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 _____ Стукалев А.И.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить и реализовать на языке программирования C++ алгоритм Форда-Фалкерсона поиска максимального потока в сети, а также фактическую величину потока, протекающего через каждое ребро.

Формулировка задания.

Вариант 3. Поиск в глубину. Рекурсивная реализация.

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).

Теоретические сведения.

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 истока = 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 истока, 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.

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

Соответствующие выходные данные для алгоритма Форда-Фалкерсона.

12

a b 6

a c 6

b d 6

c f 8

d e 2

d f 4

e c 2

Описание алгоритма.

Алгоритм Форда-Фалкерсона:

Строится изначальная остаточная сеть в которой поток через каждое ребро равен нулю, следовательно, и максимальный поток равен нулю. Затем ищется путь от истока к стоку по рёбрам, вес которых не нулевой, если пути найдено не было, то выводится текущий максимальный поток и поток через каждое ребро. Если же путь был найден, то в пути ищется ребро с минимальным весом, вес ребра вычитается из весов рёбер потока и прибавляется к величине максимального потока. Далее снова ищется путь от истока к стоку, до тех пор, пока найти путь возможно.

Сложность алгоритма по операциям: O((|E| + |V|)*F)

(|E| + |V|) – поиск в глубину, где E – рёбра, V - вершины, F – максимальный поток в графе.

Сложность алгоритма по памяти: O ((|E| + |V|)), где E – рёбра, V - вершины

Описание структур данных.

1.

```
struct edge
{
	char first;
	char second;
	int weight;//вес
	int forward;//вперёд
	int back;//назад
	bool not_oriented;
};
```

Структура отражающая сущность ребра графа, first – начальная вершина ребра, second – конечная, weight – вес ребра, forward – поток через ребро, back – обратный поток, not_oriented – логическая переменная отвечающая за то, ориентировано или нет ребро, то есть можно ли пройти по ребру в обоих

направлениях.

2.

```
struct ways
{
	char start;
	char finish;
	int res;
}
```

Структура, использующаяся для хранения заданных пользователем истока и стока, в start хранится исток, в finish – сток, а также для хранения величины максимального потока - res.

```
class Graph

{
	std::vector <edge> edges;
	std::vector <char> viewed_points;
	std::vector <char> curr_path;
}
```

Класс, необходимый для работы алгоритма Форда-Фалкерсона, edges-контейнер типа edge, используется для хранения всех рёбер, viewedpoints – контейнер типа char, используется для хранения просмотренных вершин, curr_path – контейнер типа char, используется для хранения текущего пути

Описание функций.

1. Graph()

Конструктор класса Graph. Происходит считывание из консоли введённых пользователем количества рёбер графа, а также истока и стока. Затем считываются рёбра графа и заносятся в контейнер edges. При считывании происходит проверка, является ли ребро двусторонним, если является, то переменная non_oriented в структуре edge становится true, в этом случае заполняется поле back, в остальных случаях поле back инициализируется нулём.

2. ~Graph()

Деструктор класса Graph, который очищает контейнеры с рёбрами, просмотренными вершинами и текущим путём.

3. bool isViewing(char value)

Функция класса Graph, которая принимает вершину графа(char value), и проверяет просмотрена ли она на текущий момент, если да то возвращает true, иначе false.

4. bool Search(char value, int& min)

Рекурсивная функция поиска в глубину класса Graph. Принимает в качестве аргументов вершину, из которой необходимо вести поиск (chat value), и текущий наименьший вес ребра (min). Возвращает true, если будет найдем путь до стока, иначе false.

5. void FordFalk()

Функция класса Graph, которая ищет максимальный поток в сети. Наименьший вес ребра изначально инициализируется большим числом. Затем в цикле while вызывается функция Search, пока она не вернёт false. После каждого найденного пути минимальный вес ребра вычитается из весов рёбер потока и прибавляется к величине максимального потока.

6. void print_result()

Функция класса Graph, которая выводит результат работы программы на экран, сортируя рёбра в лексикографическом порядке.

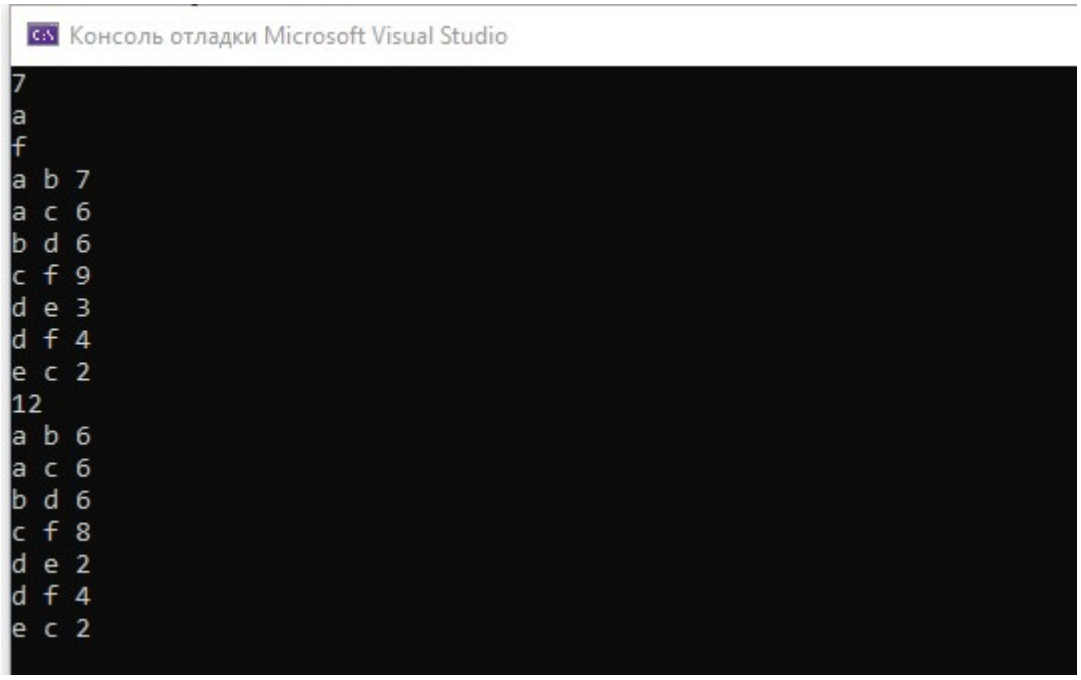
7. bool compare(edge first, edge second)

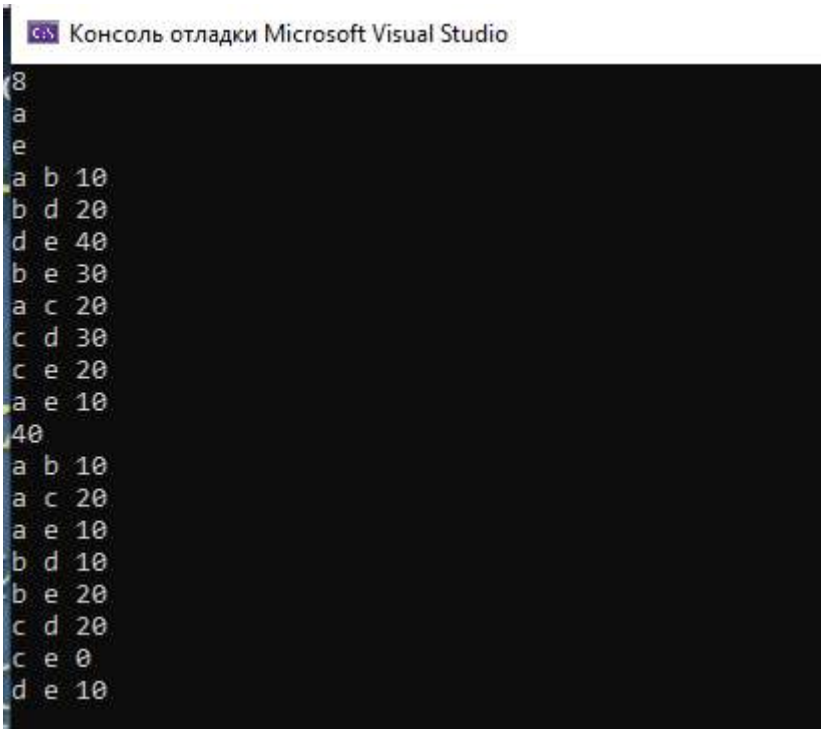
Функция-компаратор, которая сравнивает рёбра по первой вершине, возвращает true, если первая меньше второй, иначе false, если же они равные, сравнивает по второй вершине, возвращает true, если первая меньше второй, иначе false.

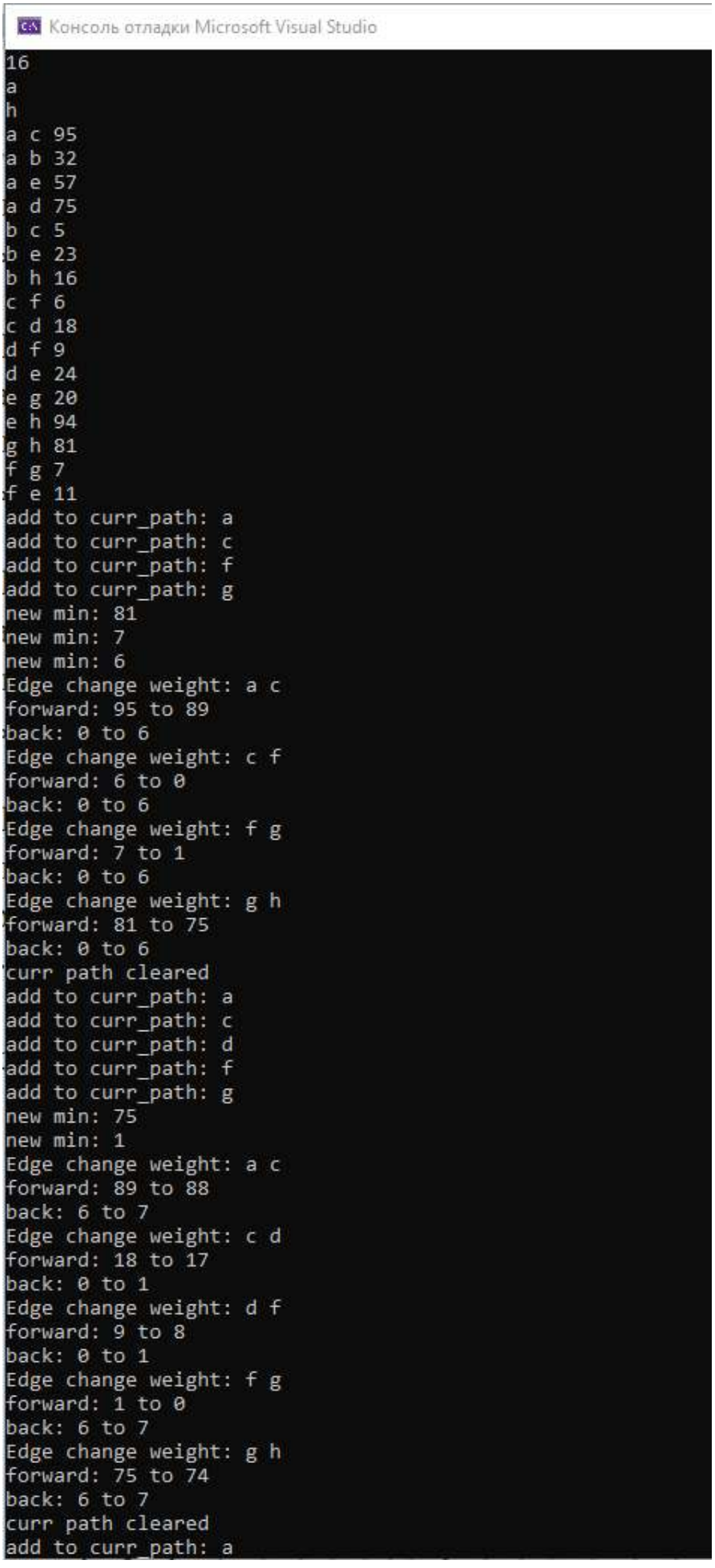
Способ хранения частичных решений.

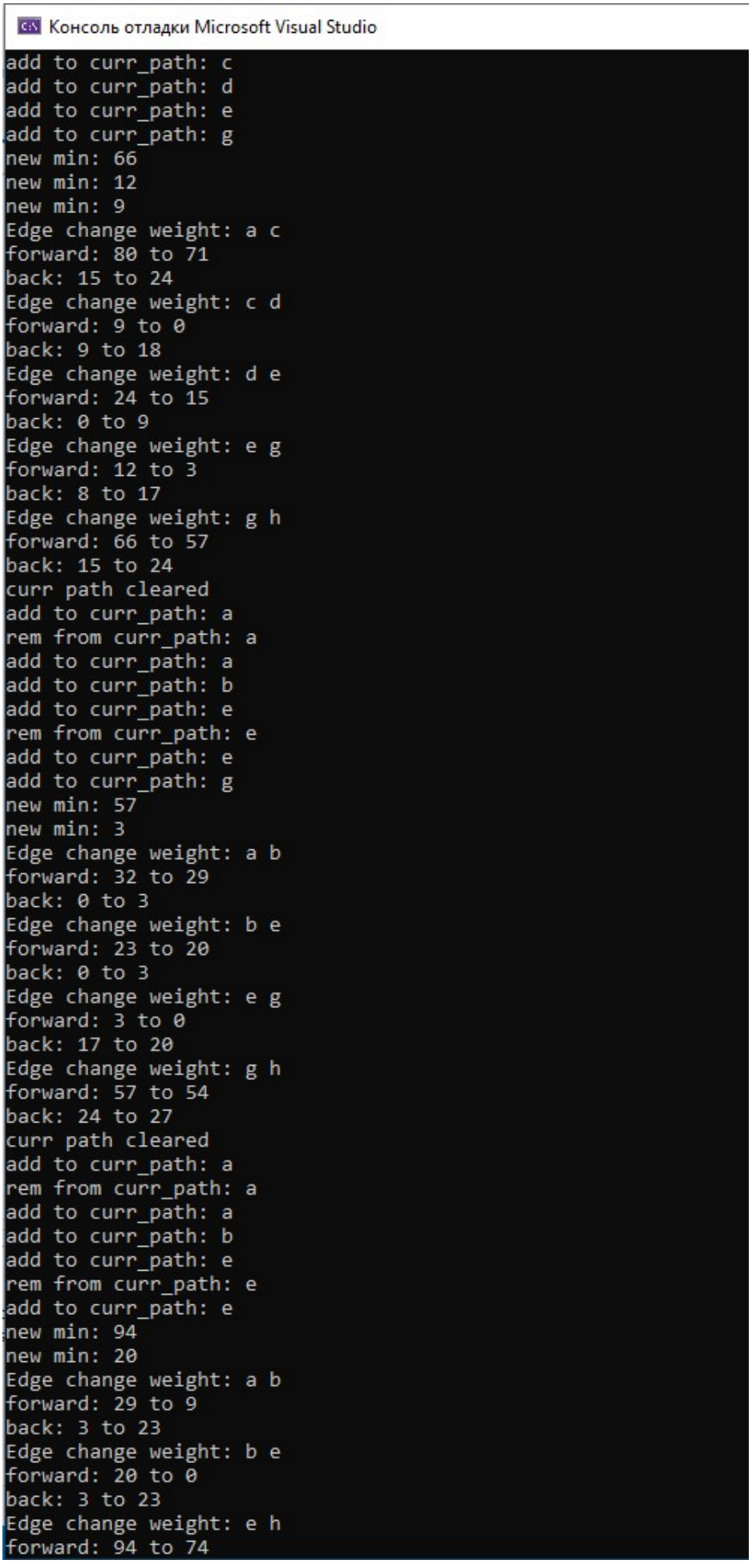
Частичные решения, т.е. текущий путь, хранятся в контейнере curr_path.

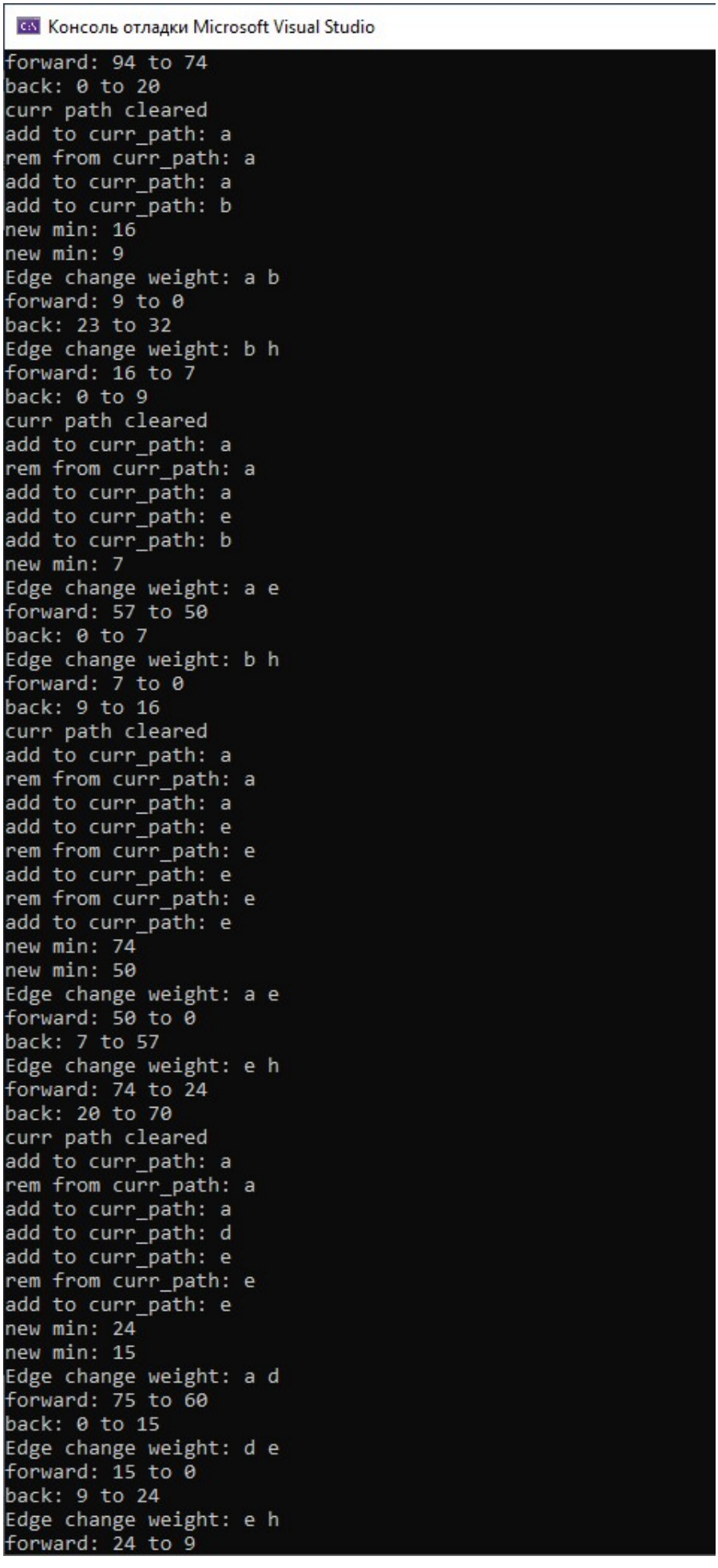
Тестирование.

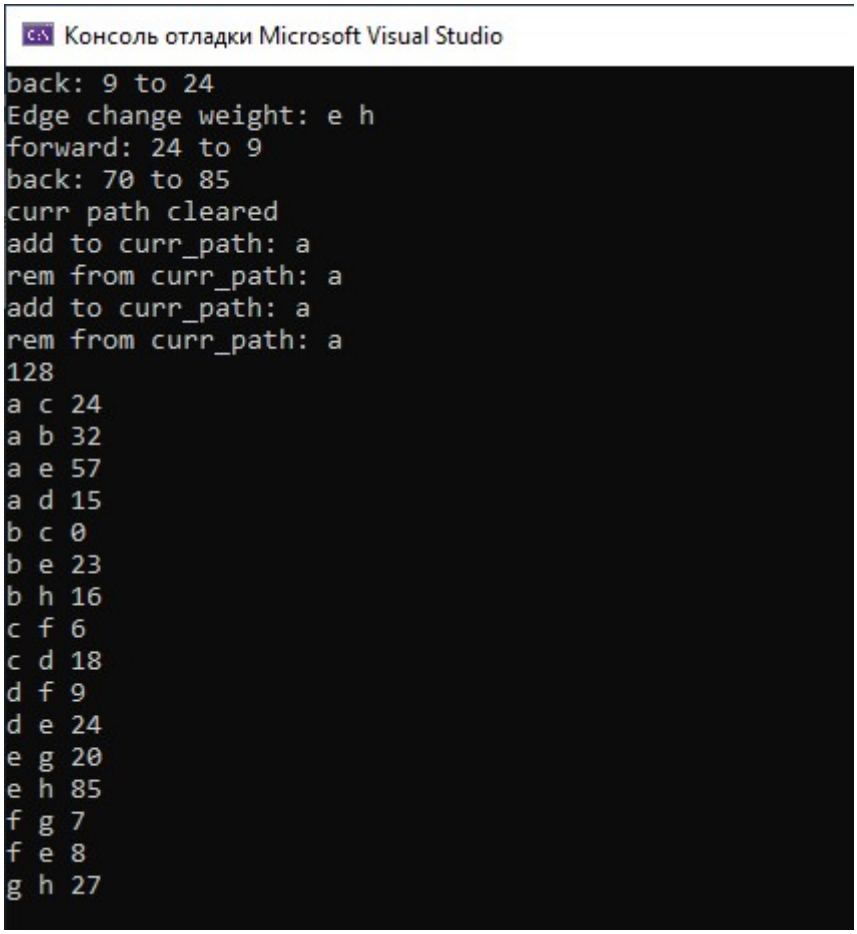





Консоль отладки Microsoft Visual Studio
16
a
h
a c 95
a b 32
a e 57
a d 75
b c 5
b e 23
b h 16
c f 6
c d 18
d f 9
d e 24
e g 20
e h 94
g h 81
f g 7
f e 11
add to curr_path: a
add to curr_path: c
add to curr_path: f
add to curr_path: g
new min: 81
new min: 7
new min: 6
Edge change weight: a c
forward: 95 to 89
back: 0 to 6
Edge change weight: c f
forward: 6 to 0
back: 0 to 6
Edge change weight: f g
forward: 7 to 1
back: 0 to 6
Edge change weight: g h
forward: 81 to 75
back: 0 to 6
curr path cleared
add to curr_path: a
add to curr_path: c
add to curr_path: d
add to curr_path: f
add to curr_path: g
new min: 75
new min: 1
Edge change weight: a c
forward: 89 to 88
back: 6 to 7
Edge change weight: c d
forward: 18 to 17
back: 0 to 1
Edge change weight: d f
forward: 9 to 8
back: 0 to 1
Edge change weight: f g
forward: 1 to 0
back: 6 to 7
Edge change weight: g h
forward: 75 to 74
back: 6 to 7
curr path cleared
add to curr_path: a


Консоль отладки Microsoft Visual Studio
add to curr_path: c
add to curr_path: d
add to curr_path: e
add to curr_path: g
new min: 66
new min: 12
new min: 9
Edge change weight: a c
forward: 80 to 71
back: 15 to 24
Edge change weight: c d
forward: 9 to 0
back: 9 to 18
Edge change weight: d e
forward: 24 to 15
back: 0 to 9
Edge change weight: e g
forward: 12 to 3
back: 8 to 17
Edge change weight: g h
forward: 66 to 57
back: 15 to 24
curr path cleared
add to curr_path: a
rem from curr_path: a
add to curr_path: a
add to curr_path: b
add to curr_path: e
rem from curr_path: e
add to curr_path: e
add to curr_path: g
new min: 57
new min: 3
Edge change weight: a b
forward: 32 to 29
back: 0 to 3
Edge change weight: b e
forward: 23 to 20
back: 0 to 3
Edge change weight: e g
forward: 3 to 0
back: 17 to 20
Edge change weight: g h
forward: 57 to 54
back: 24 to 27
curr path cleared
add to curr_path: a
rem from curr_path: a
add to curr_path: a
add to curr_path: b
add to curr_path: e
rem from curr_path: e
add to curr_path: e
new min: 94
new min: 20
Edge change weight: a b
forward: 29 to 9
back: 3 to 23
Edge change weight: b e
forward: 20 to 0
back: 3 to 23
Edge change weight: e h
forward: 94 to 74


Консоль отладки Microsoft Visual Studio
forward: 94 to 74
back: 0 to 20
curr path cleared
add to curr_path: a
rem from curr_path: a
add to curr_path: a
add to curr_path: b
new min: 16
new min: 9
Edge change weight: a b
forward: 9 to 0
back: 23 to 32
Edge change weight: b h
forward: 16 to 7
back: 0 to 9
curr path cleared
add to curr_path: a
rem from curr_path: a
add to curr_path: a
add to curr_path: e
add to curr_path: b
new min: 7
Edge change weight: a e
forward: 57 to 50
back: 0 to 7
Edge change weight: b h
forward: 7 to 0
back: 9 to 16
curr path cleared
add to curr_path: a
rem from curr_path: a
add to curr_path: a
add to curr_path: e
rem from curr_path: e
add to curr_path: e
rem from curr_path: e
add to curr_path: e
new min: 74
new min: 50
Edge change weight: a e
forward: 50 to 0
back: 7 to 57
Edge change weight: e h
forward: 74 to 24
back: 20 to 70
curr path cleared
add to curr_path: a
rem from curr_path: a
add to curr_path: a
add to curr_path: d
add to curr_path: e
rem from curr_path: e
add to curr_path: e
new min: 24
new min: 15
Edge change weight: a d
forward: 75 to 60
back: 0 to 15
Edge change weight: d e
forward: 15 to 0
back: 9 to 24
Edge change weight: e h
forward: 24 to 9



Выводы.

В ходе выполнения лабораторной работы был изучен и реализован на языке программирования C++ алгоритм Форда-Фалкерсона для поиска максимального потока через сеть.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД

Алгоритм Форда-Фалкерсона.

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

#define INT_MAX    2147483647


struct edge
{
    char first;
    char second;
    int weight;//первоначальный вес
    int forward;//вперёд
    int back;//назад
    bool not_oriented;
};


bool compare(edge first, edge second)
{
    if(first.first == second.first)
        return (int)first.second < (int)second.second;
    return (int)first.first < (int)second.first;
}

struct ways
{
    char start;
    char finish;
    int res;
}way;

class Graph
{
private:
    std::vector <edge> graph;
    std::vector <char> viewedpoints;
    std::vector <char> curr_path;

public:
    Graph()
    {
        int N;
        way.res = 0;
        std::cin >> N >> way.start >> way.finish;
        for(int i = 0; i < N; i++)//заполнения контейнера с рёбрами
        {
            edge element;
            std::cin >> element.first >> element.second >> element.weight;
            element.forward = element.weight;
            element.back = 0;
            element.not_oriented = false;
            graph.push_back(element);
        }
        for(int i = 0; i < graph.size(); i++)
        {
            for(int j = 0; j < graph.size(); j++)
```

```
            {
                if(graph[i].first == graph[j].second )//проверка на двунаправленные
рёбра
                {
                    if (graph[i].second == graph[j].first)
                    {
                        graph[i].back += graph[j].forward;
                        graph[i].not_oriented = true;
                        graph.erase(graph.begin()+j);
                    }
                }
            }
        }
    }

    bool isViewing(char value)
    {
        for(size_t i = 0; i < viewedpoints.size(); i++)
            if(viewedpoints[i] == value)
                return true;
        return false;
    }

    ~Graph()
        {
            graph.clear();
            viewedpoints.clear();
            curr_path.clear();
        }

    bool Search(char value, int& min)
    {
        if(value == way.finish)//сток достигнут
        {
            curr_path.push_back(value);
            return true;
        }
        viewedpoints.push_back(value);
        for(size_t i = 0; i < graph.size(); i++)
        {
            if((isViewing(graph[i].second) || graph[i].forward <= 0) &&
(isViewing(graph[i].first) || graph[i].back <= 0))//пропуск если вершина уже
просмотрена или ребро имеет нулевой вес
                continue;
            if(value == graph[i].first)
            {

                //std::cout << "add to curr_path: " << graph[i].first << "\n";
                curr_path.push_back(graph[i].first);
                if(Search(graph[i].second, min))//рекурсивное продолжение поиска
                {
                    min = (graph[i].forward < min) ? graph[i].forward :
min;//минимальный вес
                    //std::cout << "curr min: " << min << "\n";
                   return true;
                }
                //std::cout << "rem from curr_path: " << curr_path.back() << "\n";
                curr_path.pop_back();
            }
            if(value == graph[i].second)
            {
                //std::cout << "add to curr_path: " << graph[i].second << "\n";
                curr_path.push_back(graph[i].second);
                if(Search(graph[i].first, min))//рекурсивное продолжение поиска
```

```
                {
                    min = (graph[i].back < min) ? graph[i].back : min;//минимальный
вес
                    //std::cout << "curr min: " << min << "\n";
                   return true;
                }
                //std::cout << "rem from curr_path: " << curr_path.back() << "\n";
                curr_path.pop_back();
            }
        }
        return false;
    }


    void FordFalk()
    {
        int min = INT_MAX;
        while(Search(way.start, min))
        {
            for(int i = 1; i < curr_path.size(); i++)
            {
                for(int j = 0; j < graph.size(); j++)
                {
                    if(graph[j].first == curr_path[i-1] && graph[j].second ==
curr_path[i])//пересчёт потока через ребро
                    {
                        //std::cout << "Edge change weight: " << graph[j].first << "
" << graph[j].second;
                        //std::cout << "\nforward: " << graph[j].forward << " to "
<< graph[j].forward - min;
                        //std::cout << "\nback: " << graph[j].back << " to " <<
graph[j].back + min << "\n";
                        graph[j].forward -= min;
                        graph[j].back += min;
                    }
                    if(graph[j].second == curr_path[i-1] && graph[j].first ==
curr_path[i])//пересчёт потока через ребро
                    {
                        //std::cout << "Edge change weight: " << graph[j].first << "
" << graph[j].second;
                        //std::cout << "\nforward: " << graph[j].forward << " to "
<< graph[j].forward + min;
                        //std::cout << "\nback: " << graph[j].back << " to " <<
graph[j].back - min << "\n";
                        graph[j].forward += min;
                        graph[j].back -= min;
                    }
                }
            }
            way.res += min;
            viewedpoints.clear();
            curr_path.clear();
            min = INT_MAX;
        }
    }
    void print_result()
    {
        sort(graph.begin(), graph.end(), compare);//сортировка в лексикографическом
порядке по первой вершине, потом по второй
        std::cout << way.res << std::endl;
        for(int i = 0; i < graph.size(); i++)
        {
            int value = graph[i].weight - graph[i].forward;
            if(value <  0 - graph[i].back)
```

```
            {
                value =  0 - graph[i].back;
            }

            if(graph[i].not_oriented == false)
            {
                std::cout << graph[i].first << " " << graph[i].second << " " <<
value << std::endl;

            }
            else
            {
                if(value < 0)
                    value = 0;
                std::cout << graph[i].first << " " << graph[i].second << " " <<
value << std::endl;
                std::swap(graph[i].first, graph[i].second);
                std::swap(graph[i].back, graph[i].forward);
                graph.at(i).not_oriented = false;
                sort(graph.begin(), graph.end(), compare);
                i--;
            }
        }
    }
};

int main()
{
    Graph* tmp = new Graph();
    tmp->FordFalk();
    tmp->print_result();
    delete tmp;
    return 0;
}
```