

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: Максимальный поток

Студент гр. 9383

Гладких А.А.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Применить на практике знания о построение алгоритма Форда-Фалкерсона, реализовать алгоритм Форда-Фалкерсона для поиска максимального потока в заданном графе.

Основные теоретические положения.

Сеть — ориентированный граф, в котором каждое ребро имеет положительную пропускную способность. Сеть имеет исток и сток.

Вершина ориентированного графа называется истоком, если в неё не входит ни одно ребро, и стоком, если из неё не выходит ни одного ребра.

Поток — функция, удовлетворяющая условиям:

- 1) Антисимметричность
- 2) Ограничение пропускной способности
- 3) Выполняется закон сохранения потока.

Величина потока определяется как сумма весов всех ребер, входящих в поток.

Максимальный поток — поток наибольшей мощности в графе.

Алгоритм Форда — Фалкерсона - алгоритм нахождения максимального потока в транспортной сети.

Идея алгоритма заключается в следующем. Изначально величине потока присваивается значение 0 для всех вершин. Затем величина потока итеративно увеличивается посредством поиска увеличивающего пути (путь от источника s к стоку t , вдоль которого можно послать больший поток). Процесс повторяется, пока можно найти увеличивающий путь.

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

Выходные данные:

P_{\max} - величина максимального 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).

Вариант 5 - Поиск не в глубину и не в ширину, а по правилу: каждый раз выполняется переход по дуге, имеющей максимальную остаточную пропускную способность. Если таких дуг несколько, то выбрать ту, которая была обнаружена раньше в текущем поиске пути.

Ход работы:

1. Произведён анализ задания.
2. Был реализован алгоритм Форда-Фалкерсона:
 1. С помощью обхода графа по заданным в задании правилам программа рекурсивно проверяет существование пути от вершины истока к вершине стока, параллельно считая поток. Пройденные вершины программа помечает и записывает в структуру *map* стандартной библиотеки языка C++ для последующего восстановления пути.

2. Пока путь из пункта 1 существует, программа обновляет веса ребер, через который проходит поток: уменьшает пропускную способность ребер по пути и увеличивает пропускную способность обратных к ним ребер. При этом поток для каждого пути суммируется в отдельной переменной.
3. Когда путей от истока к стоку не будет, алгоритм заканчивает свою работу и выводит величину максимального потока.
3. Сложность алгоритма по памяти линейна — $O(E)$, где E — количество ребер в графе, а по времени сложность можно оценить как $O(E * \text{max_flow})$, где max_flow — максимальный поток в сети. Также, чтобы временная сложность не зависела от заранее неизвестного значения, ее можно оценить как $O(E * S)$, где S — сумма ребер, приходящих в сток.
4. Были написаны тесты с использованием библиотеки Catch2 для функций взаимодействия с графом: было протестировано создание графа и считывание данных о его вершинах, работа функции поиска пути от истока к стоку и непосредственно работа самого алгоритма Форда-Фалкерсона.
5. Код разработанной программы расположен в Приложении А.

Описание функций и структур данных.

1. Структура `Edge` — представление ребра графа. Имеет поля *flow* и *usedFlow* - для запоминания пропускной способности ребра и задействованной пропускной способности соответственно, *index* — для хранения численного индекса ребра — это нужно, чтобы правильно сопоставлять ребро с обратным ему, если были введены несколько ребер между двумя вершинами, булевы переменные *isInit* и *isVisited* — для хранения информации о том, было ли данное ребро изначально введено и было ли данное ребро посещено при построении потока.
2. Класс `Graph` — представление графа. Имеет следующие поля — *source_symbol_*, *sink_symbol_* - для хранения названия вершины истока и

вершины стока, *edges_amount_* для хранения количества ребер, *graph_map_* для хранения графа в структуре *map* стандартной библиотеки языка C++ и *prev_node_map_* - для хранения того, из какой вершины пришли в текущую. Также класс имеет следующие методы — *read_edges()* – метод считывания данных о ребрах из потока, метод *to_string()*, который возвращает представление графа в виде строки, метод *print_res_graph()*, который возвращает строку с представлением графа после работы алгоритма, метод *ford_fulkerson()* - реализация алгоритма Форда-Фалкерсона, метод *find_flow_path()* - метод, который ищет путь от истока к стоку и параллельно считает величину потока по этому пути, а также метод *get_flow_path()*, который по полю *prev_node_map_* создает структуру *vector* из стандартной библиотеки, в которой хранится получившийся путь.

3. Функция *main()* - функция, в которой происходит инициализация графа и запуск алгоритма.

Примеры работы программы.

Таблица 1 – Пример работы программы

№ п/п	Входные данные	Выходные данные
1.	7 a f a b 7 a c 6 b d 6 c f 9 d e 3 d f 4 e c 2	12 a b 6 a c 6 b d 6 c f 8 d e 2 d f 4 e c 2
2.	5 1 4 1 2 20 1 3 1 2 3 20 2 4 1 3 4 20	21 1 2 20 1 3 1 2 3 19 2 4 1 3 4 20
3.	7 a d a b 20 b c 10 c b 10 b c 20 c b 5 b c 5 c d 30	20 a b 20 b c 20 b c 0 b c 0 c b 0 c b 0 c d 20

Иллюстрация работы программы.

```
7
a
f
a b 7
a c 6
b d 6
c f 9
d e 3
d f 4
e c 2
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2
```

Рисунок 1 - Пример работы программы на входных данных №1

```
5
1
4
1 2 20
1 3 1
2 3 20
2 4 1
3 4 20
21
1 2 20
1 3 1
2 3 19
2 4 1
3 4 20
```

Рисунок 2 - Пример работы программы на входных данных №2

```
7
a
d
a b 20
b c 10
c b 10
b c 20
c b 5
b c 5
c d 30
20
a b 20
b c 20
b c 0
b c 0
c b 0
c b 0
c d 20
```

Рисунок 3 - Пример работы программы на входных данных №3

Выводы.

Были применены на практике знания о построение алгоритма Форда-Фалкерсона. На языке программирования C++ был реализован алгоритм Форда-Фалкерсона для поиска максимального потока в заданном графе. При этом поиск пути в алгоритме осуществлялся не в глубину и не в ширину, а по заданному правилу — при поиске пути каждый раз выполняется переход по дуге, имеющей максимальную остаточную пропускную способность. Написанная программа была протестирована на различных входных данных.

ПРИЛОЖЕНИЕ А

ФАЙЛ MAIN.CPP

```
#INCLUDE <IOSTREAM>

#include "GRAPH.HPP"

#include <SSTREAM>

INT MAIN() {

    INT EDGES_AMOUNT;
    CHAR SOURCE_NODE, SINK_NODE;

    STD::CIN >> EDGES_AMOUNT >> SOURCE_NODE >> SINK_NODE;

    GRAPH GRAPH(SOURCE_NODE, SINK_NODE, EDGES_AMOUNT);
    BOOL HAS_READ = GRAPH.READ_EDGES(STD::CIN);
    IF(!HAS_READ) {
        STD::COUT << "CAN'T READ\n";
        RETURN 0;
    }

    STD::COUT << GRAPH.FORD_FULKERSON() << '\n';

    STD::COUT << GRAPH.PRINT_RES_GRAPH();

    RETURN 0;
}
```

ФАЙЛ GRAPH.HPP

```
#ifndef GRAPH_HPP
#define GRAPH_HPP

#include <IOSTREAM>
#include <VECTOR>
#include <QUEUE>
#include <MAP>
#include <STRING>
#include <ALGORITHM>

STRUCT EDGE{
    INT FLOW = 0;
    INT USED_FLOW = 0;
    INT INDEX = 0;
    BOOL IS_INIT = FALSE;
    BOOL IS_VISITED = FALSE;
};

USING NEIGHBOUR = STD::PAIR<CHAR, EDGE>;

INLINE BOOL NEIGHBOUR_CMP(NEIGHBOUR A, NEIGHBOUR B);

CLASS GRAPH{
```

```

PUBLIC:

    GRAPH (CHAR SOURCE_NODE_SYMBOL, CHAR SINK_NODE_SYMBOL, INT EDGES_AMOUNT);
    ~GRAPH ();

    BOOL READ_EDGES (STD::ISTREAM& IN);

    STD::STRING TO_STRING ();

    STD::STRING PRINT_RES_GRAPH ();

    INT FORD_FULKERSON ();

    STD::VECTOR<CHAR> TEST_FIND_PATH (CHAR START_SYMBOL, CHAR END_SYMBOL);

PRIVATE:
    CHAR SOURCE_SYMBOL_;
    CHAR SINK_SYMBOL_;
    INT EDGES_AMOUNT_;

    STD::MAP<CHAR, STD::VECTOR<NEIGHBOUR>> GRAPH_MAP_;
    STD::MAP<CHAR, CHAR> PREV_NODE_MAP_;

    INT FIND_FLOW_PATH (CHAR START_SYMBOL, CHAR END_SYMBOL, STD::MAP<CHAR, BOOL>
IS_VISITED_MAP, INT RESULT);
    STD::VECTOR<CHAR> GET_FLOW_PATH ();

};

#endif

```

ФАЙЛ GRAPH.CPP

```

#include "GRAPH.HPP"

GRAPH::GRAPH (CHAR SOURCE_NODE_SYMBOL, CHAR SINK_NODE_SYMBOL, INT EDGES_AMOUNT) {
    SOURCE_SYMBOL_ = SOURCE_NODE_SYMBOL;
    SINK_SYMBOL_ = SINK_NODE_SYMBOL;
    EDGES_AMOUNT_ = EDGES_AMOUNT;
}

BOOL GRAPH::READ_EDGES (STD::ISTREAM& IN) {
    CHAR EDGE_START_NAME, EDGE_END_NAME;
    INT MAX_FLOW;
    FOR (INT I = 0; I < EDGES_AMOUNT_; I++) {
        IN >> EDGE_START_NAME >> EDGE_END_NAME >> MAX_FLOW;

        IF (MAX_FLOW < 0) {
            RETURN FALSE;
        }

        GRAPH_MAP_[EDGE_START_NAME].PUSH_BACK ({EDGE_END_NAME, {MAX_FLOW, 0, I,
TRUE}});
        GRAPH_MAP_[EDGE_END_NAME].PUSH_BACK ({EDGE_START_NAME, {0, MAX_FLOW,
I}});
    }
}

```

```

    }

    RETURN TRUE;
}

STD::STRING GRAPH::TO_STRING() {
    STD::STRING OUT;
    FOR (AUTO CHAR_VEC_PAIR = GRAPH_MAP_.BEGIN(); CHAR_VEC_PAIR !=
GRAPH_MAP_.END(); ++CHAR_VEC_PAIR) {
        OUT += CHAR_VEC_PAIR->FIRST;
        OUT += ": ";
        FOR(AUTO NEIGHBOUR = CHAR_VEC_PAIR->SECOND.BEGIN(); NEIGHBOUR !=
CHAR_VEC_PAIR->SECOND.END(); ++NEIGHBOUR) {
            OUT += NEIGHBOUR->FIRST;
            OUT += "(";
            OUT += STD::TO_STRING(NEIGHBOUR->SECOND.FLOW);
            OUT += "/";
            OUT += STD::TO_STRING(NEIGHBOUR->SECOND.USED_FLOW);
            OUT += ") ";
        }
        OUT += "\n";
    }
    RETURN OUT;
}

BOOL NEIGHBOUR_CMP(NEIGHBOUR A, NEIGHBOUR B) {
    RETURN A.SECOND.FLOW > B.SECOND.FLOW;
}

INT GRAPH::FIND_FLOW_PATH(CHAR START_SYMBOL, CHAR END_SYMBOL, STD::MAP<CHAR, BOOL>
IS_VISITED_MAP, INT FLOW) {

    IF (START_SYMBOL == END_SYMBOL) RETURN FLOW;

    IS_VISITED_MAP[START_SYMBOL] = TRUE;

    STD::SORT(GRAPH_MAP_[START_SYMBOL].BEGIN(), GRAPH_MAP_[START_SYMBOL].END(),
NEIGHBOUR_CMP);

    FOR (AUTO& NEIGHBOUR : GRAPH_MAP_[START_SYMBOL]) {
        NEIGHBOUR.SECOND.IS_VISITED = FALSE;
        IF (!IS_VISITED_MAP[NEIGHBOUR.FIRST] && NEIGHBOUR.SECOND.FLOW > 0) {
            NEIGHBOUR.SECOND.IS_VISITED = TRUE;
            NEIGHBOUR* PTR;
            FOR(AUTO& REVERSE_NEIGHBOUR: GRAPH_MAP_[NEIGHBOUR.FIRST]) {
                REVERSE_NEIGHBOUR.SECOND.IS_VISITED = FALSE;
                IF (REVERSE_NEIGHBOUR.FIRST == START_SYMBOL &&
REVERSE_NEIGHBOUR.SECOND.INDEX == NEIGHBOUR.SECOND.INDEX) {
                    REVERSE_NEIGHBOUR.SECOND.IS_VISITED = TRUE;
                    PTR = &REVERSE_NEIGHBOUR;
                    BREAK;
                }
            }

            FLOW = NEIGHBOUR.SECOND.FLOW;
            PREV_NODE_MAP_[NEIGHBOUR.FIRST] = START_SYMBOL;

```

```

        INT FOUND_FLOW = FIND_FLOW_PATH (NEIGHBOUR.FIRST, END_SYMBOL,
IS_VISITED_MAP, FLOW);

        IF (FOUND_FLOW) {
            RETURN STD::MIN (FLOW, FOUND_FLOW);
        }

        NEIGHBOUR.SECOND.ISVISITED = FALSE;
        PTR->SECOND.ISVISITED = FALSE;
    }
}
RETURN 0;

}

INT GRAPH::FORD_FULKERSON() {

    PREV_NODE_MAP_[SOURCE_SYMBOL_] = SOURCE_SYMBOL_;

    STD::MAP<CHAR, BOOL> IS_VISITED_MAP;
    STD::VECTOR<CHAR> PATH;

    INT MAX_FLOW = 0;
    INT FOUND_FLOW = FIND_FLOW_PATH (SOURCE_SYMBOL_, SINK_SYMBOL_, IS_VISITED_MAP,
0);

    WHILE (FOUND_FLOW) {

        MAX_FLOW += FOUND_FLOW;
        PATH = GET_FLOW_PATH();

        FOR (AUTO NODE_CHAR = PATH.RBEGIN(); NODE_CHAR != PATH.REND(); +
+NODE_CHAR) {
            FOR (AUTO& NEIGHBOUR: GRAPH_MAP_[*NODE_CHAR]) {
                IF (NEIGHBOUR.FIRST == *(NODE_CHAR + 1) &&
NEIGHBOUR.SECOND.ISVISITED) {
                    NEIGHBOUR.SECOND.FLOW -= FOUND_FLOW;
                    NEIGHBOUR.SECOND.USEDFLOW += FOUND_FLOW;

                    FOR (AUTO& REVERSE_NEIGHBOUR : GRAPH_MAP_[NEIGHBOUR.FIRST])
{
                        IF (REVERSE_NEIGHBOUR.FIRST == *NODE_CHAR &&
REVERSE_NEIGHBOUR.SECOND.ISVISITED) {
                            REVERSE_NEIGHBOUR.SECOND.FLOW += FOUND_FLOW;
                            REVERSE_NEIGHBOUR.SECOND.USEDFLOW -= FOUND_FLOW;
                        }
                    }
                }
            }
        }

        FOUND_FLOW = FIND_FLOW_PATH (SOURCE_SYMBOL_, SINK_SYMBOL_,
IS_VISITED_MAP, 0);
    }

    RETURN MAX_FLOW;
}

```

```

STD::VECTOR<CHAR> GRAPH::GET_FLOW_PATH() {
    STD::VECTOR<CHAR> RESULT;
    CHAR CURRENT = SINK_SYMBOL_;
    RESULT.PUSH_BACK(CURRENT);
    WHILE (CURRENT != SOURCE_SYMBOL_) {
        CURRENT = PREV_NODE_MAP_[CURRENT];
        RESULT.PUSH_BACK(CURRENT);
    }

    RETURN RESULT;
}

STD::STRING GRAPH::PRINT_RES_GRAPH() {
    AUTO NEIGHBOUR_LETTER_CMP = [] (NEIGHBOUR A, NEIGHBOUR B) -> BOOL {
        RETURN A.FIRST < B.FIRST;
    };
    STD::STRING OUT;
    FOR (AUTO CHAR_VEC_PAIR = GRAPH_MAP_.BEGIN(); CHAR_VEC_PAIR !=
GRAPH_MAP_.END(); ++CHAR_VEC_PAIR) {
        STD::SORT(CHAR_VEC_PAIR->SECOND.BEGIN(), CHAR_VEC_PAIR->SECOND.END(),
NEIGHBOUR_LETTER_CMP);
        FOR(AUTO NEIGHBOUR = CHAR_VEC_PAIR->SECOND.BEGIN(); NEIGHBOUR !=
CHAR_VEC_PAIR->SECOND.END(); ++NEIGHBOUR) {
            IF(NEIGHBOUR->SECOND.ISINIT) {
                OUT += CHAR_VEC_PAIR->FIRST;
                OUT += " ";
                OUT += NEIGHBOUR->FIRST;
                OUT += " ";
                OUT += STD::TO_STRING(STD::MAX(0, NEIGHBOUR->
>SECOND.USEDFLOW));
                OUT += "\n";
            }
        }
    }
    RETURN OUT;
}

GRAPH::~~GRAPH() {
}

STD::VECTOR<CHAR> GRAPH::TEST_FIND_PATH(CHAR START_SYMBOL, CHAR END_SYMBOL) {
    STD::MAP<CHAR, BOOL> IS_VISITED_MAP;
    INT RES = FIND_FLOW_PATH(START_SYMBOL, END_SYMBOL, IS_VISITED_MAP, 0);
    RETURN GET_FLOW_PATH();
}

```

ФАЙЛ TEST_FIELD.CPP

```

#define CATCH_CONFIG_MAIN

#include "../CATCH.HPP"
#include "../SOURCE/GRAPH.HPP"

```

```

#include <sstream>

TEST_CASE("GRAPH CONSTRUCTOR AND READ INPUT TEST", "[INTERNAL GRAPH TEST]" ) {

    STD::stringstream INPUT_TEST;
    INT EDGE_AMOUNT = 3;
    CHAR SOURCE_SYMBOL = 'A';
    CHAR SINK_SYMBOL = 'C';
    INPUT_TEST << "A B 3\nB C 2\nA C 5";

    GRAPH GRAPH(SOURCE_SYMBOL, SINK_SYMBOL, EDGE_AMOUNT);
    GRAPH.READ_EDGES(INPUT_TEST);

    STD::STRING RES = GRAPH.TO_STRING();
    REQUIRE(RES == "A: B(3/0) C(5/0) \NB: A(0/3) C(2/0) \NC: B(0/2) A(0/5) \N");

    STD::stringstream INPUT_TEST_2;
    INPUT_TEST_2 << "A B 3\nB C 2\nA C -5";

    GRAPH GRAPH_2(SOURCE_SYMBOL, SINK_SYMBOL, EDGE_AMOUNT);

    REQUIRE(GRAPH_2.READ_EDGES(INPUT_TEST_2) == FALSE);

    STD::stringstream INPUT_TEST_3;
    CHAR SOURCE_SYMBOL_3 = '1';
    CHAR SINK_SYMBOL_3 = '3';
    INPUT_TEST_3 << "1 2 3\n2 3 2\n1 3 5";

    GRAPH GRAPH_3(SOURCE_SYMBOL_3, SINK_SYMBOL_3, EDGE_AMOUNT);
    GRAPH_3.READ_EDGES(INPUT_TEST_3);

    RES = GRAPH_3.TO_STRING();
    REQUIRE(RES == "1: 2(3/0) 3(5/0) \N2: 1(0/3) 3(2/0) \N3: 2(0/2) 1(0/5) \N");
}

TEST_CASE("COMPARATOR AND FIND PATH TEST", "[INTERNAL GRAPH TEST]" ) {

    NEIGHBOUR FIRST_NEIGHbour = {'A', {}};
    NEIGHBOUR SECOND_NEIGHbour = {'C', {}};

    REQUIRE(NEIGHBOUR_CMP(FIRST_NEIGHbour, SECOND_NEIGHbour) == FALSE);

    AUTO JOIN_PATH = [] (STD::VECTOR<CHAR> PATH) -> STD::STRING {
        STD::STRING OUT;
        FOR(CONST AUTO& EL: PATH) {
            OUT += EL;
        }
        RETURN OUT;
    };
};

```

```

STD::STRINGSTREAM INPUT_TEST;
INT EDGE_AMOUNT = 7;
CHAR SOURCE_SYMBOL = 'A';
CHAR SINK_SYMBOL = 'F';
INPUT_TEST << "A B 7\NA C 6\NB D 6\NC F 9\ND E 3\ND F 4\NE C 2";

GRAPH GRAPH(SOURCE_SYMBOL, SINK_SYMBOL, EDGE_AMOUNT);
GRAPH.READ_EDGES(INPUT_TEST);

STD::VECTOR<CHAR> PATH1 = GRAPH.TEST_FIND_PATH(SOURCE_SYMBOL, SINK_SYMBOL);

REQUIRE(JOIN_PATH(PATH1) == "FDBA");

STD::STRINGSTREAM INPUT_TEST_2;
INT EDGE_AMOUNT_2 = 5;
CHAR SOURCE_SYMBOL_2 = '1';
CHAR SINK_SYMBOL_2 = '4';
INPUT_TEST_2 << "1 2 20\N1 3 1\N2 3 20\N2 4 1\N3 4 20";

GRAPH GRAPH2(SOURCE_SYMBOL_2, SINK_SYMBOL_2, EDGE_AMOUNT_2);
GRAPH2.READ_EDGES(INPUT_TEST_2);

STD::VECTOR<CHAR> PATH2 = GRAPH2.TEST_FIND_PATH(SOURCE_SYMBOL_2,
SINK_SYMBOL_2);

REQUIRE(JOIN_PATH(PATH2) == "4321");
}

TEST_CASE("Max Flow AND Res Graph Test", "[INTERNAL GRAPH TEST]" ) {

STD::STRINGSTREAM INPUT_TEST;
INT EDGE_AMOUNT = 7;
CHAR SOURCE_SYMBOL = 'A';
CHAR SINK_SYMBOL = 'F';
INPUT_TEST << "A B 7\NA C 6\NB D 6\NC F 9\ND E 3\ND F 4\NE C 2";

GRAPH GRAPH(SOURCE_SYMBOL, SINK_SYMBOL, EDGE_AMOUNT);
GRAPH.READ_EDGES(INPUT_TEST);

INT MAX_FLOW = GRAPH.FORD_FULKERSON();

REQUIRE(MAX_FLOW == 12);
REQUIRE(GRAPH.PRINT_RES_GRAPH() == "A B 6\NA C 6\NB D 6\NC F 8\ND E 2\ND
F 4\NE C 2\N");

STD::STRINGSTREAM INPUT_TEST_2;
INT EDGE_AMOUNT_2 = 5;
CHAR SOURCE_SYMBOL_2 = '1';
CHAR SINK_SYMBOL_2 = '4';
INPUT_TEST_2 << "1 2 20\N1 3 1\N2 3 20\N2 4 1\N3 4 20";

GRAPH GRAPH_2(SOURCE_SYMBOL_2, SINK_SYMBOL_2, EDGE_AMOUNT_2);
GRAPH_2.READ_EDGES(INPUT_TEST_2);

```

```

    INT MAX_FLOW_2 = GRAPH_2.FORD_FULKERSON();

    REQUIRE(MAX_FLOW_2 == 21);
    REQUIRE(GRAPH_2.PRINT_RES_GRAPH() == "1 2 20\n1 3 1\n2 3 19\n2 4 1\n3
4 20\n");
}

```

ФАЙЛ MAKEFILE

```

FLAGS = -STD=C++17 -Wall -Wextra
BUILD = BUILD
SOURCE = SOURCE
TEST = TEST

$(SHELL MKDIR -p $(BUILD))

ALL: LAB3 RUN_TESTS

LAB3: $(BUILD)/GRAPH.o $(BUILD)/MAIN.o
    @ECHO "TO START ENTER ./LAB3"
    @G++ $(BUILD)/MAIN.o $(BUILD)/GRAPH.o -o LAB3 $(FLAGS)

RUN_TESTS: $(BUILD)/TEST.o $(BUILD)/GRAPH.o
    @ECHO "TO RUN TESTS ENTER ./RUN_TESTS"
    @G++ $(BUILD)/TEST.o $(BUILD)/GRAPH.o -o RUN_TESTS

$(BUILD)/MAIN.o: $(SOURCE)/MAIN.CPP
    @G++ -c $(SOURCE)/MAIN.CPP -o $(BUILD)/MAIN.o

$(BUILD)/GRAPH.o: $(SOURCE)/GRAPH.CPP $(SOURCE)/GRAPH.HPP
    @G++ -c $(SOURCE)/GRAPH.CPP -o $(BUILD)/GRAPH.o

$(BUILD)/TEST.o: $(TEST)/TEST.CPP
    @G++ -c $(TEST)/TEST.CPP -o $(BUILD)/TEST.o

CLEAN:
    @RM -rf $(BUILD)/
    @RM -rf *.o LAB3 RUN_TESTS

```