found
The end of the search priority.
Result:
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2

```

```
The end of the calculating the maximum flow.

Start of recalculation flow:
Current top:f
The path from f to c was found.
New bandwidth current top=2+6=8
New bandwidth previous top=7-6=1
Current top:c
The path from c to a was not found, creating an edge.
New bandwidth previous top=6-6=0
Recount final maximum flow:6+6=12
Viewing tops is cleared.
Viewing Tops:a
Priority vertex:b

i=0
i=1
Viewing tops:a b
Priority vertex was not found
The end of the search priority.

Result:
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2
```

2.

Beginning of the algorithm:

Viewing Tops:a

Priority vertex:b

i=0

i=1

Viewing tops:a b

Priority vertex:c

i=0

i=1

i=2

Viewing tops:a b c

Priority vertex:f

i=0

i=1

i=2

i=3

Viewing tops:a b c f

Priority vertex was not found

Priority vertex:g

i=0

i=1

i=2

i=3

i=4

Viewing tops:a b c f g

The current vertex is a estuary.

Calculating the maximum flow.

Current min:10000

Current flow:c->g 10

New min:10

Current min:10

Current flow:b->c 3

New min:3

Current min:3

Current flow:a->b 4

The end of the calculating the maximum flow.

```
Start of recalculation flow:
Current top:g
The path from g to c was not found, creating an edge.
New bandwidth previous top=10-3=7
Current top:c
The path from c to b was not found, creating an edge.
New bandwidth previous top=3-3=0
Current top:b
The path from b to a was not found, creating an edge.
New bandwidth previous top=4-3=1
Recount final maximum flow:0+3=3
Viewing tops is cleared.
Viewing Tops:a
Priority vertex:b

i=0
i=1
Viewing tops:a b
Priority vertex was not found
Priority vertex:c

i=0
i=1
i=2
Viewing tops:a b c
Priority vertex:f

i=0
i=1
i=2
i=3
Viewing tops:a b c f
Priority vertex was not found
Priority vertex:g

i=0
i=1
i=2
i=3
i=4
```



```

Viewing tops:a b c f g
The current vertex is a estuary.

Calculating the maximum flow.
Current min:10000
Current flow:c->g 7
New min:7
Current min:7
Current flow:a->c 5
New min:5
The end of the calculating the maximum flow.

Start of recalculation flow:
Current top:g
The path from g to c was found.
New bandwidth current top=3+5=8
New bandwidth previous top=7-5=2
Current top:c
The path from c to a was not found, creating an edge.
New bandwidth previous top=5-5=0
Recount final maximum flow:3+5=8
Viewing tops is cleared.
Viewing Tops:a
Priority vertex:b

i=0
i=1
Viewing tops:a b
Priority vertex was not found
Priority vertex:d

i=0
i=1
i=2
Viewing tops:a b d
Priority vertex:e

i=0
i=1
i=2
i=3

```

```

Viewing tops:a b d e
Priority vertex:g

i=0
i=1
i=2
i=3
i=4
Viewing tops:a b d e g
The current vertex is a estuary.

Calculating the maximum flow.
Current min:10000
Current flow:e->g 9
New min:9
Current min:9
Current flow:d->e 3
New min:3
Current min:3
Current flow:a->d 7
The end of the calculating the maximum flow.

Start of recalculation flow:
Current top:g
The path from g to e was not found, creating an edge.
New bandwidth previous top=9-3=6
Current top:e
The path from e to d was not found, creating an edge.
New bandwidth previous top=3-3=0
Current top:d
The path from d to a was not found, creating an edge.
New bandwidth previous top=7-3=4
Recount final maximum flow:8+3=11
Viewing tops is cleared.
Viewing Tops:a
Priority vertex:b

i=0
i=1
Viewing tops:a b
Priority vertex was not found

```

```
Priority vertex:b

i=0
i=1
Viewing tops:a b
Priority vertex was not found
Priority vertex:d

i=0
i=1
i=2
Viewing tops:a b d
Priority vertex was not found
The end of the search priority.

Result:
11
a b 3
a c 5
a d 3
b c 3
c f 0
c g 8
d e 3
e g 3
```

Вывод

В ходе выполнения лабораторной работы был изучен и реализован алгоритм Форда - Фалкерсона, который находит максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро.

ПРИЛОЖЕНИЯ А. ИСХОДНЫЙ КОД

Lab3.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct Neighbour { //структура для хранения соседей
    char symbol; //вершина
    int bandwidth; //введенная пропускная способность
    int cur_bandwidth; //промежуточная пропускная способность
    int final_bandwidth; //итоговая пропускная способность
    Neighbour(char symbol, int bandwidth) : symbol(symbol), bandwidth(bandwidth),
cur_bandwidth(bandwidth) { }
    Neighbour(char symbol) : symbol(symbol) { }
};

struct Top { //структура для хранения вершины
    char symbol; //вершина
    char prevTop; //пред вершина
    vector <Neighbour> neighbours; //вектор соседей
    Top() { }
    Top(char symbol) : symbol(symbol) { }
};

//вектор вершин и просмотренных вершин

bool cmp_top(const Top& first, const Top& second) //компаратор для вершин
{
    if (first.symbol < second.symbol)
        return true;
    else
        return false;
}

bool cmp_neigh(const Neighbour& first, const Neighbour& second) //компаратор для соседей
{
    if (first.symbol < second.symbol)
        return true;
    else
        return false;
}

bool isExist(vector<Top> vector, char tmp) //проверяет вхождение вершины в вектор
{
    for (size_t i = 0; i < vector.size(); i++)
        if (vector[i].symbol == tmp)
            return true;
    return false;
}
```

```

}

int findIndexTop(vector<Top> vector, char tmp) //находит индекс вершины в векторе
{
    for (size_t i = 0; i < vector.size(); i++)
        if (tmp == vector[i].symbol)
            return i;
    return -1;
}

int findIndexNeigh(Top top, char tmp) //находит индекс соседа в векторе
{
    for (size_t i = 0; i < top.neighbours.size(); i++) {
        if (top.neighbours[i].symbol == tmp)
            return i;
    }
    return -1;
}

void init_Graph(size_t N, vector <Top>& tops) {
    char curTop, endTop; //начальная и конечная вершины
    int bandwidth, index; //пропускная способность
    for (size_t i = 0; i < N; i++) {
        cin >> curTop >> endTop >> bandwidth;
        Top first; //нач вершина
        if (!isExist(tops, curTop)) { // если начальной вершины ещё нет в
векторе вершин
            first = Top(curTop); // создаём её
            Neighbour neighbour = Neighbour(endTop, bandwidth); // и её соседа
            first.neighbours.push_back(neighbour); // добавляем соседа в
вектор соседей
            tops.push_back(first); // добавляем вершину в вектор
вершин
        }
        else { // если начальная вершина уже есть в
векторе вершин
            Neighbour neighbour = Neighbour(endTop, bandwidth); // создаём её
соседа
            index = findIndexTop(tops, curTop);
            // находим индекс начальной вершины в векторе вершин
            tops[index].neighbours.push_back(neighbour); // добавляем соседа
в вектор соседей
        }
        if (!isExist(tops, endTop)) { // если конечной вершины нет в век-
торе вершин
            Top second = Top(endTop); // создаем ее и добавляем в
вектор вершин
            tops.push_back(second);
        }
    }
}

```

```

int maxFlowCount(const vector <Top>& tops, Top estr) // считает максимальный поток в
сквозном пути
{
    cout << endl;
    cout << "Calculating the maximum flow." << endl;
    int flow, min, indexPrev, indexCur; // промежуточный поток, максимальный поток, ин-
декс текущей и предыдущей вершины
    min = 10000; // недостижимый минимум
    Top cur = estr; // текущая вершина - сток
    while (cur.prevTop != '0') { // пока не дошли до истока
        cout << "Current min:" << min << endl;
        indexPrev = findIndexTop(tops, cur.prevTop); // находим индекс вершины
        предыдущей текущей
        indexCur = findIndexNeigh(tops[indexPrev], cur.symbol); // находим среди
        соседей предыдущей вершины текущую вершину
        flow = tops[indexPrev].neighbours[indexCur].cur_bandwidth;
        // вычисляем поток через это ребро
        cout << "Current flow:" << cur.prevTop << "->" << cur.symbol << ' ' << flow <<
endl;
        if (flow < min) {
            // если он меньше минимума
            min = flow;
            // то минимум равен текущему потоку
            cout << "New min:" << min << endl;
        }
        cur = tops[indexPrev]; // текущей вершиной становится
        предыдущая вершина
    }
    cout << "The end of the calculating the maximum flow." << endl;
    return min; // возвращаем минимальную пропускную способность сквозного пути
}

void recountFlow( vector <Top>& tops, Top estr, int maxFlow) // производит пересчет потока
по сквозному пути
{
    cout << endl;
    cout << "Start of recalculation flow:" << endl;
    int indexCur, indexPrev, indexNeighCur, indexNeighPrev; // индекс текущей и
    предыдущей вершины, соседа текущей и соседа предыдущей вершины
    Top cur = estr; // текущая вершина - сток
    while (cur.prevTop != '0') { // пока не дошли до истока
        cout << "Current top:" << cur.symbol << endl;
        indexCur = findIndexTop(tops, cur.symbol); // находим индекс
        текущей вершины
        indexPrev = findIndexTop(tops, cur.prevTop); // находим индекс
        предыдущей вершины
        indexNeighCur = findIndexNeigh(tops[indexCur], cur.prevTop); // находим
        среди соседей текущей вершины предыдущую вершину
        indexNeighPrev = findIndexNeigh(tops[indexPrev], cur.symbol); // находим
        среди соседей предыдущей вершины текущую вершину
        if (indexNeighCur == -1) { // если из текущей вершины
            нет ребра до предыдущей вершины
        }
    }
}

```

```

        cout << "The path from " << cur.symbol << " to " << cur.prevTop << " was
not found, creating an edge." << endl;
        Neighbour neighbour = Neighbour(cur.prevTop);           //создаем это ребро
        neighbour.cur_bandwidth = maxFlow;                       // пропускная
        способность этого ребра инициализируется максимальной
        neighbour.bandwidth = 0;                                  // его начальная пропуск-
        ная способность равна 0
        tops[indexCur].neighbours.push_back(neighbour);
    }
    else {
        cout << "The path from " << cur.symbol << " to " << cur.prevTop << " was
found." << endl;
        cout << "New bandwidth current top=" << tops[indexCur].neighbours[in-
dexNeighCur].cur_bandwidth << '+' << maxFlow << '=' << tops[indexCur].neighbours[in-
dexNeighCur].cur_bandwidth + maxFlow << endl;
        tops[indexCur].neighbours[indexNeighCur].cur_bandwidth += maxFlow; //
        иначе добавляем к пропускной способности ребра из текущей вершины величину
        максимальной
    }
    cout << "New bandwidth previous top=" << tops[indexPrev].neighbours[in-
dexNeighPrev].cur_bandwidth << '-' << maxFlow << '=' << tops[indexPrev].neighbours[in-
dexNeighPrev].cur_bandwidth - maxFlow << endl;
    tops[indexPrev].neighbours[indexNeighPrev].cur_bandwidth -= maxFlow;
    //пропускная способность предыдущей вершины до текущей уменьшается на максимальную
    cur = tops[indexPrev];                                       //текущая вершина теперь - предыду-
    щая
}
}

void Priority(int& index, Top cur,const vector <Top>& viewingTop) { //рассчитываем приоритет
    вершин
    int priority, min; //мин приоритет, промежуточный приоритет
    bool flag; //флаг для определения расположения символа относительно начала алфа-
    вита
    for (size_t i = 0; i < cur.neighbours.size(); i++) { // перебор соседей текущей вершины
        if (cur.neighbours[i].cur_bandwidth > 0 && !isExist(viewingTop, cur.neigh-
        bours[i].symbol)) { // если пропускная способность пути до соседа больше нуля и сосед не
        находится в векторе посещенных вершин
            priority = abs(cur.neighbours[i].symbol - cur.symbol); //расчитываем
            приоритет
            if (index == 10000 || priority < min) { //если это
            первая итерация цикла или текущий приоритет меньше минимума
                min = priority; // мини-
                мум равен приоритету
                index = i; // индекс вер-
                шины с мин разницей
                if (cur.neighbours[i].symbol < cur.symbol)
                // если сосед рассматриваемой вершины находится ближе к началу алфавита
                flag = true; // ста-
                вим flag=true
            else // иначе
                помечаем flag=false
        }
    }
}

```



```

        flag = false;
    }
    else if (priority == min && cur.neighbours[i].symbol < cur.symbol && flag
== false) { // если приоритет равен минимуму, то нужно проверить расположение символа от-
носительно начала алфавита
        index = i;
        flag = true; //если сосед находится ближе к началу алфа-
вита
    }
}
}
}
}

```

```

int FordFalk(char source, char estr, vector <Top>& tops, vector <Top>& viewingTop) { //поиск
максимального потока и пересчет пропускных способностей вершин
    int i = findIndexTop(tops, source); //находим индекс истока
    tops[i].prevTop = '0'; //устанавливаем пред символ у истока
    Top cur = tops[i]; // текущая вершина - исток
    int maxFlow, finalMax = 0; //промежуточный макс поток и итоговый
    viewingTop.push_back(cur); //помещаем исток в вектор просмотренных вершин
    cout << "Viewing Tops:";
    for (size_t i = 0; i < viewingTop.size(); i++)
        cout << viewingTop[i].symbol << ' ';
    cout << endl;
    while (true) { //пока не найдем макс поток
        int index_min = 10000; //недостижимый минимум
        Priority(index_min, cur, viewingTop); //выбираем вершины по приоритету
        if (index_min != 10000) { // если нашли соседа
            cout << "Priority vertex:" << cur.neighbours[index_min].symbol << endl;
            i = findIndexTop(tops, cur.neighbours[index_min].symbol);
            // находим его индекс в векторе вершин
            tops[i].prevTop = cur.symbol; // предыдущая вершина
соседа это текущая вершина
            cur = tops[i]; // сосед становится текущей
вершиной
            viewingTop.push_back(cur); //помещается в вектор просмотренных вер-
шин
            cout << endl;
            for (size_t i = 0; i < viewingTop.size(); i++)
                cout << "i=" << i << endl;
            cout << "Viewing tops:";
            for (size_t j = 0; j < viewingTop.size(); j++)
                cout << viewingTop[j].symbol << ' ';
            cout << endl;
            if (cur.symbol == estr) { // если дошли до стока
                cout << "The current vertex is a estuary." << endl;
                maxFlow = maxFlowCount(tops, cur); // рассчитываем мини-
мальную пропускную способность
                recountFlow(tops, tops[i], maxFlow); // производим пересчет
пропускных способностей пути
                cout << "Recount final maximum flow:" << finalMax << '+' <<
maxFlow << '=' << finalMax + maxFlow << endl;
            }
        }
    }
}

```

```

        finalMax += maxFlow;                // пересчитываем итоговый
максимальный поток
        cout << "Viewing tops is cleared." << endl;
        viewingTop.clear();
        //очищаем вектор просмотренных вершин
        i = findIndexTop(tops, source);      // текущей вершиной стано-
вится исток
        cur = tops[i];
        viewingTop.push_back(cur);
        cout << "Viewing Tops:";
        for (size_t i = 0; i < viewingTop.size(); i++)
            cout << viewingTop[i].symbol << ' ';
        cout << endl;
    }

    }
    else { // если не нашли соседа
        if (cur.prevTop == '0') { // если текущая вершина исток, то выходим из
цикла
            cout << "The end of the search priority." << endl;
            break;
        }
        else {
            cout << "Priority vertex was not found" << endl;
            i = findIndexTop(tops, cur.prevTop); // иначе откатываемся к
предыдущей вершине
            cur = tops[i];
        }
    }
}
for (size_t i = 0; i < tops.size(); i++) { // рассчитываем итоговые пропускные способ-
ности через все ребра
    for (size_t j = 0; j < tops[i].neighbours.size(); j++) {
        tops[i].neighbours[j].final_bandwidth = tops[i].neighbours[j].bandwidth -
tops[i].neighbours[j].cur_bandwidth;
    }
}
return finalMax; //возвращаем максимальный поток
}

int main()
{
    vector <Top> tops, viewingTop;
    size_t N;
    char source, estr;
    cout << "Enter information about the graph:" << endl;
    cin >> N; //количество вершин
    cin >> source; //исток
    cin >> estr; //сток
    int max = 0; //макс поток
    init_Graph(N, tops); // заполняем граф
    cout << "Beginning of the algorithm:" << endl;

```

```

max = FordFalk(source, estr,tops,viewingTop);//находим макс поток
sort(tops.begin(), tops.end(), cmp_top);    //сортируем вершины графа
for (size_t i = 0; i < tops.size(); ++i) {
    sort(tops[i].neighbours.begin(), tops[i].neighbours.end(), cmp_neigh); //сортируем
соседей каждой вершины
}
cout << endl;
cout << "Result:" << endl; //выводим результат работы
cout << max << endl;
for (size_t i = 0; i < tops.size(); i++) {
    for (size_t j = 0; j < tops[i].neighbours.size(); j++) {
        if (tops[i].neighbours[j].bandwidth != 0) { // Если ребро было введено из-
начально,а не создано во время работы алгоритма
            cout << tops[i].symbol << " " << tops[i].neighbours[j].symbol << " ";
            if (tops[i].neighbours[j].final_bandwidth >= 0)
                cout << tops[i].neighbours[j].final_bandwidth<<endl;
            else
                cout << 0 << endl;
        }
    }
}
return 0;
}

```