

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

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

Студентка гр.7304

\_\_\_\_\_

Каляева А.В.

Преподаватель

\_\_\_\_\_

Филатов А.Ю

г. Санкт-Петербург

г. 2019

## **Цель работы:**

Изучить жадный алгоритм и алгоритм  $A^*$ , а так же реализовать данные алгоритмы на языке программирования C++.

## **Задание:**

- 1) Разработать 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.
- 2) Разработать 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.

## **Ход работы:**

- 1) Была написана программа, которая реализует поиск пути в ориентированном взвешенном графе с помощью жадного алгоритма. Программа работает следующим образом. При вводе данных для каждой пары вершин вызывается функция, которая инициализирует граф в виде списка смежности. После инициализации графа вызывается рекурсивная функция, которая реализует жадный поиск. В начале функции выполняется проверка на конечную вершину. В случае, если конечная вершина была достигнута, происходит выход из рекурсивной функции. Иначе вызывается функция для поиска соседа с наименьшим весом ребра, после чего рекурсивная функция вызывается заново с новой стартовой вершиной. Поиск пути с помощью жадного алгоритма не гарантирует нахождения кратчайшего пути в графе.
- 2) Была написана программа, которая реализует поиск кратчайшего пути в графе с помощью алгоритма  $A^*$ . Программа работает следующим образом. При вводе данных для каждой пары вершин вызывается функция, которая инициализирует граф в виде списка смежности. Так же в функции для создания графа вычисляется эвристическое значение, которое обозначает близость символов, обозначающих вершины графа, в таблице ASCII. После создания графа вызывается функция, реализующая алгоритм  $A^*$ . В теле этой функции выполняется проверка на достижение конечной вершины. Если конечная вершина еще не достигнута, то вызывается функция, которая вычисляет приоритет достижения соседних вершин и формирует вектор. После формирования вектора приоритетных вершин, выполняется сортировка данного вектора с помощью функции стандартной библиотеки `sort`, в порядке убывания приоритета достижения вершины. Далее в функции

Astar выбирается самая приоритетная вершина, и поиск выполняется заново с текущей вершины. В конце функции, когда путь был найден, он записывается в вектор, который хранит путь.

### **Заключение:**

В ходе выполнения данной лабораторной работы были изучены и реализованы на языке программирования C++ алгоритмы поиска пути в графе такие как жадный алгоритм и A\*.

## Приложение А

### Исходный код программы lr2\_1.cpp

```
#include <iostream>
#include <vector>

#define MAX 1000000000.0

using namespace std;

class Vertex {
public:
    char name_of_vertex;
    bool viewing;
    vector<Vertex*> neighbour;
    vector<double> weight;
    Vertex(char name):name_of_vertex(name), viewing(false){}
};

void creation_of_graph(char from, char to, double weight, vector<Vertex*>& graph){
    int index1 = -1;
    int index2 = -1;
    for (size_t i = 0; i < graph.size(); i++) {
        if (from==graph[i]->name_of_vertex) {
            index1 = i;
        }
        if (to == graph[i]->name_of_vertex) {
            index2 = i;
        }
    }
    if (index1 == -1) {
        Vertex *tmp1= new Vertex(from);
        graph.push_back(tmp1);
        index1 = graph.size() - 1;
    }
    if (index2 == -1) {
        Vertex *tmp2=new Vertex(to);
        graph.push_back(tmp2);
        index2 = graph.size() - 1;
    }
    graph[index1]->neighbour.push_back(graph[index2]);
    graph[index1]->weight.push_back(weight);
}

void print_graph(vector<Vertex*> graph) {
    for (size_t i = 0; i < graph.size(); i++) {
        cout << graph[i]->name_of_vertex << " " << graph[i]->viewing << endl;
        for (size_t j = 0; j < graph[i]->neighbour.size(); j++) {
            cout << graph[i]->neighbour[j]->name_of_vertex << " " << graph[i]-
>weight[j] << endl;
        }
        cout<< " _____ " << endl;
    }
}
```

```

    }
}

bool find_by_weight(vector<Vertex*> neighbour, vector<double> weight, Vertex*&
new_start) {
    double min = MAX;
    int index = -1;
    if (neighbour.size() == 0) {
        return false;
    }
    for (size_t i = 0; i < neighbour.size(); i++) {
        if (neighbour[i]->viewing == false && weight[i] < min) {
            min = weight[i];
            index = i;
        }
    }
    if (index >= 0) {
        new_start=neighbour[index];
        return true;
    }
    else {
        return false;
    }
}

bool greedy_search(Vertex* start, Vertex* end, vector<char>& way) {
    way.push_back(start->name_of_vertex);
    start->viewing = true;
    if (start->name_of_vertex == end->name_of_vertex) {
        return true;
    }
    else {
        Vertex *new_start=NULL;
        while (find_by_weight(start->neighbour, start->weight, new_start)) {
            if (greedy_search(new_start, end, way)) {
                return true;
            }
        }
        way.pop_back();
        return false;
    }
}

int main() {
    char start;
    char end;
    int number_start;
    int number_end;
    char from;
    char to;
    double weight;
    vector<Vertex*> graph;
    vector<char> way;

```

```

cin >> start >> end;
while (true) {
    if ((cin >> from) && (cin >> to) && (cin >> weight)) {
        creation_of_graph(from, to, weight, graph);
    }
    else {
        break;
    }
}
//print_graph(graph);
for (size_t i = 0; i < graph.size(); i++) {
    if (graph[i]->name_of_vertex == start) {
        number_start = i;
    }
    if (graph[i]->name_of_vertex == end) {
        number_end = i;
    }
}
if (greedy_search(graph[number_start], graph[number_end], way)) {
    for (size_t i = 0; i < way.size(); i++) {
        cout << way[i];
    }
}
system("pause");
return 0;
}

```

## Приложение В

### Исходный код программы lr2\_2.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class Vertex {
public:
char name_of_vertex;
bool viewing;
int heuristic;
double priority;
vector<Vertex*> neighbour;
vector<double> weight;
Vertex* from;
Vertex(char name) :name_of_vertex(name), viewing(false), heuristic(0),priority(0.0), from(NULL)
{
}
};

void creation_of_graph(char from, char to, char end, double weight, vector<Vertex*>& graph) {
int index1 = -1;
int index2 = -1;
for (size_t i = 0; i < graph.size(); i++) {
    if (from == graph[i]->name_of_vertex) {
        index1 = i;
    }
    if (to == graph[i]->name_of_vertex) {
        index2 = i;
    }
}
if (index1 == -1) {
    Vertex *tmp1 = new Vertex(from);
    graph.push_back(tmp1);
    index1 = graph.size() - 1;
}
if (index2 == -1) {
    Vertex *tmp2 = new Vertex(to);
    graph.push_back(tmp2);
    index2 = graph.size() - 1;
}
graph[index1]->neighbour.push_back(graph[index2]);
graph[index1]->weight.push_back(weight);
graph[index1]->heuristic = abs(end - from);
}

void print_graph(vector<Vertex*> graph) {
for (size_t i = 0; i < graph.size(); i++) {
    cout << graph[i]->name_of_vertex << " " << graph[i]->viewing << " " << graph[i]->heuristic << endl;
}
```

```

        for (size_t j = 0; j < graph[i]->neighbour.size(); j++) {
            cout << graph[i]->neighbour[j]->name_of_vertex << "
" << graph[i]->weight[j] << endl;
        }
        cout << "_____ " << endl;
    }

    bool cmp(const Vertex* a, const Vertex* b) {
        return a->priority > b->priority;
    }

    void creation_priority_list(Vertex* vertex, vector<Vertex*>& list) {
        bool flag = false;
        double current_path = vertex->priority - vertex->heuristic;
        for (size_t i = 0; i < vertex->neighbour.size(); i++) {
            if (vertex->neighbour[i]->viewing == false) {
                if ((vertex->neighbour[i]->priority == 0) ||
                    (current_path + vertex->weight[i] + vertex->neighbour[i]->heuristic < vertex->neighbour[i]->priority)) {
                    vertex->neighbour[i]->priority = current_path + vertex->weight[i] + vertex->neighbour[i]->heuristic;
                    vertex->neighbour[i]->from = vertex;
                    flag = false;
                    for (size_t j = 0; j < list.size(); j++) {
                        if (list[j]->name_of_vertex == vertex->neighbour[i]->name_of_vertex)
                            flag = true;
                    }
                    if (flag == false) {
                        list.push_back(vertex->neighbour[i]);
                    }
                }
            }
        }
    }

    void Astar(Vertex* start, Vertex* end, vector<char>& way) {
        vector<Vertex*> priority_list;
        bool exit_flag = false;
        Vertex* new_start = start;
        new_start->priority = new_start->heuristic;
        while (exit_flag == false) {
            if (new_start->name_of_vertex == end->name_of_vertex) {
                exit_flag = true;
            }
            else {
                new_start->viewing = true;
                creation_priority_list(new_start, priority_list);
                sort(priority_list.begin(), priority_list.end(), cmp);
                new_start = priority_list[priority_list.size() - 1];
            }
        }
    }

```



```

        priority_list.pop_back();
    }
}
while (new_start != NULL) {
    way.push_back(new_start->name_of_vertex);
    new_start = new_start->from;
}
}

int main() {
    char start;
    char end;
    int number_start;
    int number_end;
    char from;
    char to;
    double weight;
    vector<Vertex*> graph;
    vector<char> way;
    cin >> start >> end;
    while (true) {
        if ((cin >> from) && (cin >> to) && (cin >> weight)) {
            creation_of_graph(from, to, end, weight, graph);
        }
        else {
            break;
        }
    }
    //print_graph(graph);
    for (size_t i = 0; i < graph.size(); i++) {
        if (graph[i]->name_of_vertex == start) {
            number_start = i;
        }
        if (graph[i]->name_of_vertex == end) {
            number_end = i;
        }
    }
    Astar(graph[number_start], graph[number_end], way);
    for (int i = way.size()-1; i >=0; i--) {
        cout << way[i];
    }
    cout << endl;
    system("pause");
    return 0;
}

```