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* 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 представление 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

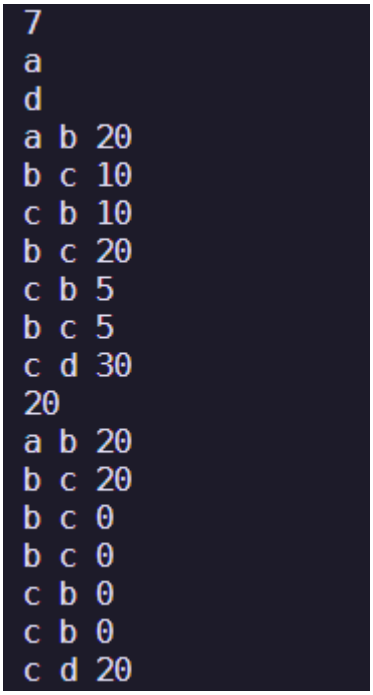

Рисунок 3 - Пример работы программы на входных данных №3

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. Данный способ поиска пути не самый оптимальный по времени — каждый раз при поиске пути приходится сортировать список соседних вершин. 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 usedFlow = 0;
    int index = 0;
    bool isInit = false;
    bool isVisited = 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.USEDFLOW);
            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.ISVISITED = FALSE;
            IF (!IS_VISITED_MAP[NEIGHBOUR.FIRST] && NEIGHBOUR.SECOND.FLOW > 0) {
              NEIGHBOUR.SECOND.ISVISITED = TRUE;
              NEIGHBOUR* PTR;
              FOR(AUTO& REVERSE_NEIGHBOUR: GRAPH_MAP_[NEIGHBOUR.FIRST]) {
                   REVERSE_NEIGHBOUR.SECOND.ISVISITED = FALSE;
                                IF(REVERSE_NEIGHBOUR.FIRST == START_SYMBOL &&
REVERSE_NEIGHBOUR.SECOND.INDEX == NEIGHBOUR.SECOND.INDEX) {
                        REVERSE_NEIGHBOUR.SECOND.ISVISITED = 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.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
```