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

Студентка гр. 8383 _____ Максимова А.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Разработка программы, решающей задачу поиска максимального потока в сети, а также фактической величины потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона, и соответствующей индивидуальному варианту.

Постановка задач.

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 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).

Входные данные:

7

a

f

a b 7

a c 6

b d 6

c f 9

d e 3

d f 4

e c 2

Выходные данные:

12

a b 6

a c 6

b d 6

c f 8

d e 2

d f 4

e c 2

Вар. 3. Поиск в глубину. Рекурсивная реализация.

Основные теоретические положения.

Алгоритм Форда-Фалкерсона используется для поиска максимального потока в транспортной сети.

Транспортная сеть - ориентированный граф, в котором каждое ребро имеет неотрицательную пропускную способность и поток. Выделяются две вершины исток и сток такие, что любая другая вершина лежит на пути из истока в сток.

Задача о максимальном потоке заключается в нахождении такого потока по транспортной сети, что сумма потоков из истока (сумма потоков в сток) максимальна.

Описание структур данных.

1. *Класс, необходимый для проверки корректности введенных пользователем данных.*

```
class Check {
protected:
    int checkInt() {
        int a = 0;

        while ((!(std::cin >> a)) || a <= 0) {
            std::cout << "Ошибка. Данные некорректны! Ожидается
натуральное число.\n";
            std::cin.clear();
            std::cin.ignore(1000, '\n');
            fflush(stdin);
        }
        std::cin.get();
        return a;
    }

    bool checkChar(char a, char b) {
        if (a == b) {
            std::cout << "Значения истока и стока совпадают.\n";
            return false;
        }
        return true;
    }

    };
```

Методы:

- `int checkInt();` - метод ничего не принимает, используется для считывания целочисленных переменных и их проверки. Если пользователем введена переменная другого типа или целочисленная переменная значения равного или меньшего нуля, то считывания повторяется, до тех пор, пока не будет получено ожидаемое значение. Метод возвращает считанное целочисленное значение.
- `bool checkChar(char a, char b);` - метод, принимающий две переменные типа char. Используется для проверки: если значения переменных a и b равны, то запуск алгоритма не имеет смысла. Метод возвращает true, если значения различны, иначе false.

2. *Класс, используемый для хранения информации о введенных пользователем ребрах графа.*

```
class Edge {
private:
    char vertSt;
    char vertFn;
    int capacity;
    int flow;

public:
    Edge(char st, char fn, int cap) :  vertSt(st), vertFn(fn),
capacity(cap), flow(0) {}
    void setFlow(int flow, int flag) {
        if (flag) this->flow += flow;
        else this->flow -= flow;
    }
    char getVertSt() {
        return vertSt;
    }
    char getVertFn() {
        return vertFn;
    }
    int getCapacity() {
        return capacity;
    }
    int getFlow() {
        return flow;
    }
    };
```

Поля:

- `char vertSt;` - имя вершины из которой исходит ребро.
- `char vertFn;` - имя вершины в которую входит ребро.
- `int capacity;` - пропускная способность ребра, показывающая какая максимальная величина потока может пройти через это ребро (эквивалентна весу ребра).
- `int flow;` - фактическая величина потока в ребре, значение, показывающее сколько величины потока проходит через данное ребро.

Методы:

- `Edge(char st, char fn, int cap);` - конструктор класса, используется для инициализации полей класса.
- `char getVertSt();` - метод, ничего не принимающий на вход. Возвращает значение поля `vertSt`.
- `char getVertFn();` - метод, ничего не принимающий на вход. Возвращает значение поля `vertFn`.
- `int getCapacity();` - метод, ничего не принимает. Возвращает значение поля `capacity`.
- `int getFlow();` - метод, ничего не принимает. Возвращает значение поля `flow`.

3. *Класс, используемый для считывания графа, реализации метода Форда-Фалкерсона и вывода результата его работы.*

* class Ford_Falkerson см. приложение А

Поля:

- `int numberEdges;` – количество ориентированных ребер графа.
- `int maxFlow;` – максимальная величина потока, которая может быть выпущена из истока и которая может пройти через все ребра графа, не вызывая переполнения ни в одном ребре.
- `int char_;` – переменная, используемая для записи вершин в матрицу смежности.
- `int depth;` – целочисленная переменная, используемая для вычисления глубины рекурсии в методе `bool algorithm()`.

- `int depth_;` - целочисленная переменная, используемая для вычисления глубины рекурсии в методе `bool deepSearch(int vertex)`.

- `char source;` - исток, вершина, из которой выходят ребра, ведущие (или входящие в сток напрямую) в другие ребра, ведущие в сток.
- `char sink;` - сток, вершина, в которую ребра только входят.
- `std::vector <int> used;` - список просматриваемых вершин, используемый для поиска в глубину.
- `std::vector <int> path;` - вектор, используемый для хранения вершин, вошедших в текущий путь.
- `std::vector <Edge> listEdge;` - список ребер, введенных пользователем.
- `std::vector <std::vector <int> > matrix;` - матрица смежности.

Методы:

- `Ford_Falkerson();` - конструктор класса, используется для инициализации полей класса.
- `void readMatrix(int flag);` - метод принимает целочисленное значение, равное единице, если ввод данных осуществляется с консоли, и 0 - если с файла.. Необходим для считывания графа. Метод ничего не возвращает.
- `bool algorithm();` - метод, который ничего не принимает. Используется для реализации алгоритма Форда-Фалкерсона (подробное описание алгоритма см. в след. разделах). Возвращает true, когда невозможно найти новый пути из истока в сток, а, следовательно, алгоритм заканчивает свою работу.
- `int findVertex(int vertex);` - метод, принимающий текущую вершину. Используется для поиска смежных вершин, ребра которых открыты (capacity != 0). Метод возвращает вершину, в случае, если она найдена, иначе -1 .
- `void clear();` - метод, который ничего не принимает. Используется

для очистки списка просмотренных вершин и обновления переменной, отвечающей за глубину рекурсии. Метод ничего не возвращает.

- `void printPath();` – метод ничего не принимает. Используется для печати пути, полученного в результате вызова метода bool deepSearch(int vertex). Метод ничего не возвращает.
- `bool deepSearch(int vertex);` – метод, принимающий вершину, от которой ведется поиск. Используется для реализации поиска в глубину (подробное описание алгоритма смотри в след. разделе). Возвращает true, если путь из истока в сток найден, false - если такой путь построить невозможно.
- `int minCapacity();` – метод, который ничего не принимает. Необходим для поиска минимального значения потока, который можно пропустить через все ребра пути, без переполнения. Возвращает минимальное значение потока.
- `void findEdge(char vertexSt, char vertexFn, int flow, int flag);` – метод, принимающий начальную и концевую вершины ребра, фактический проток, пропускаемый через ребро и флаг (в зависимости от того обратное ребро или нет). Используется для поиска расстраиваемого ребра в списке ребер, введенных пользователем. Метод ничего не возвращает.
- `void fillFlow(int flow);` - метод, принимающий значение фактической величины потока. Используется для пропускания потока через ребра (для обычных ребер пропускаемость -= поток, для обратных, наоборот, пропускаемость += поток). Метод ничего не возвращает.
- `static int cmpEdge(Edge a, Edge b);` – компаратор, принимающий два ребра. Используется для сортировки ребер в лексикографическом порядке.
- `void output();`– метод который ничего не принимает. Необходим для вывода промежуточных данных и результата. Ничего не возвращает.

Описание алгоритма.

Для реализации алгоритма Форда-Фалкерсона с целью поиска максимального пути в сети был написан рекурсивный метод `bool algorithm()`.

Метод `bool algorithm()` вызывает методы:

`bool deepSearch(int vertex);` - для нахождения любого пути из истока в сток, не рассматриваемого ранее, с помощью поиска в глубину;

`int minCapacity();` - для нахождения ребра с минимальной пропускной способностью, которая присваивается значению потока, пропускаемого через данный путь;

`void fillFlow(int flow);` - для "пропускания потока через путь", то есть значения пропускных способностей ребер на найденном пути уменьшается на величину потока, а для противоположных им ребер, увеличивается;

`void output();` - для вывода промежуточных и итогового результатов работы алгоритма.

Алгоритм можно представить в виде последовательности шагов:

Подготовка: создание матрицы смежности,. используемой для хранения графа, и заполнение ее нулями (значит, что пропускная способность ребра между двум вершинами равна 0). Добавление в матрицу взвешенных ребер графа, введенных пользователем.

Шаг 1: находим любой путь из истока в сток, запуская алгоритм поиска в глубину (от истока, предварительно добавленный в путь).

Шаг 1.1: помечаем обрабатываемую вершину как посещенную.

Шаг 1.2: для обрабатываемой вершины находим первую смежную и еще не посещенную, до которой можно дойти по ребру, пропускная способность которого не равна нулю.

Шаг 1.2.1: если такая вершина найдена, добавляем ее в путь и запускаем поиск в глубину от нее. Возвращаемся к шагу 1.1.

Шаг 1.2.2: иначе запускаем поиск от предыдущей посещенной вершины, убирая вершину из пути и закрывая ее для дальнейшего рассмотрения. Если таких вершин нет, заканчиваем поиск (путей от истока в сток больше нет).

Шаг 2: на найденном пути находим значение минимальной пропускной способности ребра, входящего в путь. Присваиваем это значение потоку, который будем пропускать.

Шаг 3: пропускаем поток через найденный путь. Для каждого ребра, входящего в путь, уменьшаем пропускную способность на величину потока, а для обратного ребра - увеличиваем. Если рассматриваемое ребро входит в список ребер, введенных пользователем, сохраняем значение потока (увеличиваем поток для ребра, а для обратного уменьшаем).

Шаг 4: стираем найденный путь и возвращаемся к шагу 1.

Иллюстрация работы алгоритма:

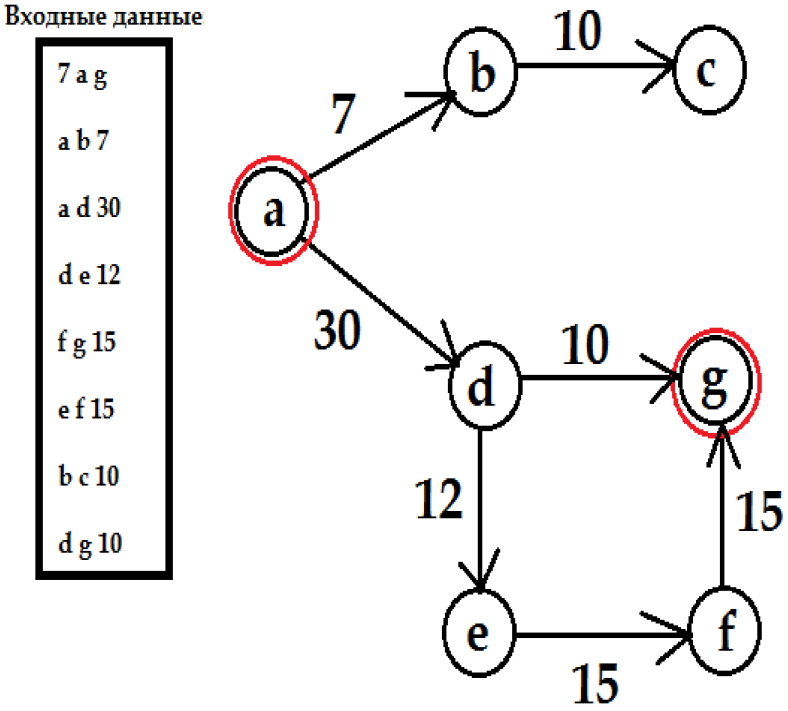


Рисунок 1 - Входные данные

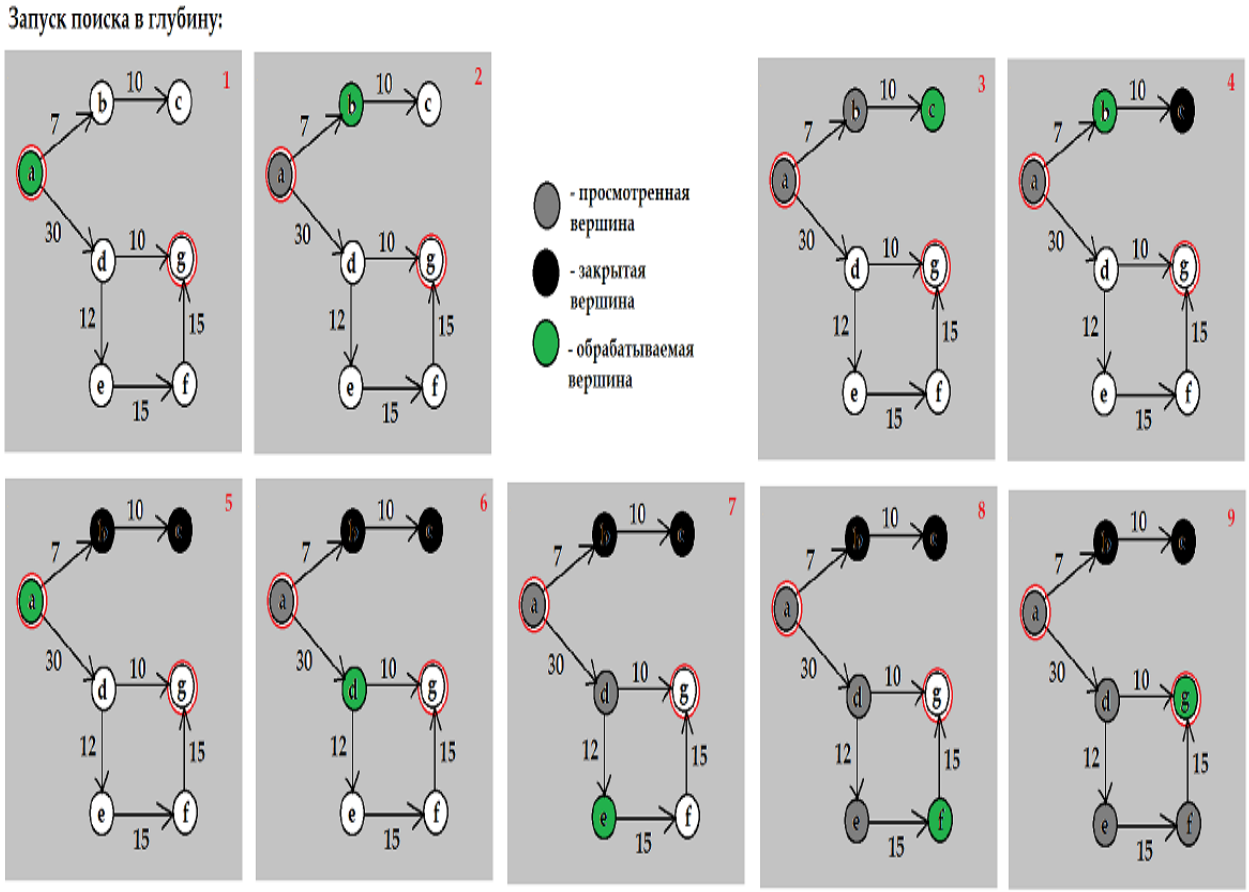


Рисунок 2 - Поиск в глубину

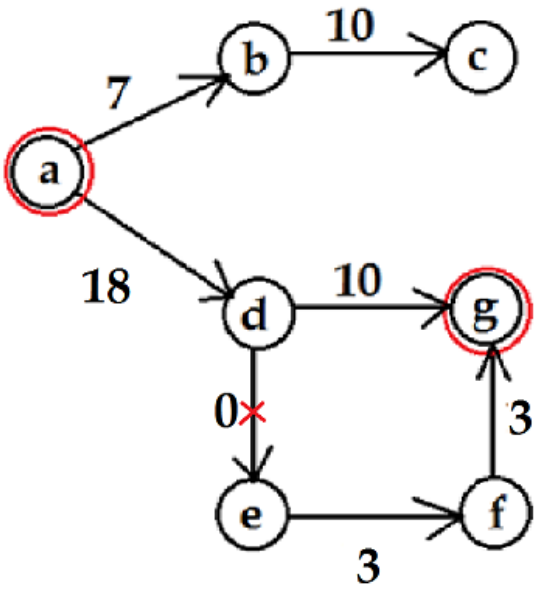


Рисунок 3 - Состояние графа после пропускания потока

Возвращаемся к шагу 1, максимальный поток в результате будет равен 22.

Оценка сложности алгоритма.

Так как задачей алгоритма является поиск максимального потока, который можно пропустить через сеть, без переполнения, то в худшем случае, если пропускные способности всех ребер - целочисленные значения (в ином случае алгоритм может работать бесконечно), на каждом шаге будет найден увеличивающий путь, по которому можно пропустить поток минимум равный единице. Следовательно, при максимальном потоке P_{max} поиск в глубину будет вызван P_{max} раз в худшем случае.

При этом сложность поиска в глубину по времени равна $O(V^2)$, при способе хранения графа в виде матрицы смежности, где V - количество вершин, так как в худшем случае поиск в глубину рекурсивно будет вызван не больше, чем V раз, так как в программе хранится список рассмотренных вершин, а внутри функции, для каждой вершины ищется смежная - то есть проверяется не более V вершин.

Таким образом, сложность алгоритма по времени равна $O(V^2 \cdot P_{max})$.

По памяти сложность алгоритма можно оценить как $O(V)$, так как в процесс работы алгоритм хранит путь, который в худшем случае может состоять из всех вершин, входящих в граф.

Тестирование.

Подробный тест.

Входные данные.

7

a

f

a b 7

a c 6

b d 6

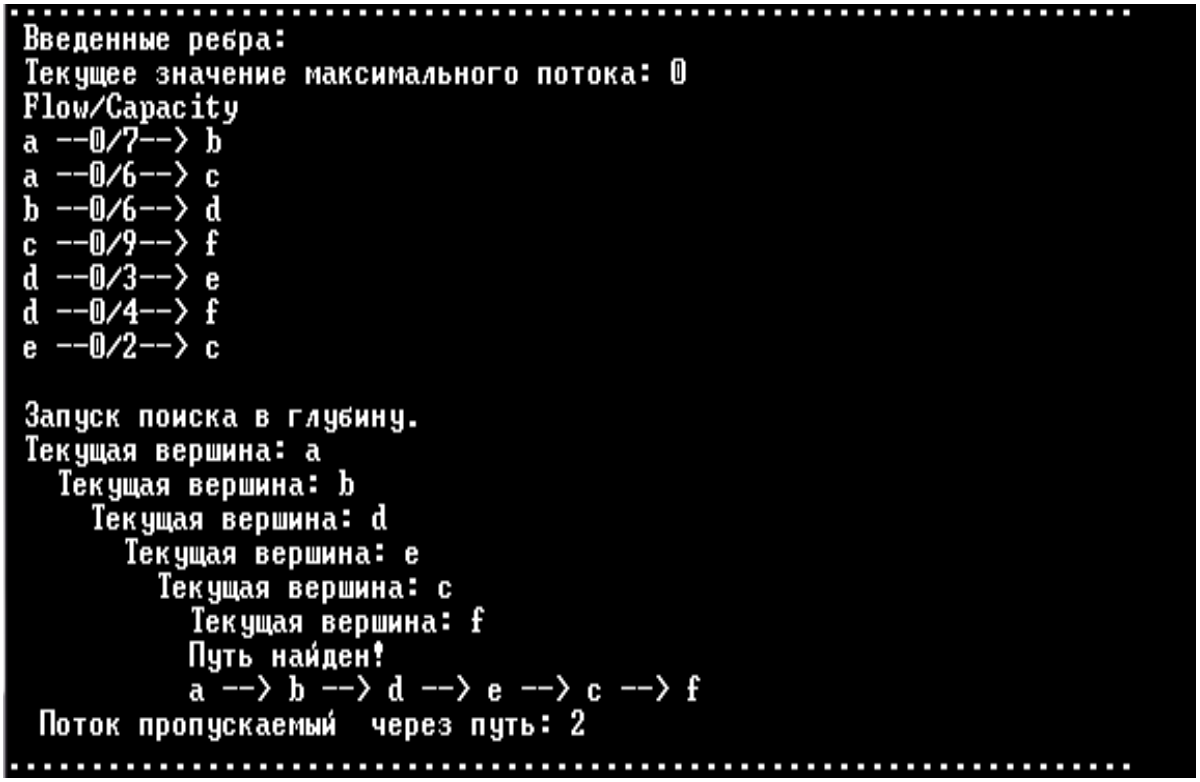
c f 9

d e 3

d f 4

e c 2

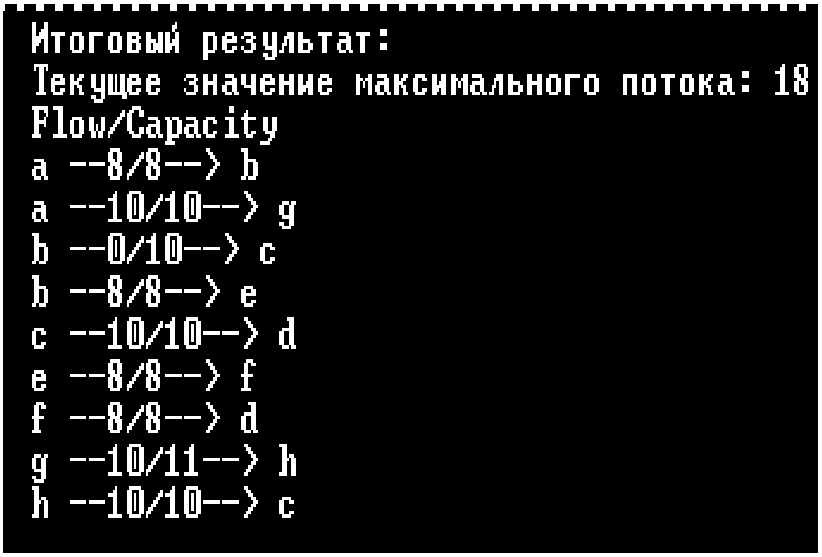
Выходные данные.

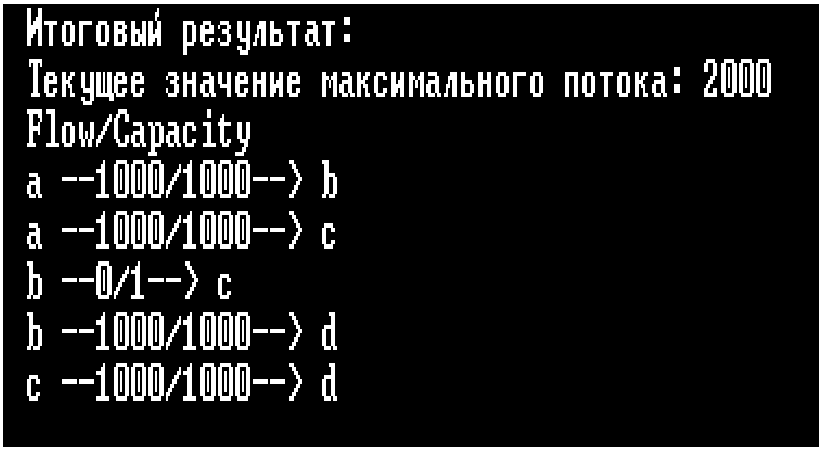


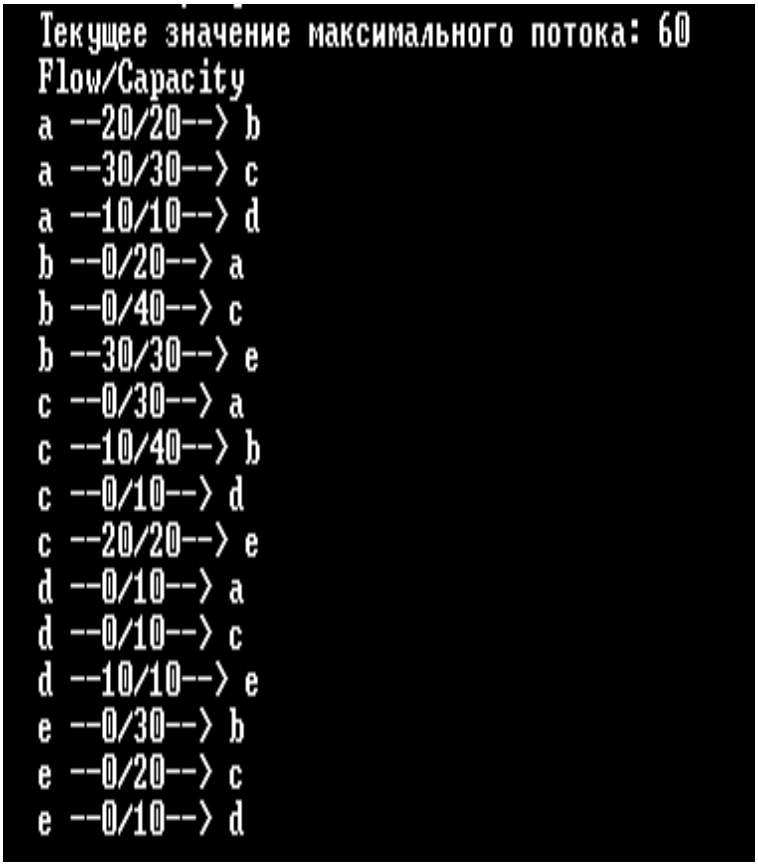
```
Текущее значение максимального потока: 2
Flow/Capacity
a --2/7--> b
a --0/6--> c
b --2/6--> d
c --2/9--> f
d --2/3--> e
d --0/4--> f
e --2/2--> c

Запуск поиска в глубину.
Текущая вершина: a
  Текущая вершина: b
    Текущая вершина: d
      Текущая вершина: e
    Возвращаемся к предыдущей
    Текущая вершина: d
      Текущая вершина: f
      Путь найден!
      a --> b --> d --> f
  Поток пропускаемый  через путь: 4
------------------------------------------------------------
  Промежуточный результат:
  Текущее значение максимального потока: 6
  Flow/Capacity
  a --6/7--> b
  a --0/6--> c
  b --6/6--> d
  c --2/9--> f
  d --2/3--> e
  d --4/4--> f
  e --2/2--> c

  Запуск поиска в глубину.
  Текущая вершина: a
    Текущая вершина: b
  Возвращаемся к предыдущей
  Текущая вершина: a
    Текущая вершина: c
      Текущая вершина: e
        Текущая вершина: d
      Возвращаемся к предыдущей
      Текущая вершина: e
    Возвращаемся к предыдущей
    Текущая вершина: c
      Текущая вершина: f
      Путь найден!
      a --> c --> f
    Поток пропускаемый  через путь: 6
------------------------------------------------------------
    Промежуточный результат:
    Текущее значение максимального потока: 12
    Flow/Capacity
    a --6/7--> b
    a --6/6--> c
    b --6/6--> d
    c --8/9--> f
    d --2/3--> e
    d --4/4--> f
    e --2/2--> c

    Запуск поиска в глубину.
    Текущая вершина: a
      Текущая вершина: b
    Возвращаемся к предыдущей
    Текущая вершина: a
    Путь не найден!
------------------------------------------------------------
    Итоговый результат:
    Текущее значение максимального потока: 12
    Flow/Capacity
    a --6/7--> b
    a --6/6--> c
    b --6/6--> d
    c --8/9--> f
    d --2/3--> e
    d --4/4--> f
    e --2/2--> c
```

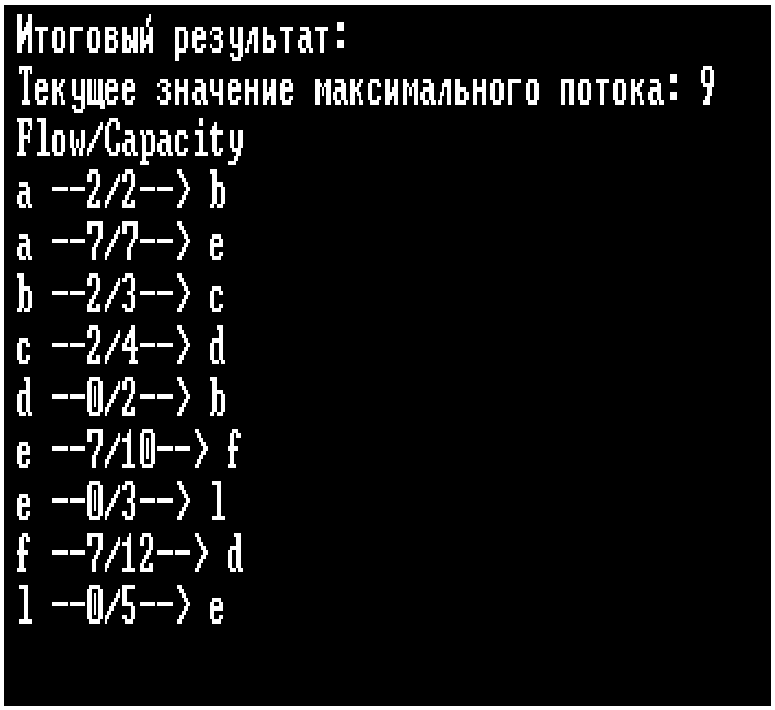
Таблица тестирования.

Входные данные	**Выходные данные**
31 a n a b 6 a c 6 a d 8 a e 9 b c 3 c d 4 d e 3 b f 4 c g 4 d h 5 e i 6 f c 9 g d 10 d i 10 i h 7 h g 8 g f 8 f j 5 g k 5 h l 5 i m 7	Итоговый результат: Текущее значение максимального потока: 26 Flow/Capacity a --6/6--> b a --6/6--> c a --8/8--> d a --6/9--> e b --2/3--> c b --4/4--> f c --4/4--> d c --4/4--> g d --0/3--> e d --5/5--> h d --7/10--> i e --6/6--> i f --0/9--> c f --5/5--> j g --0/10--> d g --1/8--> f g --5/5--> k h --0/8--> g h --5/5--> l h --6/12--> m i --2/8--> g i --6/7--> h i --5/7--> m j --6/6--> n k --0/12--> h k --1/8--> j k --6/9--> n l --2/8--> k l --8/8--> n m --5/7--> l m --6/6--> n

i g 8 k h 12 h m 12 m l 7 l k 8 k j 8 j n 6 k n 9 l n 8 m n 6	
9 a d a b 8 b c 10 c d 10 h c 10 e f 8 g h 11 b e 8 a g 10 f d 8	

5 a d a b 1000 a c 1000 b c 1 b d 1000 c d 1000	
8 a a a b 2 b c 3 c d 4 a e 7 e f 10 f d 12 e l 3 l e 5	Значения истока и стока совпадают.
3 *a e*	Нет ребер из истока.

b c 1 *c b 1* *c e 1*	
16 *a e* *a b 20* *b a 20* *a d 10* *d a 10* *a c 30* *c a 30* *b c 40* *c b 40* *c d 10* *d c 10* *c e 20* *e c 20* *b e 30* *e b 30*	

d e 10 *e d 10*	
9 a d d b 2 a b 2 b c 3 c d 4 a e 7 e f 10 f d 12 e l 3 l e 5	
0 a c a b 1 b c 1	Ошибка. Данные некорректны! Ожидается натуральное число.
3 a d	Ошибка. Данные некорректны! Ожидается натуральное число.

a b 0 b c 0 c d 0	
3 a d a b 3 b a 6 c d 1	Итоговый результат: Текущее значение максимального потока: 0 Flow/Capacity a --0/3--> b b --0/6--> a c --0/1--> d
13 1 9 1 2 3 1 5 10 1 4 15 2 3 4 3 5 7 4 5 8 7 3 2 3 6 5 6 7 2	Итоговый результат: Текущее значение максимального потока: 13 Flow/Capacity 1 --3/3--> 2 1 --8/15--> 4 1 --2/10--> 5 2 --3/4--> 3 3 --0/7--> 5 3 --3/5--> 6 4 --8/8--> 5 5 --4/5--> 6 5 --6/9--> 8 6 --0/2--> 7 6 --7/7--> 9 7 --0/2--> 3 8 --6/6--> 9

5 6 5 5 8 9 6 9 7 8 9 6	

Вывод.

В результате выполнения лабораторной работы была написана программа, реализующая алгоритм Форда-Фалкерсона, решающего задачу поиска максимального потока в сети, а также вычисляющего фактическую величину потока протекающего через каждое ребро.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД АЛГОРИТМА ФОРДА-ФАЛКЕРСОНА

```
#include <algorithm>

#include <iostream>

#include <iomanip>

#include <fstream>

#include <vector>

#define SIZE 26


std::ifstream infile("test.txt");


class Check {                                   //для проверки
корректности введенных данных

protected:

    int checkInt() {                            //ожидается
положительное число

        int a = 0;


        while ((!(std::cin >> a)) || a <= 0) {

            std::cout << "Ошибка. Данные некорректны! Ожидается
натуральное число.\n";

            std::cin.clear();

            std::cin.ignore(1000, '\n');

            fflush(stdin);

        }

        std::cin.get();

        return a;
```

```
    }


    bool checkChar(char a, char b) {        //если исток равен
стоку, то запуск алгоритма не имеет смысла
        if (a == b) {
            std::cout << "Значения истока и стока совпадают.\n";
            return false;
        }
        return true;
    }


};


class Edge {
private:
    char vertSt;
    char vertFn;
    int capacity;
    int flow;


public:
    Edge(char st, char fn, int cap) :  vertSt(st), vertFn(fn),
capacity(cap), flow(0) {}
    void setFlow(int flow, int flag) {
        if (flag) this->flow += flow;
        else this->flow -= flow;
    }
```

```
    char getVertSt() {
        return vertSt;
    }
    char getVertFn() {
        return vertFn;
    }
    int getCapacity() {
        return capacity;
    }
    int getFlow() {
        return flow;
    }
};

class Ford_Falkerson : protected Check {
private:
    int numberEdges;
//количество ориентированных рёбер графа
    int maxFlow;
//максимальная величина потока
    int char_;                                                    //для
записи вершин в матрицу

    int depth;
    int depth_;

    char source;
```

```
    char sink;


    std::vector <int> used;
//просмотренные вершины

    std::vector <int> path;
//построенный путь

    std::vector <Edge> listEdge;                                    //для
хранения ребер графа, введенных пользователем

    std::vector <std::vector <int> > matrix;
//матрица смежности взвешенного графа




    int findVertex(int vertex) {                                    //для
поиска смежных вершин

        for (int i = 0; i < SIZE; i++) {

            if (!used[i] && matrix[vertex][i] != 0) {       //не
посещенные, ребра между которыми не закрыты

                return i;

            }

        }

        return -1;

    }


    void clear() {                                                   //все
вершины снова непросмотренные

        depth_ = depth;

        for (int i = 0; i < used.size(); i++)

            used[i] = 0;

    }
```

```
    void printPath() {
        for (auto now : path)
            if (now != int(sink - char_)) std::cout << char(now + char_) << " --> ";
            else std::cout << char(now + char_) << "\n";
    }

    bool deepSearch(int vertex) {
//поиск в глубину: изначально все вершины помечены как не посещенные
        depth_ += 2;
        std::cout << std::setw(depth_) << ' ' << "Текущая вершина: " << char(vertex + char_) << "\n";

        if (vertex == (int)sink - char_) {                  //если дошли до стока
            std::cout << std::setw(depth_) << ' ' <<  "Путь найден!\n";
            std::cout << std::setw(depth_) << ' ';
            printPath();
            clear();
            return true;
        }
        else {
            used[vertex] = 1;
//помечаем как посещенную
            int index = findVertex(vertex);
            if (index != -1) {
```

```
                path.push_back(index);
//добавляем в путь

                return deepSearch(index);
//продолжаем поиск от этой вершины

            }

            else {                                              //если
таких вершин нет, берем предыдущую из списка посещенных

                used[vertex] = 2;
//больше не рассматриваем

                path.pop_back();



                if (path.empty()) {

                    if (findVertex((int)source - char_) == -1) {

                        std::cout << std::setw(depth_) << ' ' <<
"Путь не найден!\n";

                        return false;    //если невозможно
построить путь

                    }

                }

                depth_ -= 4;

                std::cout << std::setw(depth_ + 2) << ' ' <<
"Возвращаемся к предыдущей\n";

                return deepSearch(path.back());

            }

        }

    }



    int minCapacity() {                                         //поиск
максимально возможного потока через найденный путь

        int min = 9999;
```

```
        for (int i = 0, j = 1; j < path.size(); i++, j++)
            if (matrix[path[i]][path[j]] < min)
                min = matrix[path[i]][path[j]];
        return min;
    }

    void findEdge(char vertexSt, char vertexFn, int flow, int
flag) { //если рассматриваемое ребро входит в список ребер,
введенных пользователем, то запоминаем пропущенный поток
        for (int i = 0; i < listEdge.size(); i++) {
            if (listEdge[i].getVertSt() == vertexSt &&
listEdge[i].getVertFn() == vertexFn)
                listEdge[i].setFlow(flow, flag);
        }
    }

    void fillFlow(int flow) {
//пропускание потока через путь
        for (int i = 0, j = 1; j < path.size(); i++, j++) {
            matrix[path[i]][path[j]] -= flow;
//уменьшение пропускной способности на величину потока
            findEdge((char)(path[i] + char_), (char)(path[j] +
char_), flow, 1);

            matrix[path[j]][path[i]] += flow;           //обратные
ребра
            findEdge((char)(path[j] + char_), (char)(path[i] +
char_), flow, 0);
        }
    }
```

```
    static int cmpEdge(Edge a, Edge b) {                    //для
сортировки

        if (a.getVertSt() == b.getVertSt()) return a.getVertFn() <
b.getVertFn();

        return a.getVertSt() < b.getVertSt();

    }


public:

    Ford_Falkerson() : numberEdges(0), maxFlow(0), depth(-1),
depth_(-1), source('0'), sink('0'), char_(), used(SIZE, 0),
matrix(SIZE, std::vector<int>(SIZE, 0)) {}



    void readMatrix(int flag) {

        if (flag) {

            std::cout << "Введите количество ориентированных ребер
графа.\n";

            numberEdges = checkInt();

            std::cout << "Введите исток и сток графа.\n";

            std::cin >> source >> sink;

        }

        else {

            infile >> numberEdges;

            if (numberEdges <= 0) {

                std::cout << "Ошибка. Данные некорректны!
Ожидается натуральное число.\n";

                exit(1);

            }

            infile >> source >> sink;
```

```
        }

        source = tolower(source);
        sink = tolower(sink);

        if (checkChar(source, sink)) {
            char vertex1;
            char vertex2;
            int weight;

            do {
                if (flag) {
                    std::cin >> vertex1 >> vertex2;
                    weight = checkInt();
                }
                else {
                    infile >> vertex1 >> vertex2;
                    infile >> weight;
                    if (weight <= 0) {
                        std::cout << "Ошибка. Данные некорректны!
Ожидается натуральное число.\n";
                        exit(1);
                    }
                }

                vertex1 = tolower(vertex1);
                vertex2 = tolower(vertex2);
```

```
                if (isalpha(vertex1)) char_ = 97;
//если граф записывается с помощью букв

                else char_ = 49;
//цифр


                matrix[vertex1 - char_][vertex2 - char_] = weight;
//заносим в матрицу


                Edge edge(vertex1, vertex2, weight);
//и в список ребер

                listEdge.push_back(edge);


                numberEdges--;

            } while (numberEdges != 0);


            int flag2 = 0;
//если из истока нет ребер, запуск алгоритма бессмысленен

            for (int i = 0; i < SIZE; i++)

                if (matrix[source - char_][i] != 0) flag2 = 1;

            if (!flag2) {

                std::cout << "Нет ребер из истока.\n"; exit(1);

            }

            std::cout <<
"..............................................................
...\n";

            std::cout << " Введенные ребра: \n";

            depth++;

            output();
```

```
        }

        else {

            exit(1);
//если исток = стоку

        }

    }


    bool algorithm() {
//реализация алгоритма Форла-Фалкерсона

        path.push_back((int)source - char_);
//добавляем исток в путь

        std::cout << std::setw(depth) << ' ' << "Запуск поиска в
глубину.\n";

        if (!deepSearch((int)source - char_)) {
//путей больше нет

            sort(listEdge.begin(), listEdge.end(), cmpEdge);

            std::cout <<
".....................................................................
...\n";

            std::cout << std::setw(depth) << ' ' << "Итоговый
результат:\n";

            output();

            return true;

        }

        else {

            depth += 2;

            std::cout << std::setw(depth) << ' ' << "Поток
пропускаемый  через путь: " << minCapacity() << "\n";

            fillFlow(minCapacity());
//пропускание потока
```

```
            path.clear();

            std::cout <<
".....
...\n";

            std::cout << std::setw(depth) << ' ' << "Промежуточный
результат:\n";

            output();

            return algorithm();

        }

    }



    void output() {

        for (int i = 0; i < listEdge.size(); i++)

            if (listEdge[i].getVertFn() == sink) maxFlow +=
listEdge[i].getFlow();

        std::cout << std::setw(depth) << ' ' << "Текущее значение
максимального потока: " << maxFlow << "\n";

        maxFlow = 0;



        std::cout << std::setw(depth) << ' ' << "Flow/Capacity\n";

        for (int i = 0; i < listEdge.size(); i++) {

            if (listEdge[i].getFlow() > 0)

                std::cout << std::setw(depth) << ' ' <<
listEdge[i].getVertSt() << " --" << listEdge[i].getFlow() << "/"
<< listEdge[i].getCapacity() << "--> " << listEdge[i].getVertFn()
<< "\n";

            else

                std::cout << std::setw(depth) << ' ' <<
listEdge[i].getVertSt() << " --0/" << listEdge[i].getCapacity() <<
"--> " << listEdge[i].getVertFn() << "\n";
```

```
        } std::cout << "\n";


        //stepik

        /*for (int i = 0; i < listEdge.size(); i++)

            if (listEdge[i].getVertFn() == sink) maxFlow +=
listEdge[i].getFlow();

        std::cout << maxFlow << "\n";

        for (int i = 0; i < listEdge.size(); i++) {

            if (listEdge[i].getFlow() > 0)

                std::cout << listEdge[i].getVertSt() << " " <<
listEdge[i].getVertFn() << " " << listEdge[i].getFlow() << "\n";

            else

                std::cout << listEdge[i].getVertSt() << " " <<
listEdge[i].getVertFn() << " " << '0' << "\n";

        }*/

    }

};


int main()

{

    setlocale(LC_ALL, "ru");

    char answer;

    int exit = 0;

    Ford_Falkerson start;


    do {

        std::cout << "Введите y, если будете вводить граф с
консоли\n";
```

```
        std::cout << "n, если из файла.\n";
        std::cin >> answer;
        answer = tolower(answer);

        switch (answer) {
        case 'y':
            start.readMatrix(1);
            break;

        case 'n':
            start.readMatrix(0);
            break;

        default:
            std::cout << "Ошибка: введен неккоректный символ.
Попробуйте еще раз.\n";
            exit = 1;
        }
    } while (exit == 1);

    start.algorithm();
    return 0;
}
```