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 алгоримов»

Тема: Жадный алгоритм и A*

Студент гр. 8304 _____ Матросов Д.В.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2019

Цель работы.

Изучить и реализовать на языке программирования C++ жадный алгоритм поиска пути в графе и алгоритм A* поиска кратчайшего пути в графе между двумя заданными вершинами.

Задание.

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.

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.

Вар. 8: Перед выполнением A* выполнять предобработку графа: для каждой вершины отсортировать список смежных вершин по приоритету.

Ход выполнения работы:

1. Описание СД.

Для жадного алгорима были разработаны следующие СД:

- struct Rib – ребро графа, представленное в виде двух вершин и веса ребра.
- class Graph – сам граф, класс имеет методы заполнения графа, проферки доступности вершины, печати и поиска пути.

Для алгоритма A* были разработынны следующие СД:

- struct Rib – ребро графа, представленное в виде двух вершин и веса ребра.
- class Graph – сам граф, класс имеет методы заполнения графа, проферки доступности вершины, печати и поиска пути.
- struct Step – проделанный алгоритмом путь.

2. Описание функций и методов:

Для жадного алгорима были разработаны следующие методы:

- void input() – заполнение графа
- bool is_visible(char value) – проверка нфершины на доступность.
- bool Search(char value) – метод поиска пути.
- void Print() – метод для печати результата.

Для алгоритма A* былы также разработа эвристическая функция min_elem.

Выводы.

В ходе выполнения данной лабораторной работы были изучены и реализованы два алгоритма. Первый – жадный алгоритм поиска пути в ориентированном графе. Второй – алгоритм поиска минимального пути в ориентированном графе A*, который является модификацией алгоритма Дейкстры.

```
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <cfloat>
#include <algorithm>
#include <fstream>

using namespace std;

struct Rib
{
    char begin;
    char end;
    double weight;

    friend std::istream& operator>> (std::istream& in,
Rib& point);
};


class Graph
{
private:
    vector <Rib> graph;
    vector <char> res;
    vector <char> path;
    char src;
    char dst;

public:

    Graph(){}

    void input();

    bool is_visible(char value);

    bool Search(char value);

    void init_search();

    void Print();
};

void Graph::input() {
    cin >> src >> dst;
    Rib r;
    while (cin >> r)
```

```
    {
        graph.push_back(r);
    }
}

bool Graph::is_visible(char value)
{
    for (char i : path)
        if (i == value)
            return true;
    return false;
}

bool Graph::Search(char value)
{
    if (src != dst) {
        if (value == dst)
        {
            res.push_back(value);
            return true;
        }

        path.push_back(value);

        for (auto& i : graph)
        {
            if (value == i.begin)
            {
                if (is_visible(i.end))
                    continue;
                res.push_back(i.begin);
                bool flag = Search(i.end);
                if (flag)
                    return true;
                res.pop_back();
            }
        }
    }
    return false;
}

void Graph::init_search() {
    Search(src);
}

void Graph::Print()
{
    for (auto i : res)
        cout << i;
}
```

```
std::istream& operator>> (std::istream& in, Rib& r)
{
    in >> r.begin;
    in >> r.end;
    in >> r.weight;
    return in;
}

int main()
{
    Graph g;
    g.input();
    g.init_search();
    g.Print();
    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <cfloat>
#include <algorithm>
#include <fstream>

using namespace std;

struct Rib
{
    char begin;
    char end;
    double weight;

    friend std::istream& operator>> (std::istream& in,
Rib& point);
};

std::istream& operator>> (std::istream& in, Rib& r)
{
    in >> r.begin;
    in >> r.end;
    in >> r.weight;
    return in;
}

struct Step
{
    string path;
    double length;
    char estuary;
};

class Graph {
private:
    vector<Rib> graph;
    vector<Step> res;
    vector<char> curr;
    char source;
    char estuary;

public:
    Graph() {};

    void input() {
        char tmp;
        Rib elem;
```

```
        cin >> source >> estuary;
        while (cin >> elem) {
            graph.push_back(elem);
        }
        string buf = "";
        buf += source;
        for (auto& i : graph) {
            if (i.begin == source) {
                buf += i.end;
                res.push_back({ buf, i.weight });
                res.back().estuary = estuary;
                buf.resize(1);
            }
        }
        curr.push_back(source);
    }

    size_t min_elem()
    {
        double min;
        min = DBL_MAX;
        size_t temp = -1;
        for (size_t i(0); i < res.size(); i++) {
            if (res.at(i).length + abs(estuary -
res.at(i).path.back()) < min) {
                if (is_visible(res.at(i).path.back())) {
                    res.erase(res.begin() + i);
                }
                else {
                    min = res.at(i).length + abs(estuary
- res.at(i).path.back());
                    temp = i;
                }
            }
        }
        return temp;
    }

    bool is_visible(char value)
    {
        for (char i : curr) {
            if (i == value) {
                return true;
            }
        }
        return false;
    }

    void Search() {
        sort(res.begin(), res.end(), [](const Step& a,
const Step& b) -> bool {
```

```
            return a.length + a.estuary - a.path.back() >
b.length + b.estuary - b.path.back();
            });
        while (true) {
            size_t min = min_elem();
            if (min == -1) {
                break;
            }
            if (res.at(min).path.back() == estuary) {
                cout << res.at(min).path;
                return;
            }
            for (auto& i : graph) {
                if (i.begin == res.at(min).path.back()) {
                    string buf = res.at(min).path;
                    buf += i.end;
                    res.push_back({ buf, i.weight +
res.at(min).length });
                }
            }
            curr.push_back(res.at(min).path.back());
            res.erase(res.begin() + min);
        }
    }
};



int main(int argc, char* argv[])
{
    Graph element;
    element.input();
    element.Search();
    return 0;
}
```