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

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритмы на графах

Студент гр. 7383 _____ Медведев И. С.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2019

Содержание

Цель работы

Ознакомиться с алгоритмом поиска кратчайшего пути A*, написать программу, реализующую этот алгоритм.

Реализация задачи

Для решения поставленной задачи были использованы `struct Node`, в которой хранятся поля char name (имя вершины), `Node* parent` (указатель на ближайшую вершину, которая указывает на данную), bool is_visited (показывает посещена ли вершина), `std::vector <double> weights` (веса ребер в смежные вершины), `std::vector <Node*> nodes` (указатели на смежные вершины), `int h` (число, которым задается эвристическая функция), `int key` (переменная, показывающая приоритет вершины). Так же были написаны функции:

`void set_keys (Node* tmp, std::vector <Node*>& str)` – функция, которая на вход принимает указатель на вершину и вектор указателей. Задает для смежных нерассмотренных вершин приоритет нахождения в очереди, и записывает их в вектор `str`.

`std::string Astar (Node* start, Node* finish)` – функция, реализующая алгоритм A*. Пока рассматриваемая вершина не будет равна конечной запускаем функцию `set_keys`, сортируем строку по убыванию приоритетов, в которую записали указатели вершин, делаем последний элемент отсортированного вектора текущей вершиной, и удаляем ее из строки. Когда встретили конечную вершину, вставляем в строку `ans` имя вершины и переходим к ближайшей смежной вершине. Возвращаем строку с ответом.

`void create_list(std::vector <Node*>& list, char n1,char n2, double weight)` – функция, принимает на вход вектор, куда записываются все вершины, имена первой вершины (откуда исходит ребро) и второй (куда ребро входит) и вес ребра. Функция ищет вершины в `list` у

которых имена совпадают с переданными, если не находит, то создает новые вершины и добавляет их в `list`. Если находит, то добавляет в поля `nodes` и `weights` первой вершины новые записи.

В головной функции мы считываем имена вершин и веса, создаем список указателей на эти вершины с помощью функции `create_list`. Затем считываем эвристические функции для каждой вершины и запускаем функцию `Astar`. Т.к. ответ записывается с последней вершины, то с помощью `std::reverse` разворачиваем строку и выводим ее на экран.

Исходный код программы представлен в Приложении А.

Тестирование

Программа собрана в операционной системе Ubuntu 17.04 с использованием компилятора g++. В других ОС и компиляторах тестирование не проводилось. Результаты тестирования показали, что поставленная цель выполнена. Результаты тестирования представлены в Приложении Б.

Сложность алгоритма

Функция `set_keys` задает для каждой вершины свой приоритет, т.к. приоритет требуется задать для каждой не просмотренной смежной для данной вершины, то сложность будет O(|V|+|E|), где |V| – количество вершин в графе, |E| – количество ребер. Функция `Astar` в худшем случае пройдет по всем вершинам, поэтому ей понадобится O(|V|) итераций. Следовательно, сложность алгоритма можно оценить, как $O(|E|^2+|V||E|)$.

Память в данном алгоритме выделяется под вершины графа, поэтому по памяти сложность алгоритма можно оценить, как O(|V|).

Вывод

В ходе выполнения лабораторной работы был изучен алгоритм A*, была написана программа реализующая этот алгоритм и примерно оценена сложность алгоритма.

ПРИЛОЖЕНИЕ А.

КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include <queue>
struct Node{
    char name;
    Node* parent;
    bool is_visited;
    std::vector <double> weights;
    std::vector <Node*> nodes;
    int h;
    int key;
};

bool cmp(const Node* a, const Node* b) {return a->key > b->key;}


void create_list(std::vector <Node*>& list, char n1, char n2, double weight){
        int it1 = -1, it2 = -1, i = 0;

        for (auto nd : list)
            {
                if (nd->name == n1)
                    it1 = i;
                if (nd->name == n2)
                    it2 = i;
                i++;
            }

        if (it1 == -1){
            Node* first = new Node;
            first->name = n1;
            first->is_visited = false;
            first->key = 0;
            first->parent = nullptr;
            list.push_back(first);
            it1 = list.size() - 1;

        }

        if (it2 == -1){
            Node* second = new Node;
            second->name = n2;
            second->is_visited = false;
            second->key = 0;
            second->parent = nullptr;
            list.push_back(second);
```

```
            it2 = list.size() - 1;
        }

        list[it1]->nodes.push_back(list[it2]);
        list[it1]->weights.push_back(weight);

}



void set_keys(Node* tmp, std::vector <Node*>& str){
    std::vector <Node*>:: iterator it;
    double path = tmp->key - tmp->h;
    size_t i = 0;
    for (; i < tmp->nodes.size(); i++){
        if(tmp->nodes[i]->is_visited == false){
            if(tmp->nodes[i]->key == 0 || (tmp->nodes[i]->h + path + tmp->weights[i]
< tmp->nodes[i]->key)){
                tmp->nodes[i]->key = tmp->nodes[i]->h + path + tmp->weights[i];
                tmp->nodes[i]->parent = tmp;
                it = std::find (str.begin(), str.end(), tmp->nodes[i]);
                if( it == str.end()){
                    str.push_back(tmp->nodes[i]);
                }

            }
        }
    }
}


std::string Astar(Node* start, Node* finish){
    std::vector <Node*> str;
    std::string ans;
    Node* tmp = start;
    tmp->key = tmp->h;
    while (true){
        if(tmp->name == finish->name)
            break;
        else{
            tmp->is_visited = true;
            set_keys(tmp, str);
            sort(str.begin(), str.end(), cmp);
            tmp = str[str.size() - 1];
            str.pop_back();
        }
    }
    while(tmp){
        ans.push_back(tmp->name);
        tmp = tmp->parent;
    }
    delete tmp;
    return ans;
```

```
}



int main()
{
    Node node;
    std::vector <Node*> list;
    std::string ans;
    char start, finish, n1, n2;
    double weight;
    int i = 0, it1 = 0, it2 = 0, heruistic = 0;
    std::cin >> start >> finish;
    while (std::cin >> n1 ){
        if(n1 == '!')
            break;
        std::cin >> n2 >> weight;
        create_list(list, n1, n2, weight);
    }
    for (auto nd : list){
        std::cout << "Enter the heruistic for "<<nd->name<<std::endl;
        std::cin.clear();
        std::cin >> heruistic;
        while (heruistic < 0){
            std::cout<<"Enter the right heruistic for " << n1 <<std::endl;
            std::cin>>heruistic;
        }
        nd->h = heruistic;
        if (nd->name == start)
            it1 = i;
        if (nd->name == finish)
            it2 = i;
        i++;
    }

    ans = Astar(list[it1], list[it2]);
    std::reverse(ans.begin(), ans.end());
    std::cout<<ans<<std::endl;
    return 0;

}
```

ПРИЛОЖЕНИЕ Б.

ТЕСТОВЫЕ СЛУЧАИ

Результаты тестирования представлены на рис. 1-3.

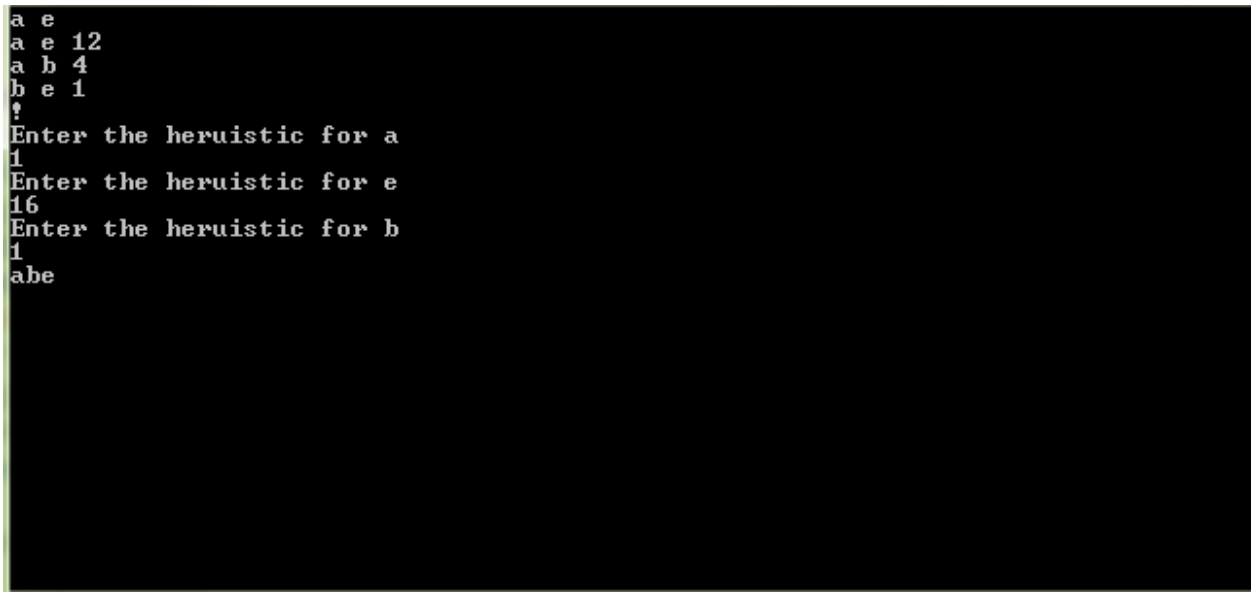

Рисунок 1 – Тест с корректными данными

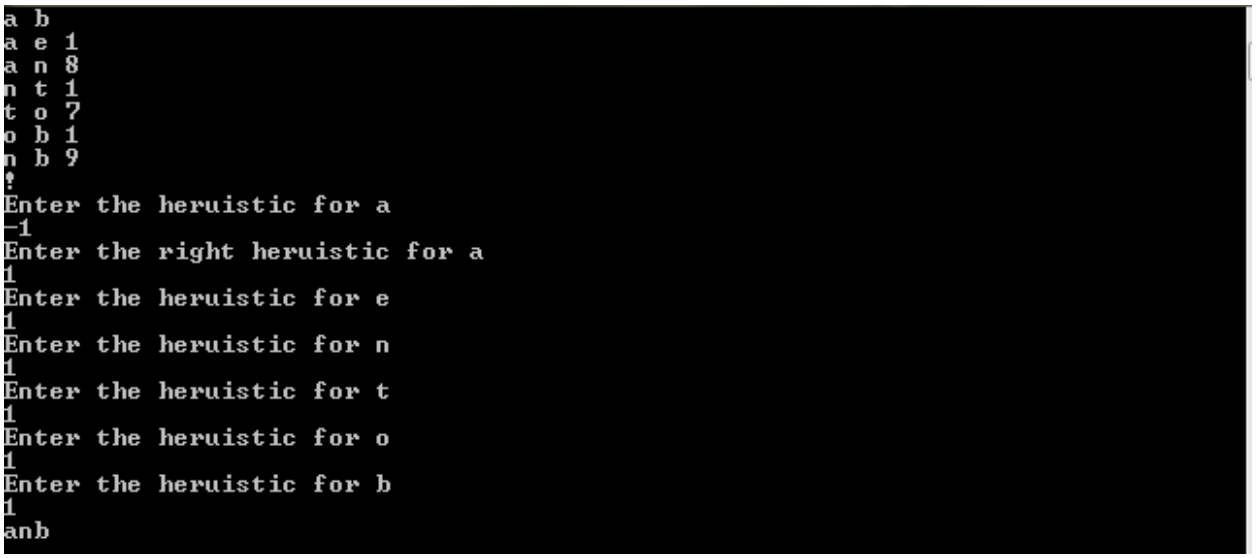

Рисунок 2 – Попытка ввода отрицательного числа

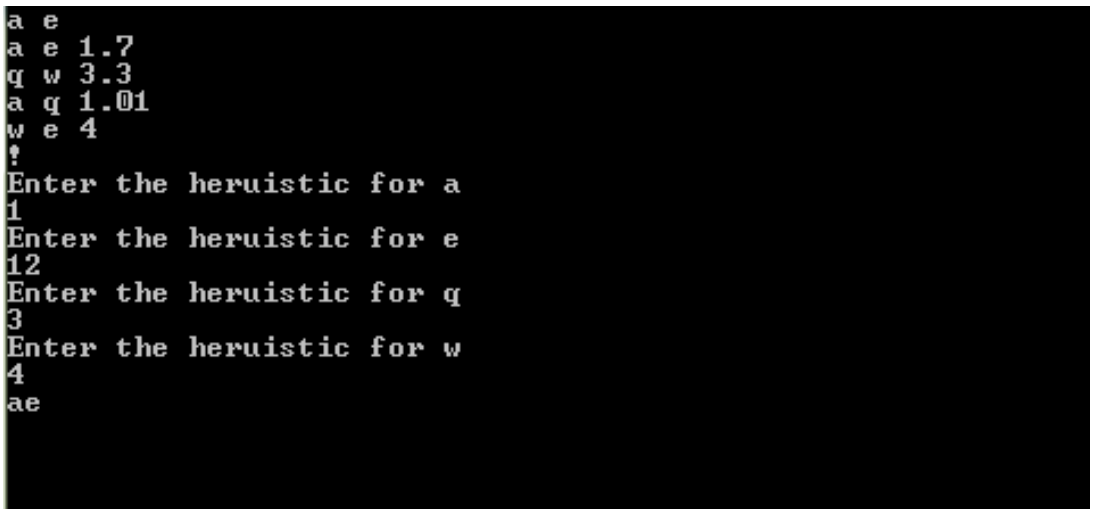

Рисунок 3 – Ввод действительных весов.