

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**  
**Кафедра МО ЭВМ**

**ОТЧЕТ**  
**по лабораторной работе №2**  
**по дисциплине «Построение и анализ алгоритмов»**  
**Тема: Жадный алгоритм и  $A^*$**

Студент гр. 9383

\_\_\_\_\_

Гладких А.А.

Преподаватель

\_\_\_\_\_

Фирсов М.А.

Санкт-Петербург

2021

### **Цель работы.**

Применить на практике знания о построение жадного алгоритма поиска пути в графе и алгоритма  $A^*$  – «А звездочка». Реализовать программу, которая считывает граф и находит в нем путь от стартовой вершины к конечной с помощью жадного алгоритма и алгоритма  $A^*$ .

### **Задание.**

#### **Жадный алгоритм.**

Разработайте программу, которая решает задачу построения пути в ориентированном графе при помощи жадного алгоритма. Жадность в данном случае понимается следующим образом: на каждом шаге выбирается последняя посещённая вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешёвым из последней посещённой вершины. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес.

Пример входных данных:

*a e*

*a b 3.0*

*b c 1.0*

*c d 1.0*

*a d 5.0*

*d e 1.0*

В первой строке через пробел указываются начальная и конечная вершины.

Далее в каждой строке указываются ребра графа и их вес.

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет:

*abcde*

**A\*.**

Разработайте программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом A\*. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес. В качестве эвристической функции следует взять близость символов, обозначающих вершины графа, в таблице ASCII.

Пример входных данных:

*a e*

*a b 3.0*

*b c 1.0*

*c d 1.0*

*a d 5.0*

*d e 1.0*

В первой строке через пробел указываются начальная и конечная вершины.

Далее в каждой строке указываются ребра графа и их вес.

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет:

*ade*

**Вариант 5.** Реализация алгоритма, оптимального по используемой памяти (абсолютно, не только через O-нотацию).

**Ход работы:**

1. Произведён анализ задания.
2. Был реализован жадный алгоритм поиска пути в графе:
  1. Сперва все вершины помещаются в двоичную кучу. Изначально метки всех вершин равны максимально возможному значению — в нашем случае это максимальное значение типа float.

2. Стартовой вершине задается метка 0.
3. Затем начинается цикл, который прекратится тогда, когда в двоичной куче не останется элементов.
  1. Из кучи выбирается вершина с минимальной меткой (Сперва это стартовая вершина).
  2. Эта вершина удаляется из кучи и помечается как просмотренная.
  3. Если текущая вершина — конечная, то цикл завершается.
  4. Для всех вершин, соседствующих с текущей:
    1. Если вершина уже была посещена, то она пропускается.
    2. Если вершина еще не была посещена и ее метка больше, чем путь до нее через соединяющее эту вершину с текущей ребро, то ее метка обновляется.
4. В конце программа восстанавливает путь от конечной вершины до стартовой.
3. Был реализован поиск пути в графе с помощью алгоритма A\*. Алгоритм в целом повторяет жадный алгоритм, за исключением пункта 3.4 — даже, если вершина уже была посещена, алгоритм будет обновлять ее метку, если она больше, чем сумма пути и эвристической функции для данной вершины.
4. В ходе оптимизации алгоритмов по памяти были сделаны следующие корректировки в алгоритмах:
  1. Граф было принято считывать в структуру `vector` из стандартной библиотеки языка C++. Это было сделано для того, чтобы в дальнейшем сделать на основе этого динамического массива двоичную кучу с помощью функции `std::make_heap()`, чтобы не создавать ее дополнительно и копировать в нее заново вершины.
  2. Для создания кучи был сделан специальный компаратор `vertex_pair_cmp`, сравнивающий метки вершин.

3. Единственные издержки по памяти — дополнительный массив `utility_vector_`, в который копируются указатели на вершины. Это было сделано ради очистки памяти по итогу программы, формально к самим алгоритмам этот массив имеет лишь косвенное отношение.
5. В случае реализации алгоритмов на бинарной куче их сложность можно оценить как  $O(V \cdot \log V)$  во временном плане и  $O(V)$  в плане занимаемого места, где  $V$  — количество вершин.
6. Были написаны тесты с использованием библиотеки `Catch2` для тестирования корректности работы обоих алгоритмов. Были протестированы функции ввода данных и их корректная работа, а также работа самих алгоритмов.

```
followjust@DESKTOP-0H9PDI5:/mnt/c/Users/gladk/Desktop/Sync/UniSync/secondYear/second_sem/Algorithms/lab2$ ./run_tests
=====
All tests passed (11 assertions in 4 test cases)
```

Рисунок 1 - Иллюстрация результатов тестирования программы

7. Код разработанной программы расположен в Приложении А.

### **Описание функций и структур данных:**

1. Структура `Vertex` — представление вершины графа. Имеет поле `name` для хранения названия вершины, поле `edges_arr` — массив ребер графа, указатель на вершину, из которой в данную вершину пришли `prev`, поле типа `bool` для того, чтобы отмечать была посещена вершина или нет, `isVisited` и поле типа `float` для хранения метки вершины — `priority`.
2. Структура `Edge` — представление ребра вершины графа. Содержит два поля — вес ребра `weight` и указатель на вершину, в которую ребро приходит, `end_vertex`.
3. Класс `Graph` — представление графа. Имеет следующие поля — `start_vertex_symbol_` и `end_vertex_symbol` — названия стартовой и конечной вершины, `bool input_error_` - для проверки корректности ввода, `vertices_arr_` -

массив вершин и его копия `utility_vector_`, указатель на стартовую и конечную вершину `start_vertex_` и `end_vertex_` соответственно, а также строка `path_` для записи полученного пути.

Также класс имеет следующие методы — `find_path_a_star()` и `find_path_greedy()` для поиска пути алгоритмом A\* и жадным алгоритмом, метод для считывания ребер из потока — `read_edges()`, метод вывода считанного массива вершин в виде строки `print_vector_to_string()`, метод `check_input_error()` для проверки корректности ввода, метод `heuristics()` для получения значения эвристической функции для двух вершин и метод поиска вершины в векторе вершин `get_vertex_from_vector()`.

4. Функция `main()` - функция, в которой происходит инициализация графа и запуск алгоритмов.

## Примеры работы программы.

Таблица 1 – Пример работы программы

№ п/п	Входные данные	Выходные данные
1.	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	Greedy Path: abcde
2.	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	A Star Path: ade
3.	a z a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	A Star Path: no path

## Иллюстрация работы программы.

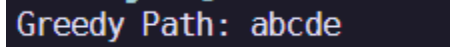
A dark blue rectangular box with the text "Greedy Path: abcde" in a light blue, monospaced font.

Рисунок 2 - Пример работы программы на входных данных №1

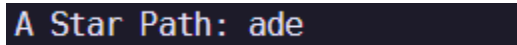
A dark blue rectangular box with the text "A Star Path: ade" in a light blue, monospaced font.

Рисунок 3 - Пример работы программы на входных данных №2

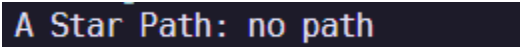
A dark blue rectangular box with the text "A Star Path: no path" in a light blue, monospaced font.

Рисунок 4 - Пример работы программы на входных данных №3



## **Выводы.**

Были применены на практике знания о построение жадного алгоритма поиска пути и алгоритма  $A^*$  – «А звездочка». Была реализована программа, которая считывает граф и находит в нем путь от стартовой вершины к конечной с помощью жадного алгоритма и алгоритма  $A^*$ . Была выполнена оптимизация алгоритмов по используемой памяти — для этого было минимизировано копирование структур, хранящих вершины графа: бинарная куча строится на основе массива, в который изначально происходит чтение информации о вершинах.

## ПРИЛОЖЕНИЕ А

### ФАЙЛ MAIN.CPP

```
#INCLUDE "GRAPH.HPP"

INT MAIN() {

    CHAR START_VERTEX, END_VERTEX;
    STD::CIN >> START_VERTEX >> END_VERTEX;

    GRAPH GRAPH(START_VERTEX, END_VERTEX, STD::CIN);
    //STD::COUT << GRAPH.PRINT_VECTOR_TO_STRING() << "\n";
    STD::COUT << "A STAR PATH: " << GRAPH.FIND_PATH_A_STAR() << '\n';
    //STD::COUT << "GREEDY PATH: " << GRAPH.FIND_PATH_GREEDY() << '\n';

    RETURN 0;
}
```

### ФАЙЛ GRAPH.HPP

```
#INCLUDE <Iostream>
#include <vector>
#include <queue>
#include <map>
#include <string>
#include <algorithm>

STRUCT EDGE;

STRUCT VERTEX{
    CHAR NAME;
    STD::VECTOR<EDGE> EDGES_ARR;
    VERTEX* PREV;
    BOOL ISVISITED = FALSE;
    FLOAT PRIORITY = __FLT_MAX__;
};

STRUCT EDGE{
    VERTEX* END_VERTEX;
    FLOAT WEIGHT;
};

CLASS GRAPH{
PUBLIC:

    GRAPH(CHAR START_VERTEX, CHAR END_VERTEX, STD::Istream& IN);
    ~GRAPH();

    STD::STRING FIND_PATH_A_STAR();
    STD::STRING FIND_PATH_GREEDY();

    VOID READ_EDGES(STD::Istream& IN);
```

```

    STD::STRING PRINT_VECTOR_TO_STRING();

    BOOL CHECK_INPUT_ERROR() { RETURN INPUT_ERROR_; };

    FLOAT HEURISTICS(CONST CHAR& START, CONST CHAR& FINISH);

PRIVATE:
    CHAR START_VERTEX_SYMBOL_, END_VERTEX_SYMBOL_;

    BOOL INPUT_ERROR_;

    STD::VECTOR<VERTEX*> VERTICES_ARR_;
    STD::VECTOR<VERTEX*> UTILITY_VECTOR_;

    VERTEX* START_VERTEX_;
    VERTEX* END_VERTEX_;

    STD::STRING PATH_;

    VERTEX* GET_VERTEX_FROM_VECTOR(CHAR TARGET_VERTEX_NAME);

};

```

## ФАЙЛ GRAPH.HPP

```

#include "GRAPH.HPP"

GRAPH::GRAPH(CHAR START_VERTEX, CHAR END_VERTEX, STD::ISTREAM& IN) {
    START_VERTEX_SYMBOL_ = START_VERTEX;
    END_VERTEX_SYMBOL_ = END_VERTEX;

    INPUT_ERROR_ = FALSE;

    READ_EDGES(IN);
}

STD::STRING GRAPH::FIND_PATH_A_STAR() {
    IF (VERTICES_ARR_.SIZE() < 2 || INPUT_ERROR_) RETURN "Error";

    AUTO VERTEX_PAIR_CMP = [] (VERTEX* LEFT, VERTEX* RIGHT) -> BOOL {
        IF (LEFT->PRIORITY == RIGHT->PRIORITY && LEFT->NAME < RIGHT->NAME)
            RETURN TRUE;
        RETURN LEFT->PRIORITY > RIGHT->PRIORITY;
    };

    START_VERTEX_ = GET_VERTEX_FROM_VECTOR(START_VERTEX_SYMBOL_);
    START_VERTEX_>PRIORITY = 0 + HEURISTICS(START_VERTEX_SYMBOL_,
END_VERTEX_SYMBOL_);

    END_VERTEX_ = GET_VERTEX_FROM_VECTOR(END_VERTEX_SYMBOL_);

```

```

STD::MAKE_HEAP(VERTICES_ARR_.BEGIN(), VERTICES_ARR_.END(), VERTEX_PAIR_CMP);

VERTEX* CUR_VERTEX;

WHILE(!VERTICES_ARR_.EMPTY()) {
    CUR_VERTEX = VERTICES_ARR_.FRONT();
    IF (CUR_VERTEX->NAME == END_VERTEX_SYMBOL_) {
        WHILE (CUR_VERTEX->PREV) {
            PATH_ += CUR_VERTEX->NAME;
            CUR_VERTEX = CUR_VERTEX->PREV;
        }
        PATH_ += CUR_VERTEX->NAME;
        BREAK;
    }

    FOR (AUTO& EDGE : CUR_VERTEX->EDGES_ARR) {
        FLOAT TEMP = CUR_VERTEX->PRIORITY - HEURISTICS(CUR_VERTEX->NAME,
END_VERTEX_SYMBOL_) + EDGE.WEIGHT;
        IF (EDGE.END_VERTEX->PRIORITY > TEMP) {

            EDGE.END_VERTEX->PRIORITY = TEMP + HEURISTICS(EDGE.END_VERTEX-
>NAME, END_VERTEX_SYMBOL_);
            EDGE.END_VERTEX->PREV = CUR_VERTEX;
        }
    }
    VERTICES_ARR_.ERASE(VERTICES_ARR_.BEGIN());

    STD::MAKE_HEAP(VERTICES_ARR_.BEGIN(), VERTICES_ARR_.END(),
VERTEX_PAIR_CMP);
}

REVERSE(PATH_.BEGIN(), PATH_.END());

IF (VERTICES_ARR_.EMPTY() || PATH_[0] != START_VERTEX_SYMBOL_)
    RETURN "NO PATH";

RETURN PATH_;
}

STD::STRING GRAPH::FIND_PATH_GREEDY() {
    IF (VERTICES_ARR_.SIZE() < 2 || INPUT_ERROR_) RETURN "Error";

    AUTO VERTEX_PAIR_CMP = [] (VERTEX* LEFT, VERTEX* RIGHT) -> BOOL {
        RETURN LEFT->PRIORITY > RIGHT->PRIORITY;
    };

    START_VERTEX_ = GET_VERTEX_FROM_VECTOR(START_VERTEX_SYMBOL_);
    START_VERTEX_->PRIORITY = 0;

    STD::MAKE_HEAP(VERTICES_ARR_.BEGIN(), VERTICES_ARR_.END(), VERTEX_PAIR_CMP);

    VERTEX* CUR_VERTEX;

```

```

WHILE (!VERTICES_ARR_.EMPTY()) {

    CUR_VERTEX = VERTICES_ARR_.FRONT();

    CUR_VERTEX->ISVISITED = TRUE;

    IF (CUR_VERTEX->NAME == END_VERTEX_SYMBOL_) {
        WHILE (CUR_VERTEX->PREV) {
            PATH_ += CUR_VERTEX->NAME;
            CUR_VERTEX = CUR_VERTEX->PREV;
        }
        PATH_ += CUR_VERTEX->NAME;
        BREAK;
    }

    FOR (AUTO& EDGE : CUR_VERTEX->EDGES_ARR) {
        IF (EDGE.END_VERTEX->PRIORITY > EDGE.WEIGHT && !EDGE.END_VERTEX-
>ISVISITED) {
            EDGE.END_VERTEX->PRIORITY = EDGE.WEIGHT;
            EDGE.END_VERTEX->PREV = CUR_VERTEX;
        }
    }

    VERTICES_ARR_.ERASE(VERTICES_ARR_.BEGIN());

    STD::MAKE_HEAP(VERTICES_ARR_.BEGIN(), VERTICES_ARR_.END(),
VERTEX_PAIR_CMP);
}

REVERSE(PATH_.BEGIN(), PATH_.END());

IF (VERTICES_ARR_.EMPTY() || PATH_[0] != START_VERTEX_SYMBOL_)
    RETURN "NO PATH";

RETURN PATH_;
}

VOID GRAPH::READ_EDGES(STD::ISTREAM& IN) {
    CHAR EDGE_START_NAME, EDGE_END_NAME;
    FLOAT EDGE_WEIGHT;

    VERTEX* FIRST_VERTEX;
    VERTEX* SECOND_VERTEX;

    WHILE (IN >> EDGE_START_NAME >> EDGE_END_NAME >> EDGE_WEIGHT) {

        IF (EDGE_WEIGHT < 0.0 || !ISALPHA(EDGE_START_NAME) || !
ISALPHA(EDGE_END_NAME)) {
            INPUT_ERROR_ = TRUE;
            RETURN;
        }

        FIRST_VERTEX = GET_VERTEX_FROM_VECTOR(EDGE_START_NAME);
        IF (!FIRST_VERTEX) {

```

```

        FIRST_VERTEX = NEW VERTEX ({EDGE_START_NAME});
        VERTICES_ARR_.PUSH_BACK (FIRST_VERTEX);
    }

    SECOND_VERTEX = GET_VERTEX_FROM_VECTOR (EDGE_END_NAME);
    IF (!SECOND_VERTEX) {
        SECOND_VERTEX = NEW VERTEX ({EDGE_END_NAME});
        VERTICES_ARR_.PUSH_BACK (SECOND_VERTEX);
    }

    FIRST_VERTEX->EDGES_ARR.PUSH_BACK ({SECOND_VERTEX, EDGE_WEIGHT});

    IF (IN.EOF()) {
        BREAK;
    }

}

IF (VERTICES_ARR_.SIZE() == 0) {
    INPUT_ERROR_ = TRUE;
}

UTILITY_VECTOR_ = VERTICES_ARR_;
}

STD::STRING GRAPH::PRINT_VECTOR_TO_STRING() {
    STD::STRING OUT = "";
    FOR (CONST AUTO& CUR_VERTEX : VERTICES_ARR_) {
        OUT += CUR_VERTEX->NAME;
        OUT += " = [ ";
        FOR (CONST AUTO& NEIGHBOUR : CUR_VERTEX->EDGES_ARR) {
            OUT += NEIGHBOUR.END_VERTEX->NAME;
            OUT += " ";
        }
        OUT += "]; ";
    }
    RETURN OUT;
}

GRAPH::~~GRAPH() {
    FOR (AUTO& VERTEX : UTILITY_VECTOR_) {
        DELETE VERTEX;
    }
}

VERTEX* GRAPH::GET_VERTEX_FROM_VECTOR (CHAR TARGET_VERTEX_NAME) {
    FOR (AUTO& CUR_VERTEX : VERTICES_ARR_) {
        IF (CUR_VERTEX->NAME == TARGET_VERTEX_NAME) {
            RETURN CUR_VERTEX;
        }
    }

    RETURN nullptr;
}

```

```

}

FLOAT GRAPH::HEURISTICS(CONST CHAR& START, CONST CHAR& FINISH) {
    RETURN ABS(START - FINISH);
}

```

## 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]" ) {

    CHAR FROM_VERTEX = 'A';
    CHAR TO_VERTEX = 'E';

    STD::STRINGSTREAM INPUT_TEST;
    INPUT_TEST << "A B 3.0\NB C 1.0\NC D 1.0\NA D 5.0\ND E 1.0";

    GRAPH GRAPH(FROM_VERTEX, TO_VERTEX, INPUT_TEST);

    STD::STRING RES = GRAPH.PRINT_VECTOR_TO_STRING();
    REQUIRE(RES == "A = [ B D ]; B = [ C ]; C = [ D ]; D = [ E ]; E = [ ];
");

    STD::STRINGSTREAM INPUT_TEST2;
    INPUT_TEST2 << "A B 3.0\NB C 1.0\NC D -1.0\NA D 5.0\ND E 1.0";

    GRAPH GRAPH2(FROM_VERTEX, TO_VERTEX, INPUT_TEST2);

    REQUIRE(GRAPH2.CHECK_INPUT_ERROR() == TRUE);

    STD::STRINGSTREAM INPUT_TEST3;
    INPUT_TEST3 << "; B 3.0\NB C 1.0\NC D 1.0\NA D 5.0\ND E 1.0";

    GRAPH GRAPH3(FROM_VERTEX, TO_VERTEX, INPUT_TEST3);

    REQUIRE(GRAPH3.CHECK_INPUT_ERROR() == TRUE);

    STD::STRINGSTREAM INPUT_TEST4;
    INPUT_TEST4 << "";

    GRAPH GRAPH4(FROM_VERTEX, TO_VERTEX, INPUT_TEST4);

    REQUIRE(GRAPH4.CHECK_INPUT_ERROR() == TRUE);

}

```

```

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

    CHAR FROM_VERTEX = 'A';
    CHAR TO_VERTEX = 'E';

    STD::STRINGSTREAM INPUT_TEST;
    INPUT_TEST << "A B 3.0\NB C 1.0\NC D 1.0\NA D 5.0\ND E 1.0";

    GRAPH GRAPH(FROM_VERTEX, TO_VERTEX, INPUT_TEST);

    STD::STRING RES = GRAPH.FIND_PATH_GREEDY();
    REQUIRE(RES == "ABCDE");

    STD::STRINGSTREAM INPUT_TEST2;
    INPUT_TEST2 << "A B 3.0\NB C 1.0";

    GRAPH GRAPH2(FROM_VERTEX, TO_VERTEX, INPUT_TEST2);

    RES = GRAPH2.FIND_PATH_GREEDY();
    REQUIRE(RES == "NO PATH");
}

TEST_CASE("HEURISTIC TEST", "[INTERNAL GRAPH TEST]" ) {

    CHAR FROM_VERTEX = 'A';
    CHAR TO_VERTEX = 'E';

    STD::STRINGSTREAM INPUT_TEST;
    INPUT_TEST << "A B 3.0\NB C 1.0\NC D 1.0\NA D 5.0\ND E 1.0";

    GRAPH GRAPH(FROM_VERTEX, TO_VERTEX, INPUT_TEST);

    REQUIRE(GRAPH.HEURISTICS('A', 'E') == 4);

    REQUIRE(GRAPH.HEURISTICS('B', 'C') == 1);
}

TEST_CASE("FIND PATH A* TEST", "[INTERNAL GRAPH TEST]" ) {

    CHAR FROM_VERTEX = 'A';
    CHAR TO_VERTEX = 'E';

    STD::STRINGSTREAM INPUT_TEST;
    INPUT_TEST << "A B 3.0\NB C 1.0\NC D 1.0\NA D 5.0\ND E 1.0";

    GRAPH GRAPH(FROM_VERTEX, TO_VERTEX, INPUT_TEST);

    STD::STRING RES = GRAPH.FIND_PATH_A_STAR();
    REQUIRE(RES == "ADE");

    STD::STRINGSTREAM INPUT_TEST2;
    INPUT_TEST2 << "A B 3.0\NB C 1.0\NC D 1.0\NA D 5.0\ND E 1.0\NA E 1.0";

    GRAPH GRAPH2(FROM_VERTEX, TO_VERTEX, INPUT_TEST2);

```



```

RES = GRAPH2.FIND_PATH_A_STAR();
REQUIRE (RES == "AE");

STD::stringstream INPUT_TEST3;
INPUT_TEST3 << "A B 3.0\nB C 1.0";

GRAPH GRAPH3 (FROM_VERTEX, TO_VERTEX, INPUT_TEST3);

RES = GRAPH3.FIND_PATH_A_STAR();
REQUIRE (RES == "NO PATH");
}

```

## ФАЙЛ MAKEFILE

```

FLAGS = -std=c++17 -Wall -Wextra
BUILD = BUILD
SOURCE = SOURCE
TEST = TEST

$(SHELL) mkdir -p $(BUILD)

ALL: LAB2 RUN_TESTS

LAB2: $(BUILD)/MAIN.O $(BUILD)/GRAPH.O
    @ECHO "TO START ENTER ./LAB2"
    @g++ $(BUILD)/MAIN.O $(BUILD)/GRAPH.O -o LAB2 $(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
    @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 LAB2

```