

**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: Алгоритмы поиска пути в графах**

Студент гр. 8383

\_\_\_\_\_

Костарев К.В.

Преподаватель

\_\_\_\_\_

Фирсов М.А.

Санкт-Петербург

2020

### **Цель работы.**

Ознакомиться с принципами работы жадного алгоритма поиска пути в графе и алгоритма A\*.

### **Задача.**

1. 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.

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

```
abcde
```

2. 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 № 2.

В  $A^*$  эвристическая функция для каждой вершины задаётся неотрицательным числом во входных данных.

### **Описание жадного алгоритма.**

Алгоритм подразумевает, что если на каждом этапе выбирать локальное оптимальное решение, то и все решение будет считаться оптимальным.

Т.е. для начальной вершины просматриваются все соседи и выбирается тот, который имеет наименьшее «расстояние» (вес) с изначальной, далее просматриваются по такому же принципу соседи этой вершины. Если у некоторой вершины нет непросмотренных соседей, то алгоритм просматривает соседей родителя этой вершины по такому же принципу.

### **Описание алгоритма A\*.**

Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость» (обычно обозначаемой как  $f(x)$ ). Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины ( $x$ ) из начальной (обычно обозначается как  $g(x)$ ) и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как  $h(x)$ ).

От жадного алгоритма, который тоже является алгоритмом поиска по первому лучшему совпадению, его отличает то, что при выборе вершины он учитывает, помимо прочего, весь пройденный до неё путь. Составляющая  $g(x)$  — это стоимость пути от начальной вершины, а не от предыдущей, как в жадном алгоритме.

В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение  $f(x)$ , после чего этот узел раскрывается. На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа — множеством частных решений, — которое размещается в очереди с приоритетом. Приоритет пути определяется по значению  $f(x) = g(x) + h(x)$ . Алгоритм продолжает свою работу до тех пор, пока значение  $f(x)$  целевой вершины не окажется меньшим, чем любое значение в очереди, либо пока всё дерево не будет просмотрено.

### **Описание структур данных для жадного алгоритма.**

1. class Path. Класс, необходимый для хранения информации об одном ребре графа. Имеет поля:
  - a. char start, end – начальная и конечная вершины ребра;
  - b. double length – вес;
  - c. bool checked – прошелся ли алгоритм по ребру или нет;
2. class minPath. Класс для хранения графа и с жадным алгоритмом:
  - a. char first, last – начальная и конечная вершины, путь для которых надо найти;

- b. `vector <Path> graph` – вектор, который хранит все ребра между вершинами;
- c. `stack <char> checkedPath` – стек из вершин, путь от начальной вершины до конечной;
- d. `void getMinimumPath()` – собственно алгоритм, который по мере своей работы добавляет в `checkedPath` самую оптимальную вершину на каждом пути, пока не дойдет до конечной.

### **Описание структур данных для алгоритма A\*.**

1. `class Path`. Класс, необходимый для хранения информации об одном ребре графа. Имеет поля:
  - a. `char start, end` – начальная и конечная вершины ребра;
  - b. `double length` – вес;
2. `class Dot`. Класс, необходимый для хранения информации об одной вершине графа:
  - a. `char name` – название вершины;
  - b. `bool checked` – обработана ли вершина алгоритмом;
  - c. `double f, g, h` – эвристическая функция;
  - d. `Dot(), Dot(char a)` – конструкторы;
3. `class minPath`. Хранение графа, методов для работы с ним и алгоритм:
  - a. `Dot first` – начальная вершина, откуда нужно найти путь;
  - b. `char last` – наименование вершины, до которой нужно найти путь;
  - c. `vector<Path> graph` – множество ребер;
  - d. `vector<Dot> dots` – множество необработанных алгоритмом вершин;
  - e. `map<char, double> heuristicOfDot` – словарь для хранения соответствия вершине ее эвристической функции;
  - f. `map<char, char> minPath` – словарь, карта для хранения последовательного пути от конечной до начальной вершины;

- g. `string reconstruct(char s, char f)` – «разворачивает» карту `minPath` от вершины `s` к вершине `f` и возвращает строку-последовательность вершин пути от `s` до `f`. Аргументы:
  - i. `char s, f` – начальная и конечная вершины;
- h. `string getMinimumPath()` – собственно алгоритм. Возвращает путь от `first` до `last`.

### **Сложность алгоритмов по времени.**

Сложность метода  $A^*$  по времени можно оценить как

$$O(N^{N-1}).$$

Так как в худшем случае проверяются все узлы  $N$  и все смежные ей вершины  $N-1$ .

Сложность жадного алгоритма по времени можно оценить как

$$O(N).$$

Так как в худшем случае проверяются все узлы  $N$ .

### **Сложность алгоритмов по памяти.**

Сложность метода  $A^*$  по памяти можно оценить как

$$O(2|V| + |E|).$$

Такая оценка исходит из того, что программа хранит все вершины и все ребра, а так же есть переменная, в которой количество элементов равно количеству вершин графа.

Сложность жадного алгоритма по памяти можно оценить как

$$O(|V| + |E|).$$

Так как программа хранит только граф.

### **Тестирование.**

Демонстрация работы программ, реализующих жадный алгоритм и  $A^*$ , приведена на рис. 1 и рис. 2 соответственно. Входные данные для жадного алгоритма:

a e  
a b 3.0  
b c 1.0  
c d 1.0  
a d 5.0  
d e 1.0

```
Enter the path you want to find:  
a e  
Enter the paths (enter '!' if you want to finish):  
a b 3.0  
b c 1.0  
c d 1.0  
a d 5.0  
d e 1.0  
!  
a->d it's no optimal  
a->b it's optimal  
b->c it's optimal  
c->d it's optimal  
d->e it's end!!!  
abcde
```

Рисунок 1 – Демонстрация жадного алгоритма

Входные данные для алгоритма A\*:

a 4.0  
b 1.0  
c 3.0  
d 2.0  
e 0.0  
a e  
a b 3.0  
b c 1.0  
c d 1.0  
a d 5.0  
d e 1.0

```

Enter the heuristic node functions (enter '!' if you want to finish):
a 4.0
b 1.0
c 3.0
d 2.0
e 0.0
!
Enter the path you want to find:
a e
Enter the paths (enter '!' if you want to finish):
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0
!
1 iteration.
  Dots: a(4,unchecked)
  Dots: a(4,checked) b(0,unchecked)
  Map: ab
  Dots: a(4,checked) b(4,unchecked) d(0,unchecked)
  Map: ad
2 iteration.
  Dots: a(4,checked) b(4,unchecked) d(7,unchecked)
  Dots: a(4,checked) b(4,checked) d(7,unchecked) c(0,unchecked)
  Map: abc
3 iteration.
  Dots: a(4,checked) b(4,checked) d(7,unchecked) c(7,unchecked)
  Dots: a(4,checked) b(4,checked) d(7,checked) c(7,unchecked) e(0,unchecked)
  Map: ade
4 iteration.
  Dots: a(4,checked) b(4,checked) d(7,checked) c(7,unchecked) e(6,unchecked)
ade

```

Рисунок 2 – Демонстрация алгоритма A\*

Тестирование программ приведено в табл. 1. В ходе тестирования все выходные данные оказались корректными.

Таблица 1 – Тестирование программ

Входные данные	Выходные данные (жадный алгоритм)	Выходные данные (алгоритм A*)
Только для A*: a 6.0 b 4.0 c 3.0 Общие: a c a a 1.0 a b 3.0 a c 6.0 b c 1.0	ac	abc
Только для A*: a 6.0 b 5.0 c 4.0 d 3.0	a	a



f 2.0 <b>Общие:</b> a a a a 60.0 a b 30.0 b d 15.0 d f 20.0 b c 10.0		
<b>Только для А*:</b> a 9.0 b 3.0 c 4.0 d 5.0 e 6.0 g 2.0 k 1.0 <b>Общие:</b> a k a b 3.0 a e 8.0 b c 8.0 b d 7.0 e g 4.0 e k 9.0 g k 1.0	aek	aegk
<b>Только для А*:</b> a 13.0 b 10.0 c 11.0 d 9.0 e 4.0 f 6.0 k 2.0 r 1.0 <b>Общие:</b> a r a b 8.0 a c 9.0 a d 6.0 b f 1.0 c e 5.0 d r 15.0 f k 1.0 e k 4.0 k r 10.0	adr	abfkr

### **Выводы.**

В данной лабораторной работе были изучены два алгоритма нахождения минимального пути в графе – жадный и  $A^*$ .  $A^*$  работает правильнее, чем жадный, но имеет большую сложность.

## ПРИЛОЖЕНИЕ А

### КОД ПРОГРАММЫ РЕАЛИЗАЦИИ ЖАДНОГО АЛГОРИТМА

```
#include <iostream>
#include <vector>
#include <string>
#include <stack>

class Path{ //ребро
public:
    char start{};    //начало
    char end{};      //конец
    double length{}; //длина
    bool checked = false; //обошел ли алгоритм ребро
};

class minPath{ //граф и алгоритм
public:
    char first{};    //вершина от которой найти путь
    char last{};     //до которой
    std::vector<Path> graph; //множество ребер
    std::stack<char> checkedPath; //путь
    void getMinimumPath(); //алгоритм
};

void minPath::getMinimumPath(){
    double minL = 100000.0;
    unsigned int minI = 100000;
    if (checkedPath.empty()){ //добавление начальной вершины
        checkedPath.push(first);
    } else{
        if (checkedPath.top() != first) //добавление вершины, если
она еще не была добавлена
            checkedPath.push(first);
    }
    for (unsigned int i = 0; i < graph.size(); i++){
        if (graph[i].start == first){ //находим начальную вершину
            if (graph[i].end == last){ //если сосед конечная
вершина то заканчиваем алгоритм
                std::cout << graph[i].start << "->" << last << "
it's end!!!" << std::endl;
                checkedPath.push(last);
                return;
            }
            if (graph[i].length < minL && !(graph[i].checked)){
//находим оптимального соседа
                minL = graph[i].length;
                minI = i;
            } else{
                std::cout << first << "->" << graph[i].end << "
it's no optimal" << std::endl;
            }
        }
    }
    if (minI == 100000){ //если оптимального соседа нет, то
прыгаем назад, к родителю
```

```

        checkedPath.pop();
        std::cout << first << " havn't optimal neighbors, jump to "
<< checkedPath.top() << std::endl;
        first = checkedPath.top();
        getMinimumPath();
        return;
    }
    std::cout << first << "->" << graph[minI].end << " it's
optimal" << std::endl;
    graph[minI].checked = true;
    first = graph[minI].end;    //теперь оптимальный сосед является
начальной вершиной
    getMinimumPath();
}

int main() {
    auto graphM = new minPath();
    std::stack<char> newP;
    Path path{};
    std::cout << "Enter the path you want to find:" << std::endl;
    std::cin >> graphM->first >> graphM->last;
    std::cout << "Enter the paths (enter '!' if you want to
finish):" << std::endl;
    while(true){
        std::cin >> path.start;
        if (path.start == '!')
            break;
        std::cin >> path.end >> path.length;
        graphM->graph.push_back(path);
    }
    graphM->getMinimumPath();
    do{
        newP.push(graphM->checkedPath.top());
        graphM->checkedPath.pop();
    } while (!graphM->checkedPath.empty());
    do{
        std::cout << newP.top();
        newP.pop();
    } while (!newP.empty());
    return 0;
}

```

## ПРИЛОЖЕНИЕ Б

### КОД ПРОГРАММЫ РЕАЛИЗАЦИИ АЛГОРИТМА А\*

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

class Path{ //ребро
public:
    char start{};    //откуда
    char end{};      //куда
    double length{}; //длина(вес)
};

class Dot{ //вершина
public:
    char name{};
    bool checked{}; //проходили ли мы ее (для астар)
    double g{};
    double f{};
    double h{};
    Dot(){ //конструктор
        checked = false; //по умолчанию вершина не пройдена
    }
    explicit Dot(char a){ //перезагрузка
        name = a;
    }
};

class minPath{ //граф и алгоритм
public:
    Dot first; //откуда нужно найти путь
    char last{}; //куда
    std::vector<Path> graph; //множество ребер
    std::vector<Dot> dots; //вершины которые надо пройти для астар
    std::map<char, double> heuristicOfDot; //эвристические функции
    //вершин введенные пользователем
    std::map<char, char> minPath; //карта минимального пути
    std::string getMinimumPath(); //алгоритм
    std::string reconstruct(char s, char f){ //вывод минимального пути
        //из алгоритма
        std::string newStr;
        char curr = f;
        newStr += f;
        while (curr != s){
            curr = minPath[curr];
            newStr += curr;
        }
        std::reverse(newStr.begin(), newStr.end());
        return newStr;
    }
    int heuristicFunc(char a){ //обращение к эвристической функции
        return heuristicOfDot[a];
    }
};
```

```

bool allchecked(){          //все ли вершины из dots пройдены
    for (auto & dot : dots){
        if (!dot.checked)
            return false;
    }
    return true;
}
bool isClosed(char a){      //пройдена ли вершина a из dots
    for (auto & i: dots){
        if (i.name == a && i.checked)
            return true;
    }
    return false;
}
unsigned int inDots(Dot* a){  //индекс+1 вершины a из dots (если
нет то 0)
    for (unsigned int i = 0; i < dots.size(); i++){
        if (dots[i].name == a->name)
            return i+1;
    }
    return 0;
}
void printDots(){           //печать dots
    for (auto & i: dots){
        std::cout << i.name << "(" << i.f << ",";
        if (i.checked){
            std::cout << "checked) ";
        } else{
            std::cout << "unchecked) ";
        }
    }
    std::cout << std::endl;
}
};

std::string minPath::getMinimumPath(){
    first.g = 0;           //сначала вычисляем эвристику для начальной вершины
    first.h = heuristicFunc(first.name);
    first.f = first.g + first.h;
    dots.push_back(first); //закидываем начальную в dots
    int iterations = 0;
    while (!allchecked()){ //пока все вершины не пройдены, если пройдены
то пути нет
        iterations++;
        std::cout << iterations << " iteration." << std::endl;
        double fmin = 10000.0;
        unsigned int imin = 0;
        std::cout << "\tDots: ";
        printDots();
        for (unsigned int i = 0; i < dots.size(); i++) {           //нахождение
в dots вершины с минимальной f
            if (dots[i].f < fmin && !dots[i].checked){
                fmin = dots[i].f;
                imin = i;
            }
        }
    }
}

```

```

        if (dots[imin].name == last){ //если это конечная вершина
завершаем алгоритм
            return reconstruct(first.name, last);;
        }
        dots[imin].checked = true; //взятая вершина метится пройденной
        for (auto & i: graph){ //находим соседей вершины
            if (i.start == dots[imin].name){
                bool gFuncIsBetter;
                if (isClosed(i.end)){ //соседи которые уже пройдены
пропускаем
                    continue;
                }
                auto neighbor = new Dot(i.end);
                for (auto & dot : dots){
                    if (neighbor->name == dot.name)
                        neighbor = &dot;
                }
                double length = i.length + dots[imin].g; //считаем
значение функции
                if (!inDots(neighbor)){ //если соседа нет в dots то
пушаем
                    dots.push_back(*neighbor);
                    std::cout << "\tDots: ";
                    printDots();
                    gFuncIsBetter = true;
                } else{
                    gFuncIsBetter = neighbor->g > length;
                }
                if (gFuncIsBetter){ //если нашли лучшее значение функции
для вершины
                    minPath[neighbor->name] = dots[imin].name; //создаем
цепь сосед-вершина
                    dots[inDots(neighbor)-1].g = length;
//перезаписываем данные
                    dots[inDots(neighbor)-1].h =
heuristicFunc(dots[inDots(neighbor)-1].name);
                    dots[inDots(neighbor)-1].f = dots[inDots(neighbor)-
1].g + dots[inDots(neighbor)-1].h;
                    std::cout << "\tMap: " << reconstruct(first.name,
neighbor->name) << std::endl;
                }
            }
        }
        return "0"; //если путь не найден
    }

int main() {
    auto graphM = new minPath(); //граф с которым будет работа
    Path path{};
    std::cout << "Enter the heuristic node functions (enter '!' if you
want to finish):" << std::endl;
    while (true) { //ввод эвристических функций для вершин
        char dot;
        std::cin >> dot;
        if (dot == '!')
            break;
    }
}

```

```

        std::cin >> graphM->heuristicOfDot[dot];
    }
    std::cout << "Enter the path you want to find:" << std::endl;
    std::cin >> graphM->first.name >> graphM->last;
    std::cout << "Enter the paths (enter '!' if you want to finish):" <<
std::endl;
    while(true){        //ввод ребер
        std::cin >> path.start;
        if (path.start == '!')
            break;
        std::cin >> path.end >> path.length;
        graphM->graph.push_back(path);
    }
    std::cout << graphM->getMinimumPath(); //вывод результат алгоритма
    return 0;
}

```