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: Алгоритм A*

Студентка гр. 7383 _____ Маркова А. В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Содержание

Цель работы

Исследовать и реализовывать задачу построения кратчайшего пути в ориентированном графе, используя алгоритм A*.

Формулировка задачи: необходимо разработать программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом A* до любой из представленных вершин. Каждая вершина в графе имеет буквенное обозначение («a», «b», «c» …), каждое ребро имеет неотрицательный вес.

Вариант 2м: граф представлен в виде матрицы смежности, эвристическая функция для каждой вершины задаётся положительным числом во входных данных.

Входные данные: в первой строке указываются начальная и конечная вершины. Затем количество вершин и их эвристические значения. Далее идут данные о рёбрах графа и их весе.

Выходные данные: кратчайший путь из стартовой вершины в конечную.

Реализация задачи

В данной работе используются главная функция `main()` и структуры данных.

Был использован следующий класс:

```
class A_star_algorithm {
private:
    double graph[Alphabet][Alphabet];
    std:: vector <int> result;
    std:: vector <int> str;
public:
    void insert_paths(int i, int j, double
path_length);
    void print();
    void algorithm(int first, int finish, std::
priority_queue <Priority> &queue, double common_way, std::
vector <double> &evristic);
};
```

Параметры, хранящиеся в классе:

- `graph` – двумерный массив, в котором хранится граф в виде матрицы смежности;
- `result` – строка результата;
- `str` – строка для хранения пути до текущей вершины.

Методы класса:

- `insert_paths` – функция заполнения матрицы весами рёбер;
- `print` – функция вывода результата на консоль;
- `algorithm` – функция поиска кратчайшего пути методом A*.

Параметры, передаваемые в `void insert_paths(int i, int j, double path_length)`:

- `i` – индекс строки матрицы, вершина из которой идём;
- `j` – индекс столбца матрицы, вершина в которую идём;
- `path_length` – вес ребра между `i` и `j` вершинами.

Параметры, передаваемые в `void algorithm(int first, int finish, std:: priority_queue <Priority> &queue, double common_way, std:: vector <double> &evristic)`:

- `first` – текущая вершина графа;
- `finish` – конечная вершина;
- `queue` – приоритетная очередь;
- `common_way` – общая длина пути до текущей вершины;
- `evristic` – вектор, хранящий значения эвристики для каждой из вершин.

Класс `A_star_algorithm` отвечает за хранение графа в виде матрицы смежности, а так же содержит функцию `algorithm`, которая осуществляет поиск кратчайшего пути из начальной вершины в конечную, используя A*.

Структура `Priority`:

```
typedef struct Priority {
    std:: vector <int> path;
    double prior_vertex;
    double characteristic;
} Priority;
```

Параметры структуры:

- `path` – путь до данной вершины;
- `prior_vertex` – значение эвристической функции, по которому строится приоритетная очередь;
- `characteristic` – эвристика вершины.

Данная структура используется для хранения данных в очереди с приоритетами.

Параметры, передаваемые в `bool operator < (const Priority &comp_var_1, const Priority &comp_var_2)` являются элементами структуры. Данная функция используется для того, чтобы в приоритетной очереди сначала шли элементы с меньшим значением эвристической функции.

В функции `main()` считываются начальная и конечная вершины, между которыми нужно найти кратчайший путь. Затем записываются в массив значения эвристики для каждой из вершин графа. Далее в цикле начинается считывание рёбер графа и их вес. Запускается алгоритм A*, который рассматривает всех соседей текущей вершины, помещает пути до них в очередь, в зависимости от значения эвристической функции, и находит кратчайший путь для вершины, которая на данном шаге находится ближе. Пока очередь не пуста алгоритм выполняет свою работу.

Рассмотрим пример работы программы. На рис. 1 показано в каком виде хранятся данные о графе.

Characteristic (эвристика вершин графа)

a	b	c	d	e	f	g	. . .	z
0	3	1	7	2	0	8	0	0

Graph (матрица смежности)

I \ j	a	b	c	d	e	f	g	. . .	z
a	0	0	0	0	0	0	0	0	0
b	0	0	3.0	0	5.0	0	0	0	0
c	0	0	0	1.0	0	0	0	0	0
d	0	0	0	0	1.0	0	0	0	0
e	0	0	0	0	0	0	1.0	0	0
f	0	0	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0	0
. . .	0	0	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0	0	0

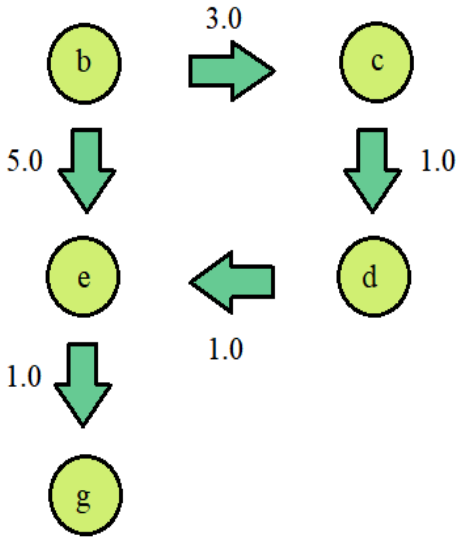


Рисунок 1 – представление графа матрицей смежности

Программа считывает значения, хранящиеся в текстовом файле, и заполняет поля класса. На первом шаге алгоритма рассматриваются две соседние вершины от стартовой b: c и e, подробнее представлено на рис. 2

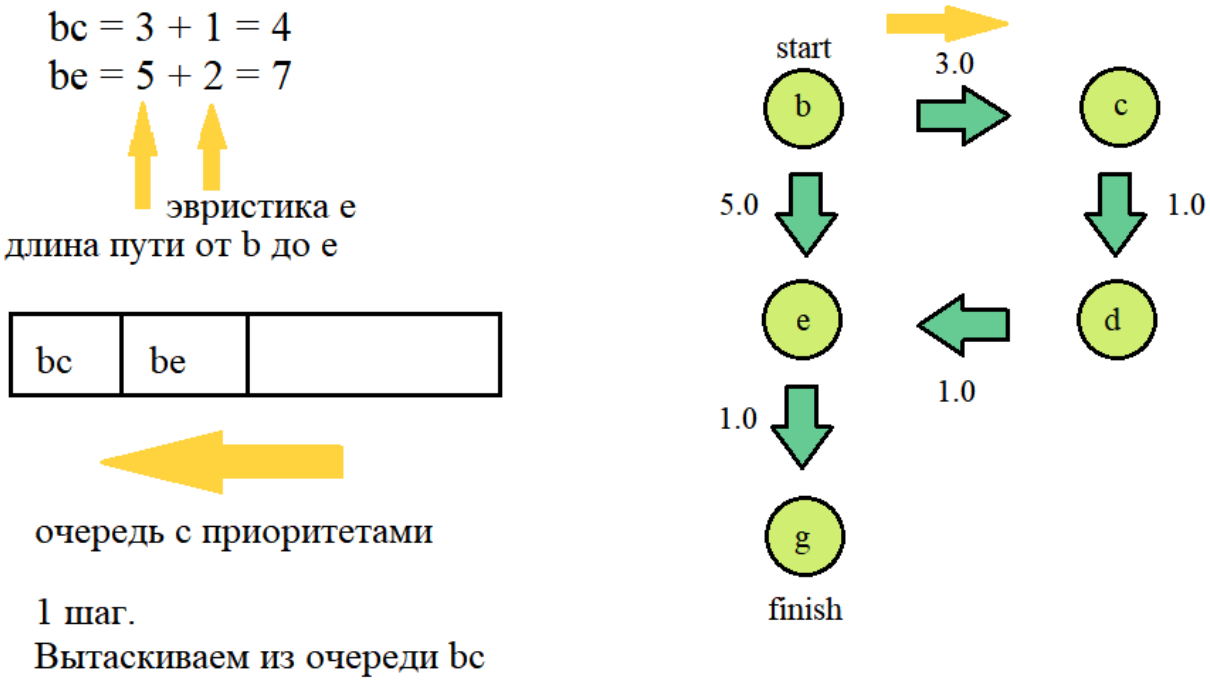


Рисунок 2 – первый шаг алгоритма.

Следующие шаги алгоритма показаны на рис. 3, 4.

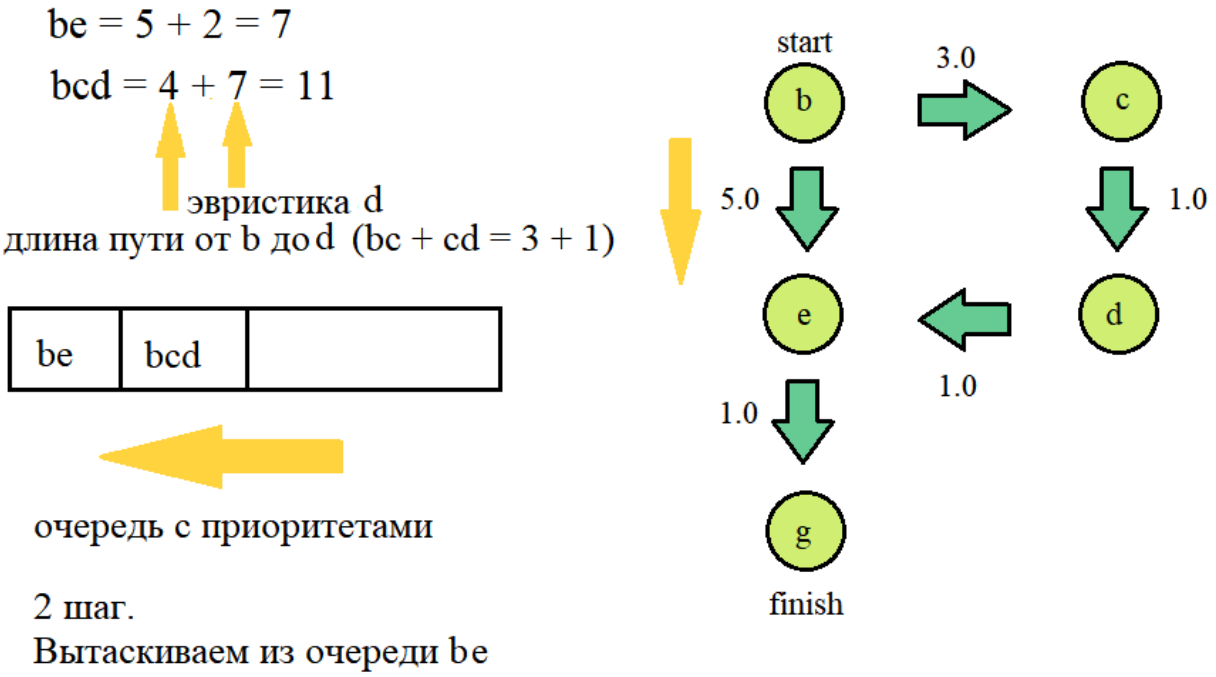
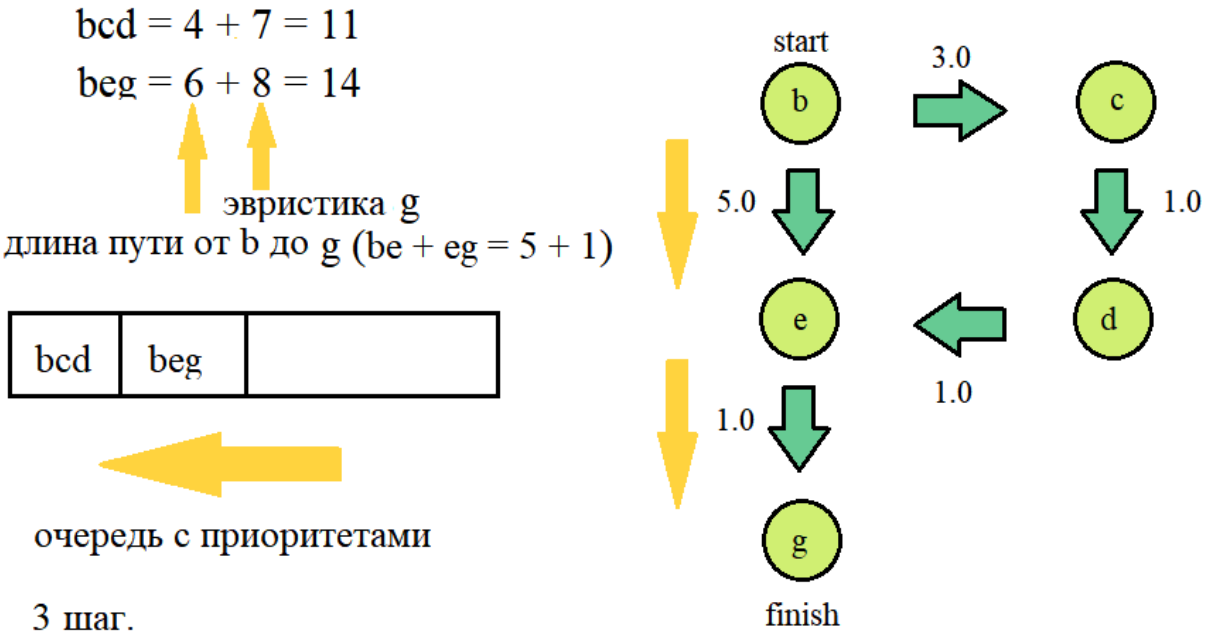


Рисунок 3 – второй шаг алгоритма.



3 шаг.

Вытаскиваем из очереди bcd, но так как дошли до финиша алгоритм завершает работу

Рисунок 4 – третий шаг алгоритма.

После того как алгоритм завершает работу, заполняется строка `result`, а затем выводится на экран в качестве ответа.

Исходный код программы представлен в приложении Б.

Тестирование

Программа собрана в операционной системе Ubuntu 17.04, с использованием компилятора g++ версии 5.4.0 20160609. В других ОС и компиляторах тестирование не проводилось.

Программа может быть скомпилирована с помощью команды:

g++ <имя файла>.cpp

Тестовые случаи представлены в Приложении А.

Исходя из тестовых случаев можно заметить, что тестовые случаи не выявили неправильного поведения программы, что говорит о том, что по результатам тестирование было показано, поставленная задача была выполнена.

Исследование

Сложность алгоритма составляет O(|V|•|E|), где |V| – количество вершин в графе, |E| – количество ребер в графе. На каждом шаге работы программы просматриваются всевозможные пути из искомой вершины. В худшем случае могут быть просмотрены все пути данного графа. Тогда сложность зависит от количества вершин.

Выводы

В ходе лабораторной работы был изучен алгоритм поиска кратчайшего пути A*. Был написан код на языке программирования C++, который применял этот метод для поставленной задачи. Сложность реализованного алгоритма составляет O(|V|•|E|).

ПРИЛОЖЕНИЕ А

Тестовые случаи

Ввод	Вывод	Верно?
a e 5 a 3 b 1 c 9 d 1 e 4 a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade	Да
b g 5 b 3.0 c 1.0 d 7.0 e 2.0 g 8.0 b c 3.0 c d 1.0 d e 1.0 b e 5.0 e g 1.0	beg	Да

ПРИЛОЖЕНИЕ Б

Код программы

```
#include <iostream>
#include <cstring>
#include <climits>
#include <queue>

#define Alphabet 26

typedef struct Priority {
   std:: vector <int> path;      //путь до текущей вершины
   double prior_vertex;          //приоритет – значение
эврестической функции
   double characteristic;
} Priority;

bool operator < (const Priority &comp_var_1, const Priority
&comp_var_2) {  //элементы с меньшим приоритетом будут идти сначала
   return comp_var_1.prior_vertex > comp_var_2.prior_vertex;
}


class A_star_algorithm {
private:
   double graph[Alphabet][Alphabet]; //матрица смежности
   std:: vector <int> result;        //результат работы алгоритма
   std:: vector <int> str;           //временная строка
public:
   A_star_algorithm(int start) {     //инициализация
       result.push_back(start);
        for(int i = 0; i < Alphabet; i++)
        memset(graph[i], 0.0, Alphabet * sizeof(double));
   }

   ~A_star_algorithm() {
       result.clear();
       str.clear();
   }

   void insert_paths(int i, int j, double path_length) {
```

```
        graph[i][j] = path_length;
    }

    void print(){ //вывод результата
        if(result.size()) {
            for(int i = 0; i < result.size(); i++)
                std:: cout <<(char)(result[i] + 'a');
            std:: cout << std:: endl;
        }
        else std:: cout << "Пути не существует" << std:: endl;
    }

void algorithm(int first, int finish, std:: priority_queue
<Priority> &queue, double common_way, std:: vector <double>
&evristic) {
        int run = 1;
        while(run) {
            for(int j = 0; j < Alphabet; j++)
                if(graph[first][j] != 0){
                    Priority new_elem;
                    new_elem.characteristic = evristic[j];
                    new_elem.prior_vertex = graph[first][j] +
common_way + new_elem.characteristic;
                    for(int i = 0; i < str.size(); i++)
                        new_elem.path.push_back(str[i]);
                    new_elem.path.push_back(j);
                    queue.push(new_elem);
                }
            if(queue.empty())
                run = 0;
            if(!queue.empty()) {
                Priority temp;
                temp = queue.top();
                queue.pop();
                first = temp.path[temp.path.size()-1];
                str = temp.path;
                common_way = temp.prior_vertex -
temp.characteristic;
            }
            if(str[str.size() - 1] == finish){
                for(int i = 0; i < str.size(); i++)
                    result.push_back(str[i]);
                run = 0;
```

```
            }
        }
    }
};

int main() {
   char start, finish, from, to, curr;
   double path_length, evristic;
   int count = 0;
   std:: vector <double> characteristic(Alphabet, 0);
   std:: cin >> start >> finish;
   std:: cin >> count;
   for(int i = 0; i < count; i++) {
        std:: cin >> curr >> evristic;
        if(evristic >= 0.0)
             characteristic[curr - 'a'] = evristic;
        else {
             std:: cout << "Введены неверные значения!" << std::
endl;
             return 0;
        }
   }
   std:: priority_queue <Priority> queue;
   A_star_algorithm graph(start - 'a');
   while(std:: cin >> from >> to >> path_length)
        graph.insert_paths(from - 'a', to - 'a', path_length);
   graph.algorithm(start - 'a', finish - 'a', queue, 0.0,
characteristic);
   graph.print();
   characteristic.clear();
   return 0;
}
```