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: Алгоритм Форда-Фалкерсона

Студент гр. 8303 _____ Деркач Н.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить алгоритм Форда-Фалкерсона для поиска максимального потока в сети.

Формулировка задачи.

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
vn - сток
vi vj ωij- ребро графа
vi vj ωij- ребро графа
…

Выходные данные:

Pmax - величина максимального потока
vi vj ωij - ребро графа с фактической величиной протекающего потока
vi vj ωij - 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 — 3. Поиск в глубину. Рекурсивная реализация.

Суть и сложность алгоритмов.

На первом шаге алгоритма строится остаточная сеть , в которой изначально поток через каждое ребро равен 0, максимальный поток в сети устанавливается в 0.

На втором шаге идёт рекурсивный поиск в глубину пути от истока к стоку через рёбра, которые имеют ненулевой вес. В случае если пойти дальше некуда, переход на шаг 4.

На третьем шаге в найденном пути ищется ребро с минимальным весом, его величина добавляется к максимальному потоку в графе, вычитается из весов всех ребёр и прибавляется к значению обратных рёбер, после чего переход на шаг 2.

На четвертом шаге выводится максимальный поток и величина потока через каждое ребро.

Сложность алгоритма по времени — O((E + V)*F), где E – количество рёбер, V — количество вершин и F — максимальный поток в сети.

Сложность алгоритма по памяти – O(E + V), где E – кол-во рёбер, V — кол-во вершин.

Описание функций и структур.

struct **Edge{char start;char end;int weight;int straight;int back;}** - структура для хранения ребра графа. Start — начальная вершина, end – конечная, weight – вес ребра, straight — остаточный поток вперёд, back — остаточный поток назад.

bool **compare**(Edge first, Edge second) — функция сравнения двух рёбер, которые передаются в качестве аргументов, по символам их начальных и конечных вершин. Функция нужна для сортировки с помощью библиотечной функции sort, имеющей сигнатуру void sort(RandomIt first, RandomIt last, Compare comp);

В конструкторе класса Graph заполняется вектор структур рёбер.

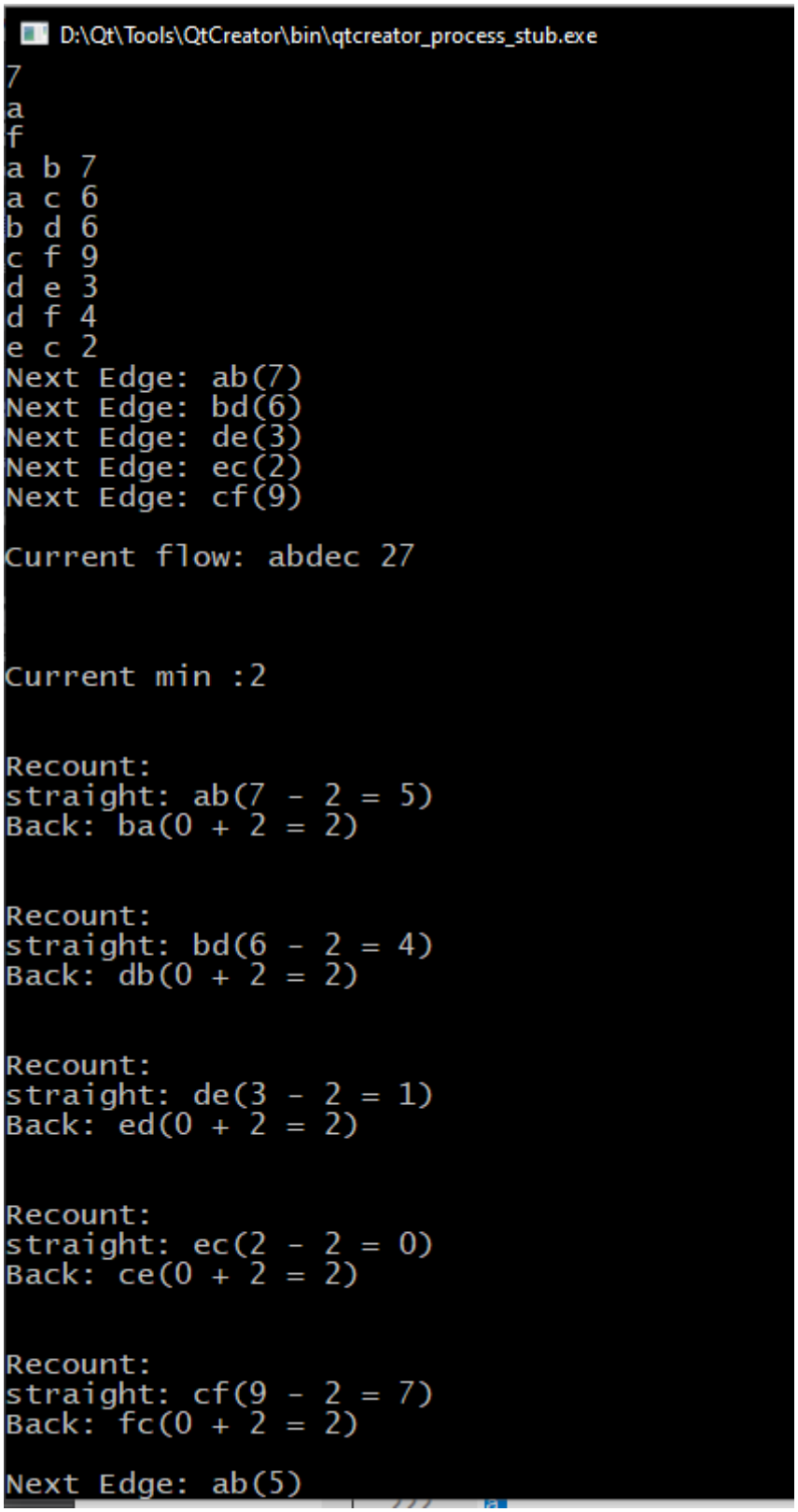
bool **isVisited**(char peak) — функция проверяет, была ли уже просмотрена вершина, которая передаётся в качестве аргумента. Просмотренные вершины хранятся в vector <char> visited , который является полем классов обоих алгоритмов.

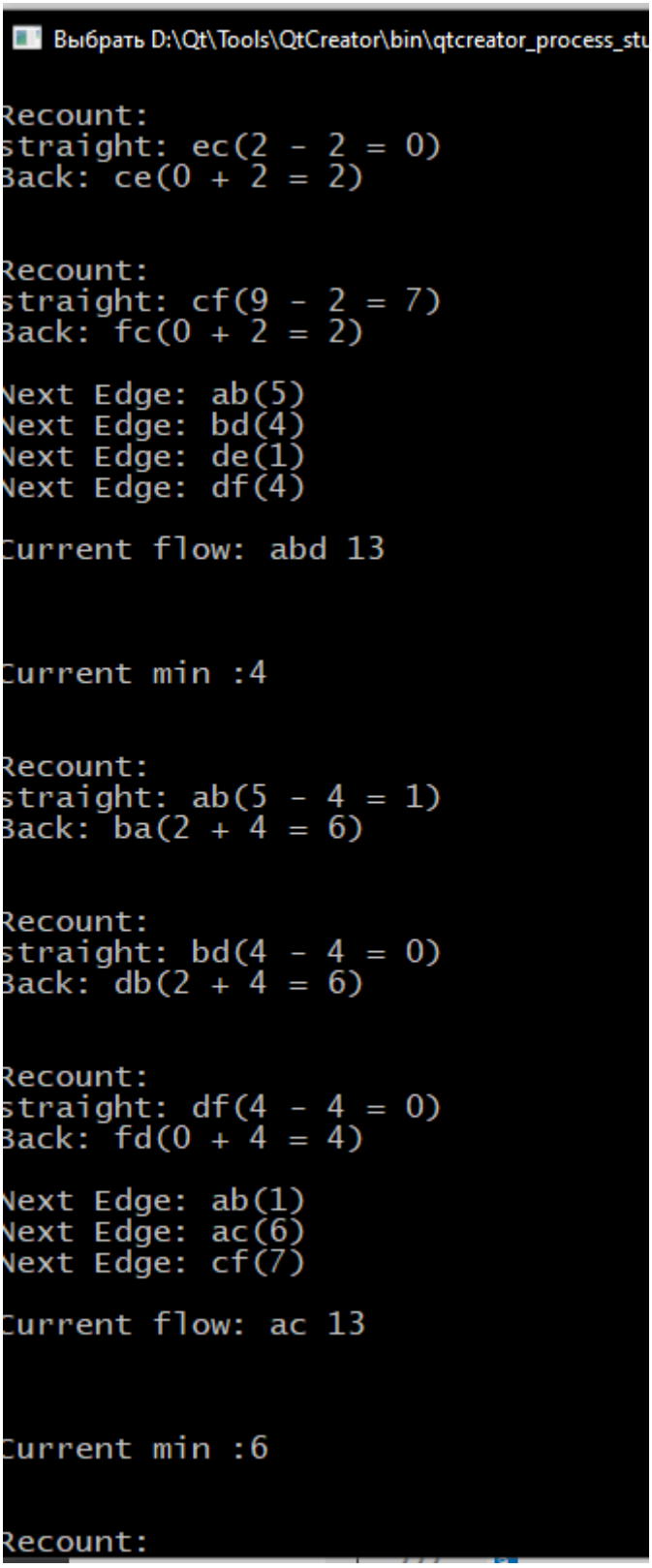
bool **Search**(char peak, int& min) – выполняет рекурсивный поиск пути от истока к стоку. Peak – очередная просматриваемая вершина, min — величина минимальной пропускной способности из всех рёбер в найденном потоке.

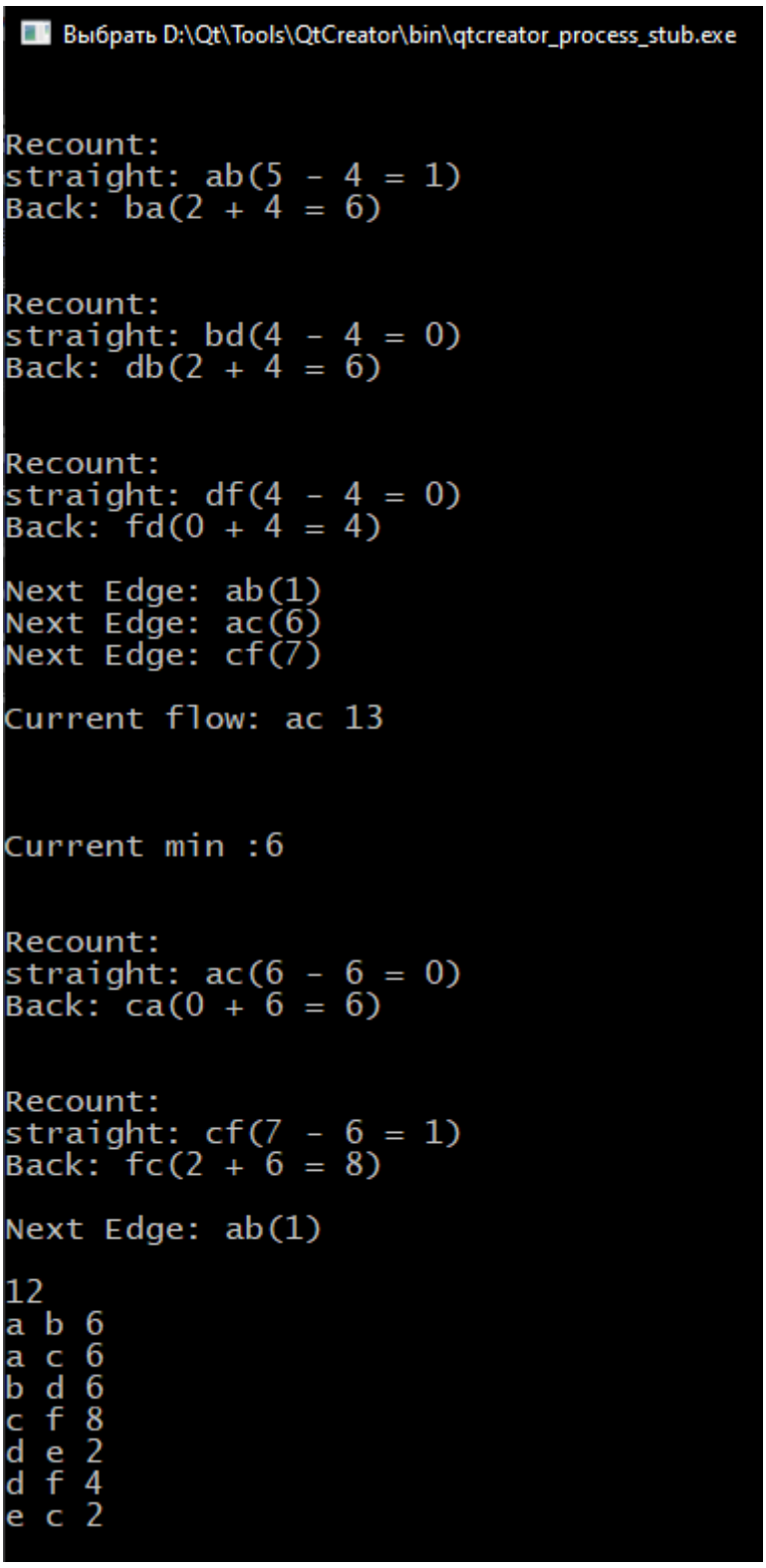
void **FFSearch**() - Запускает рекурсивный поиск, который меняет значение min, на основе которого пересчитываются пропускные способности рёбер. После нахождения максимального потока выводит результат.

Тестирование.

Тест 1:


D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe
7
a
f
a b 7
a c 6
b d 6
c f 9
d e 3
d f 4
e c 2
Next Edge: ab(7)
Next Edge: bd(6)
Next Edge: de(3)
Next Edge: ec(2)
Next Edge: cf(9)
Current flow: abdec 27
Current min :2
Recount:
straight: ab(7 - 2 = 5)
Back: ba(0 + 2 = 2)
Recount:
straight: bd(6 - 2 = 4)
Back: db(0 + 2 = 2)
Recount:
straight: de(3 - 2 = 1)
Back: ed(0 + 2 = 2)
Recount:
straight: ec(2 - 2 = 0)
Back: ce(0 + 2 = 2)
Recount:
straight: cf(9 - 2 = 7)
Back: fc(0 + 2 = 2)
Next Edge: ab(5)


Выбрать D:\Qt\Tools\QtCreator\bin\qtcreator_process_stu
Recount:
straight: ec(2 - 2 = 0)
Back: ce(0 + 2 = 2)
Recount:
straight: cf(9 - 2 = 7)
Back: fc(0 + 2 = 2)
Next Edge: ab(5)
Next Edge: bd(4)
Next Edge: de(1)
Next Edge: df(4)
Current flow: abd 13
Current min :4
Recount:
straight: ab(5 - 4 = 1)
Back: ba(2 + 4 = 6)
Recount:
straight: bd(4 - 4 = 0)
Back: db(2 + 4 = 6)
Recount:
straight: df(4 - 4 = 0)
Back: fd(0 + 4 = 4)
Next Edge: ab(1)
Next Edge: ac(6)
Next Edge: cf(7)
Current flow: ac 13
Current min :6
Recount:


Выбрать D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe
Recount:
straight: ab(5 - 4 = 1)
Back: ba(2 + 4 = 6)
Recount:
straight: bd(4 - 4 = 0)
Back: db(2 + 4 = 6)
Recount:
straight: df(4 - 4 = 0)
Back: fd(0 + 4 = 4)
Next Edge: ab(1)
Next Edge: ac(6)
Next Edge: cf(7)
Current flow: ac 13
Current min :6
Recount:
straight: ac(6 - 6 = 0)
Back: ca(0 + 6 = 6)
Recount:
straight: cf(7 - 6 = 1)
Back: fc(2 + 6 = 8)
Next Edge: ab(1)
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2

Тест 2:

```
D:\Qt\Tools\QtCreator\bin\qtcreator_proc
19
a
j
a b 2
a c 9
a d 3
a e 5
b e 4
b h 7
c d 8
c f 5
c g 4
d g 6
e g 8
e h 1
e j 6
f g 3
f i 2
g j 4
g i 9
h j 5
i j 8
Next Edge: ab(2)
Next Edge: be(4)
Next Edge: eg(8)
Next Edge: gj(4)

Current flow: abeg 18



Current min :2


Recount:
straight: ab(2 - 2 = 0)
Back: ba(0 + 2 = 2)


Recount:
straight: be(4 - 2 = 2)
Back: eb(0 + 2 = 2)


Recount:
straight: eg(8 - 2 = 6)
Back: ge(0 + 2 = 2)


Recount:
straight: gj(4 - 2 = 2)
Back: jg(0 + 2 = 2)

Next Edge: ac(9)
Next Edge: cd(8)
Next Edge: dg(6)
Next Edge: ge(2)
Next Edge: eb(2)
Next Edge: bh(7)
Next Edge: hj(5)

Current flow: acdgebh 39
```

```
Current min :2


Recount:
straight: ac(9 - 2 = 7)
Back: ca(0 + 2 = 2)


Recount:
straight: cd(8 - 2 = 6)
Back: dc(0 + 2 = 2)


Recount:
straight: dg(6 - 2 = 4)
Back: gd(0 + 2 = 2)


Recount:

Recount:

Recount:
straight: bh(7 - 2 = 5)
Back: hb(0 + 2 = 2)


Recount:
straight: hj(5 - 2 = 3)
Back: jh(0 + 2 = 2)

Next Edge: ac(7)
Next Edge: cd(6)
Next Edge: dg(4)
Next Edge: gj(2)

Current flow: acdg 19



Current min :2


Recount:
straight: ac(7 - 2 = 5)
Back: ca(2 + 2 = 4)


Recount:
straight: cd(6 - 2 = 4)
Back: dc(2 + 2 = 4)


Recount:
straight: dg(4 - 2 = 2)
Back: gd(2 + 2 = 4)


Recount:
```

```
Recount:
straight: gj(2 - 2 = 0)
Back: jg(2 + 2 = 4)

Next Edge: ac(5)
Next Edge: cd(4)
Next Edge: dg(2)
Next Edge: gi(9)
Next Edge: ij(8)

Current flow: acdgi 28



Current min :2


Recount:
straight: ac(5 - 2 = 3)
Back: ca(4 + 2 = 6)


Recount:
straight: cd(4 - 2 = 2)
Back: dc(4 + 2 = 6)


Recount:
straight: dg(2 - 2 = 0)
Back: gd(4 + 2 = 6)


Recount:
straight: gi(9 - 2 = 7)
Back: ig(0 + 2 = 2)


Recount:
straight: ij(8 - 2 = 6)
Back: ji(0 + 2 = 2)

Next Edge: ac(3)
Next Edge: cd(2)
Next Edge: cf(5)
Next Edge: fg(3)
Next Edge: gi(7)
Next Edge: ij(6)

Current flow: acfgi 24



Current min :3


Recount:
straight: ac(3 - 3 = 0)
Back: ca(6 + 3 = 9)
```

```
straight: ac(3 - 3 = 0)
Back: ca(6 + 3 = 9)


Recount:
straight: cf(5 - 3 = 2)
Back: fc(0 + 3 = 3)


Recount:
straight: fg(3 - 3 = 0)
Back: gf(0 + 3 = 3)


Recount:
straight: gi(7 - 3 = 4)
Back: ig(2 + 3 = 5)


Recount:
straight: ij(6 - 3 = 3)
Back: ji(2 + 3 = 5)

Next Edge: ad(3)
Next Edge: dc(6)
Next Edge: cf(2)
Next Edge: fi(2)
Next Edge: ig(5)
Next Edge: ij(3)

Current flow: adcfi 16




Current min :2


Recount:
straight: ad(3 - 2 = 1)
Back: da(0 + 2 = 2)


Recount:

Recount:
straight: cf(2 - 2 = 0)
Back: fc(3 + 2 = 5)


Recount:
straight: fi(2 - 2 = 0)
Back: if(0 + 2 = 2)


Recount:
straight: ij(3 - 2 = 1)
Back: ji(5 + 2 = 7)

Next Edge: ad(1)
Next Edge: dc(4)
Next Edge: cg(4)
```

```
Next Edge: cg(4)
Next Edge: gf(3)
Next Edge: gi(4)
Next Edge: ij(1)

Current flow: adcgi 14




Current min :1


Recount:
straight: ad(1 - 1 = 0)
Back: da(2 + 1 = 3)


Recount:

Recount:
straight: cg(4 - 1 = 3)
Back: gc(0 + 1 = 1)


Recount:
straight: gi(4 - 1 = 3)
Back: ig(5 + 1 = 6)


Recount:
straight: ij(1 - 1 = 0)
Back: ji(7 + 1 = 8)

Next Edge: ae(5)
Next Edge: eg(8)
Next Edge: gc(1)
Next Edge: cd(5)
Next Edge: gf(3)
Next Edge: gi(3)
Next Edge: eh(1)
Next Edge: hb(2)
Next Edge: hj(3)

Current flow: aeh 9



Current min :1


Recount:
straight: ae(5 - 1 = 4)
Back: ea(0 + 1 = 1)


Recount:
straight: eh(1 - 1 = 0)
Back: he(0 + 1 = 1)
```

```
Back: ea(0 + 1 = 1)


Recount:
straight: eh(1 - 1 = 0)
Back: he(0 + 1 = 1)


Recount:
straight: hj(3 - 1 = 2)
Back: jh(2 + 1 = 3)

Next Edge: ae(4)
Next Edge: eg(8)
Next Edge: gc(1)
Next Edge: cd(5)
Next Edge: gf(3)
Next Edge: gi(3)
Next Edge: ej(6)

Current flow: ae 10



Current min :4


Recount:
straight: ae(4 - 4 = 0)
Back: ea(1 + 4 = 5)


Recount:
straight: ej(6 - 4 = 2)
Back: je(0 + 4 = 4)


19
a b 2
a c 9
a d 3
a e 5
b e 0
b h 2
c d 3
c f 5
c g 1
d g 6
e g 0
e h 1
e j 4
f g 3
f i 2
g i 6
g j 4
h j 3
i j 8
```

Выводы.

В лабораторной работе был изучен алгоритм Форда-Фалкерсона путём написания программы на языке C++.

Приложение А. Исходный код.

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

using namespace std;

struct Edge
{
    bool fl;
    char start;
    char end;
    int weight;        // вес
    int straight;         // ост. поток вперёд
    int back;           // ост. поток назад
};

bool compare(Edge first, Edge second)
{
    if(first.start == second.start)
        return first.end < second.end;
    return first.start < second.start;
}

class Graph
{
private:
    vector <Edge> graph;
    char source;
    char finish;
    int N;
    vector <char> visited;
    vector <char> result;

public:
    Graph()
    {
        cin >> N;
        cin >> source >> finish;
        for(int i = 0; i < N; i++)
        {
            Edge element;
            cin >> element.start >> element.end >> element.weight;
            element.straight = element.weight;
            element.back = 0;
            element.fl = false;
            bool flag = true;
            for(int i = 0; i < graph.size(); i++)                    // заполняет граф
рёбрами
            {
```

```
                if(graph[i].start == element.end && graph[i].end == element.start)
                {
                    graph[i].back += element.straight;
                    flag = false;
                    graph[i].fl = true;
                    break;
                }
            }
            if(!flag)
                continue;
            graph.push_back(element);
        }
    }

    bool isVisited(char peak)
    {
        for(size_t i = 0; i < visited.size(); i++)
            if(visited[i] == peak)
                return true;
        return false;
    }

    bool Search(char peak, int& min)        // поиск минимальной пропускной
способности в потоке
    {
        if(peak == finish)
        {
            result.push_back(peak);
            cout << "\nCurrent flow: ";
            int flow = 0;

            for(int i = 1; i < result.size(); i++)
            {
                for(int j = 0; j < graph.size(); j++)
                {
                    if(graph[j].start == result[i-1] && graph[j].end == result[i])
// пересчёт пропускных способностей
                    {
                        flow += graph[j].straight;
                    }
                    if(graph[j].end == result[i-1] && graph[j].start == result[i])
                    {
                        flow += graph[j].back;
                    }
                }
            }

            for (int k = 0; k < result.size()-1; k++){
                cout << result[k];
            }
            cout << " " << flow;
            cout << "\n\n";
            return true;
        }
        visited.push_back(peak);           // peak - очередная вершина
        for(size_t i(0); i < graph.size(); i++) // проход по графу
        {
            if(peak == graph[i].start)     // если равна вершине-началу очередного
ребра
            {
                if(isVisited(graph[i].end) || graph[i].straight == 0){  // если ещё
не были в вершине-конце этого ребра или остат. проп. способ. вперёд равна 0
                    if (i != 0){
                     //  cout << "Returning\n";
                    }
                    continue;
                }
```

```
                cout << "Next Edge: " << graph[i].start << graph[i].end << "(" <<
graph[i].straight << ")" << endl;
                result.push_back(graph[i].start);      // вершина-начало добавляется
к ответу
                bool flag = Search(graph[i].end, min);       // идёт рекурсия в
глубину графа(подаётся вершина-конец этого графа)
                if(flag)        // вернет тру только когда дойдёт до конечной
вершины, и только тогда, возвращаясь из рекурсии, зайдёт в этот иф
                {
                   if(graph[i].straight < min)
                       min = graph[i].straight;
                   return true;
                }
                result.pop_back();
            }
            if(peak == graph[i].end)     // если равна вершине-концу очередного ребра
            {
                if(isVisited(graph[i].start) || graph[i].back == 0) { // если ещё не
были в вершине-конце этого ребра или остат. проп. способ. назад равна 0
                    if (i != 0){
                    //   cout << "Returning\n";
                    }
                    continue;
                }
                cout << "Next Edge: " << graph[i].end << graph[i].start << "(" <<
graph[i].back << ")"  << endl;
                result.push_back(graph[i].end);  // вершина-конец добавляется к
ответу
                bool flag = Search(graph[i].start, min);
                if(flag)
                {
                    cout << "\n\n";
                   if(graph[i].back < min)
                        min = graph[i].back;
                   return true;
                }
                result.pop_back();
            }
        }
        return false;
    }

    void FFSearch()
    {
        int res = 0;
        int min = 9999;

        while(Search(source, min))
        {
            cout << "\n\nCurrent min :" << min << "\n\n";
            for(int i = 1; i < result.size(); i++)
            {
                cout << "\nRecount:\n";
                for(int j = 0; j < graph.size(); j++)
                {
                    if(graph[j].start == result[i-1] && graph[j].end == result[i])
// пересчёт пропускных способностей
                    {
                        cout << "straight: ";
                        cout << graph[j].start << graph[j].end << "(" <<
graph[j].straight << " - " << min;
                        graph[j].straight -= min;
                        cout << " = " << graph[j].straight << ")\n";
                        cout << "Back: ";
                        cout << graph[j].end << graph[j].start << "(" <<
graph[j].back << " + " << min;
                        graph[j].back += min;
```

```
                        cout << " = " << graph[j].back << ")\n\n";
                    }
                    if(graph[j].end == result[i-1] && graph[j].start == result[i])
                    {
                        graph[j].straight += min;
                        graph[j].back -= min;
                    }
                }
            }
            res += min;
            visited.clear();
            result.clear();
            min = 9999;
        }

        sort(graph.begin(), graph.end(), compare);
        cout << "\n" << res << endl;
        for(int i = 0; i < graph.size(); i++)
        {
            int peak = max(graph[i].weight - graph[i].straight, 0 - graph[i].back);
            if(graph[i].fl == true)
            {
                if(peak < 0)
                    peak = 0;
                cout << graph[i].start << " " << graph[i].end << " " << peak << endl;
                swap(graph[i].start, graph[i].end);
                swap(graph[i].back, graph[i].straight);
                graph[i].fl = false;
                sort(graph.begin(), graph.end(), compare);
                i--;
            }
            else
            {
                cout << graph[i].start << " " << graph[i].end << " " << peak << endl;
            }
        }
    }
};

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    Graph element;
    element.FFSearch();

    return a.exec();
}
```