

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*

Студентка гр. 9383

Лапина 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.

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 3.

Написать функцию, проверяющую эвристику на допустимость и монотонность.

Основные теоретические положения.

Алгоритм A^* (англ. A star) — алгоритм поиска, который находит во взвешенном графе маршрут наименьшей стоимости от начальной вершины до выбранной конечной.

Описание:

В процессе работы алгоритма для вершин рассчитывается функция $f(v)=g(v)+h(v)$, где

$g(v)$ — наименьшая стоимость пути в v из стартовой вершины,

$h(v)$ — эвристическое приближение стоимости пути от v до конечной цели.

Фактически, функция $f(v)$ — длина пути до цели, которая складывается из пройденного расстояния $g(v)$ и оставшегося расстояния $h(v)$. Исходя из этого, чем меньше значение $f(v)$, тем раньше мы откроем вершину v , так как через неё мы предположительно достигнем расстояние до цели быстрее всего. Открытые алгоритмом вершины можно хранить в очереди с приоритетом по значению $f(v)$. A^* действует подобно алгоритму Дейкстры и просматривает среди всех маршрутов ведущих к цели сначала те, которые благодаря имеющейся информации (эвристическая функция) в данный момент являются наилучшими.

Свойства:

Чтобы A^* был оптимален, выбранная функция $h(v)$ должна быть допустимой эвристической функцией.

Говорят, что эвристическая оценка $h(v)$ **допустима**, если для любой вершины v значение $h(v)$ меньше или равно весу кратчайшего пути от v до цели.

Допустимая оценка является оптимистичной, потому что она предполагает, что стоимость решения меньше, чем оно есть на самом деле.

Второе, более сильное условие — функция $h(v)$ должна быть монотонной.

Эвристическая функция $h(v)$ называется **монотонной** (или преобладающей), если для любой вершины v_1 и ее потомка v_2 разность $h(v_1)$ и

$h(v_2)$ не превышает фактического веса ребра $c(v_1, v_2)$ от v_1 до v_2 , а эвристическая оценка целевого состояния равна нулю.

Выполнение работы:

В программе реализован класс `rebro` для хранения информации о ребре:

С переменными:

`char v` — куда идет ребро;

`float metrika` — вес (метрика) ребра;

С методами:

`char name()` - для получения `v`;

`float len()` - для получения `metrika`;

В программе были использованы следующие функции:

* `int h(char name, char finish)` — для вычисления эвристической функции;

* `void Check_h(std::vector <char> v, char finish, int path, Graph G)` -
определяет является ли функция допустимой и монотонной;

* `void Read(Graph &G, std::vector <char> &v_for_check)`; — считывает
ребра и заполняет `Graph`;

* `std::string Algoritm_A(float &priority, char start, char finish, Graph G)` —
функция для выполнения алгоритма A^* , возвращает результат в виде строки;

Описание алгоритма работы программы:

Программа получает на вход 2 вершины `start` и `finish`, затем ребра -
<исходящая вершина> <входящая вершина> <вес> до окончания ввода (`ctrl+D`),
считывание происходит в функции `Read()` в `Graph` — словарь, где `first` —
исходящая вершина, а `second` — вектор из структур (на рисунке 1 изображена
структура для входных данных из задания лабораторной). Затем с помощью
функции `Algoritm_A` ищем наименьший путь в графе: пока начальная вершина

не равна конечной составляем очередь приоритетов, выбираем нужные вершины, записываем в *past*, а затем пробегаемся и собираем путь с конца, записываем результат в строку и переворачиваем. С помощью функции *Check_h* исследуем эвристику на допустимость (проверяем, что для любой вершины *v* значение $h(v)$ меньше или равно весу кратчайшего пути от *v* до цели.) и монотонность (проверяем, что для любой вершины *v1* и ее потомка *v2* разность $h(v1)$ и $h(v2)$ не превышает фактического веса ребра $c(v1,v2)$ от *v1* до *v2*, а эвристическая оценка целевого состояния равна нулю).

first	second	
a	v = b; metrika = 3.0	v = d; metrika = 5.0
b	v = c; metrika = 1.0	
c	v = d; metrika = 1.0	
d	v = e; metrika = 1.0	

Рисунок 1 — изображение словаря Graph для примера из задания

Тестирование

1) Входные данные:

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

Выходные данные:

Проверка эвристики на допустимость:

Вершина a удовлетворяет условию допустимости

Вершина e удовлетворяет условию допустимости

Вершина b удовлетворяет условию допустимости

Вершина c удовлетворяет условию допустимости

Вершина d удовлетворяет условию допустимости

Эвристическая функция допустима

Проверка эвристики на монотонность

Ребро ab не нарушает монотонность

Ребро ad не нарушает монотонность

Ребро bc не нарушает монотонность

Ребро cd не нарушает монотонность

Ребро de не нарушает монотонность

Эвристическая функция монотонна

2) Входные данные:

a z

a w 17

a b 1

b w 1

w x 1

x y 18

y z 1

Выходные данные:

abwxyz

Проверка эвристики на допустимость:

Вершина a не удовлетворяет условию допустимости

Вершина z удовлетворяет условию допустимости

Вершина w удовлетворяет условию допустимости

Вершина b не удовлетворяет условию допустимости

Вершина x удовлетворяет условию допустимости

Вершина u удовлетворяет условию допустимости
Эвристическая функция не допустима

Проверка эвристики на монотонность

Ребро aw нарушает монотонность

Ребро ab не нарушает монотонность

Ребро bw нарушает монотонность

Ребро wx не нарушает монотонность

Ребро xu не нарушает монотонность

Ребро yz не нарушает монотонность

Эвристическая функция не монотонна

3) Входные данные:

$b \in$

$b \in 1$

$b \in 5$

$c \in 100$

$d \in 10$

Выходные данные:

bde

Проверка эвристики на допустимость:

Вершина b удовлетворяет условию допустимости

Вершина e удовлетворяет условию допустимости

Вершина c удовлетворяет условию допустимости

Вершина d удовлетворяет условию допустимости

Эвристическая функция допустима

Проверка эвристики на монотонность

Ребро bc не нарушает монотонность

Ребро bd не нарушает монотонность

Ребро ce не нарушает монотонность

Ребро de не нарушает монотонность

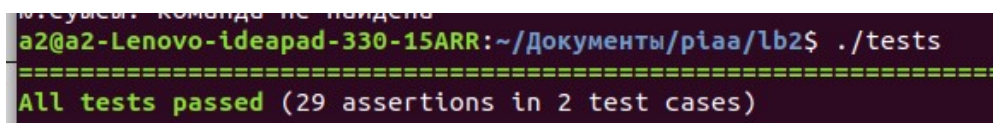
Эвристическая функция монотонна

Также были реализованы тесты:

1) Для проверки работы эвристической функции h — рассмотрен средний случай работы функции, случай, когда $start=finish$ и случай, если ребра заданы не типом `char`, а типом `float` или `int`.

2) Для проверки функции `Algoritm_A`, смотрим меняется ли переменная `priority`, а также правильный ли выводится результат

Результаты работы написанных тестов предоставлены на рисунке 2



```
а2@a2-Lenovo-ideapad-330-15ARR:~/Документы/piaa/lb2$ ./tests
=====
All tests passed (29 assertions in 2 test cases)
```

Рисунок 2 — Результаты тестов программы

Разработанный программный код см. в приложении А.

Вывод.

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

Приложение А

Исходный код программы

Название файла: main.cpp

```
#include "lb2.hpp"

int main(){
    char start, finish;
    std::cin>>start;
    std::cin>>finish;
    Graph G;
    std::vector <char> v_for_check;
    v_for_check.push_back(start);
    v_for_check.push_back(finish);
    Read(G, v_for_check);
    float priority = 0;    //путь до предыдущей вершины
    std::string result = Algoritm_A(priority, start, finish, G);
    std::cout << result << '\n';
    Check_h(v_for_check, finish, priority, G);

    return 0;
}
```

Название файла: lb2.cpp

```
#include "lb2.hpp"

int h(char name, char finish) {    //эврист ф-ция
    return abs(finish - name);
};

void Read(Graph &G, std::vector <char> &v_for_check){
    char source , dest;
    float metr;
    while (std::cin >> source >> dest >> metr){
        rebro g(dest, metr);
        //проверяем есть ли уже вектор с такой вершиной
        if (G.count(source) == 0) {
            std::vector<rebro> d = {g};
            G[source] = d;
        } else {
            G[source].push_back(g);
        }
        auto is_not_find =std::find(v_for_check.begin(),
v_for_check.end(), dest);
        if (is_not_find==v_for_check.end())
            v_for_check.push_back(dest);
    }
}

void Check_h(std::vector <char> v, char finish, int path, Graph G){
    std::cout << "Проверка эвристики на допустимость:\n";
    bool check_dop = true;
    for (int i = 0; i!=v.size();i++){
        if(h(v[i], finish)<=path){
```

```

        std::cout << "\tВершина " <<v[i] << " удовлетворяет
условию допустимости\n";
    }
    else {
        std::cout << "\tВершина " <<v[i] << " не удовлетворяет
условию допустимости\n";
        check_dop = false;
    }
}
if (check_dop)
    std::cout << "Эвристическая функция допустима\n";
else
    std::cout << "Эвристическая функция не допустима\n";

std::cout << "\nПроверка эвристики на монотонность\n";
bool check_mon = true;
for (auto it = G.begin(); it!=G.end();it++){
    auto source = it->first;
    for (auto i = it->second.begin();i!=it->second.end(); i++){
        if (abs(h(source, finish) - h(i->name(), finish)) <= i-
>len()){
            std::cout << "\tРебро " << source << i->name() << "
не нарушает монотонность\n";
        }
        else{
            std::cout << "\tРебро " << source << i->name() << "
нарушает монотонность\n";
            check_mon = false;
        }
    }
}
if (check_mon && ( h(finish, finish) == 0))
    std::cout << "Эвристическая функция монотонна\n";
else
    std::cout << "Эвристическая функция не монотонна\n";
}

std::string Algoritm_A(float &priority, char start, char finish,
Graph G){

    std::vector<std::tuple<char, char, float>> past; //вектор
кортежей пройденных ребер (float = путь до вершины)
    std::vector<std::tuple<char, char, float>> queue; //в.к.
очереди с приоритетом (float = путь до вершины+эвристика)
    char cur = start;
    while (cur != finish){
        for(auto it = G[cur].begin(); it != G[cur].end(); it++){
            queue.push_back(std::make_tuple(cur, it->name(),
priority + it->len() + h(it->name(), finish)));
        }
        std::sort(queue.begin(), queue.end(), [] (const
std::tuple<char, char, float> &p1, const std::tuple<char, char, float>
&p2) { //по убыванию, берем с конца наименьший
            if (std::get<2>(p1) > std::get<2>(p2)){
                return true;
            }
            return false;
        });
    }
}

```

```

bool check = false;
char prev;
while(!queue.empty()){
    prev = std::get<0>(queue.back());    //предыдущий
    cur = std::get<1>(queue.back());    //нынешний
    priority = std::get<2>(queue.back()) - h(cur, finish);
//путь

    queue.pop_back();
    int flag = 0;
    for(auto it = past.begin(); it != past.end(); it++){
        if (std::get<1>(*it) == cur){
            if(std::get<2>(*it) <= priority){
                flag = -1;
                break;
            }else{
                past.erase(it);
                break;
            }
        }
    }
    if (flag == -1){ //если есть путь короче, то берет
следующий
        continue;
    }
    check = true;
    break;
}
if(!check){
    std::cout << "Пути нет!\n";
    return 0;
}
past.push_back(std::make_tuple(prev, cur, priority));
}
std::string result = "";
cur = finish;
result.push_back(cur);
while(cur != start){
    for (auto it = past.begin(); it != past.end(); it++){

        if(std::get<1>(*it) == cur){ //для поиска результата
            cur = std::get<0>(*it);
            break;
        }
    }
    result.push_back(cur);
}

std::reverse(result.begin(), result.end()); //перевернули
return result;
}

```

Название файла: lb2.hpp

```

#include <iostream>
#include <map>
#include <tuple>

```

```

#include <vector>
#include <algorithm>

class rebro {
public:
    rebro(char v, float metrika){
        this->v = v;
        this->metrika = metrika;
    }

    char name()const{
        return this->v;
    }

    float len()const{
        return this->metrika;
    }

private:
    char v;
    float metrika;
    bool view;
};

using Graph = std::map<char, std::vector<rebro>>;

int h(char name, char finish);
void Read(Graph &G, std::vector<char> &v_for_check);
void Check_h(std::vector<char> v, char finish, int path, Graph G);
std::string Algoritm_A(float &priority, char start, char finish,
Graph G);

```

Название файла: tests_programm.cpp

```

#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include "../source/lb2.hpp"
TEST_CASE("Тест для эвристической функции h") {
    SECTION("Сравниваем, когда finish == finish") {
        REQUIRE(h('a', 'a') == 0);
        REQUIRE(h('n', 'n') == 0);
        REQUIRE(h('w', 'w') == 0);
        REQUIRE(h('z', 'z') == 0);
    }

    SECTION("Средний случай при вводе finish>start") {
        REQUIRE(h('a', 'z') == 25);
        REQUIRE(h('b', 'z') == 24);
        REQUIRE(h('n', 'r') == 4);
        REQUIRE(h('c', 'd') == 1);
        REQUIRE(h('f', 'i') == 3);
        REQUIRE(h('x', 'z') == 2);
        REQUIRE(h('d', 'y') == 21);
        REQUIRE(h('d', 'w') == 19);
    }
}

```

```

        SECTION("При вводе start>finish (не должно ломаться, тк
используем модуль)") {
            REQUIRE(h('z', 'a') == 25);
            REQUIRE(h('z', 'b') == 24);
            REQUIRE(h('r', 'n') == 4);
            REQUIRE(h('d', 'c') == 1);
        }

        SECTION("Если вершины не типа char, а типа int/float") {
            REQUIRE(h(98, 122) == 24);
            REQUIRE(h(100, 118) == 18);
            REQUIRE(h(100.8, 122.5) == 22);
            REQUIRE(h(99.1, 100.2) == 1);
        }
    }

    TEST_CASE("Тест для функции Algoritm_A") {

        SECTION("1 случай") {
            float priority = 0;
            char start = 'a';
            char finish = 'e';
            Graph G = {{'a', {{'b', 3.0}, {'d', 5.0}}}, {'b', {{'v',
1.0}}}, {'c', {{'d', 1.0}}}, {'d', {{'e', 1.0}}}};

            std::string result = Algoritm_A(priority, start, finish, G);
            CHECK(priority!=0);
            CHECK(priority==6);
            CHECK(result=="ade");
        }

        SECTION("2 случай") {
            float priority = 0;
            char start = 'a';
            char finish = 'z';
            Graph G = {{'a', {{'w', 17}, {'b', 1}}}, {'b', {{'w', 1}}},
{'w', {{'x', 1}}}, {'x', {{'y', 18}}}, {'y', {{'z', 1}}}};

            std::string result = Algoritm_A(priority, start, finish, G);
            CHECK(priority!=0);
            CHECK(priority==22);
            CHECK(result=="abwxyz");
        }

        SECTION("3 случай") {
            float priority = 0;
            char start = 'b';
            char finish = 'e';
            Graph G = {{'b', {{'c', 1}, {'d', 5}}}, {'c', {{'e', 100}}},
{'d', {{'e', 10}}}};

            std::string result = Algoritm_A(priority, start, finish, G);
            CHECK(priority!=0);
            CHECK(priority==15);
            CHECK(result=="bde");
        }
    }
}

```