**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: Жадный алгоритм и A***

Студент гр. 9383 _____ Орлов Д.С.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

**Цель работы.**

Применить на практике знания о построение жадного алгоритма поиска пути в графе и алгоритма A* – «A звездочка». Реализовать программу, которая считывает граф и находит в нем путь от стартовой вершины к конечной с помощью жадного алгоритма и алгоритма A*.

**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.

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.

**Вариант 3.**

Написать функцию, проверяющую эвристику на допустимость и монотонность.

**Основные теоретические положения.**

**Описание жадного алгоритма.** Для решения задачи был реализован жадный алгоритм поиска пути в графе. Построение пути начинается с выбора стартовой вершины. Дальше на каждом шаге делается следующие:

1. Выбирается ребро с наименьшим весом, выполняется переход по этому ребру к новой вершине, которая становится текущей и помечается посещенной.

2. Если из текущей вершины нет исходящих ребер в не посещенные вершины, текущей вершиной становится предыдущая посещенная вершина.

3. Если текущая вершина совпадает с конечной или рассмотрены все возможные вершины, алгоритм завершен.

**Оценка сложности жадного алгоритма.** Так как алгоритм перебирает все ребра и проходит по всем вершинам (в худшем случае), сложность данного алгоритма будет O(V+E)

**Алгоритм A*** — алгоритм поиска, который находит во взвешенном графе маршрут наименьшей стоимости от начальной вершины довыбранной конечной.

Описание:

В процессе работы алгоритма для вершин рассчитывается функция $f(v) = g(v)+h(v)$, где

$g(v)$ — наименьшая стоимость пути в v из стартовой вершины,

$h(v)$ — эвристическое приближение стоимости пути от v до конечной цели.

Фактически, функция $f(v)$ — длина пути до цели, которая складывается из пройденного расстояния $g(v)$ и оставшегося расстояния $h(v)$. Исходя из этого, чем меньше значение $f(v)$, тем раньше мы откроем вершину v, так как через неё мы предположительно достигнем расстояние до цели быстрее всего. Открытые алгоритмом вершины можно хранить в очереди с приоритетом по значению $f(v)$. A* действует подобно алгоритму Дейкстры и просматривает среди всех маршрутов, ведущих к цели, сначала те, которые благодаря имеющейся информации (эвристическая функция) в данный момент являются наилучшими.

**Оценка сложности алгоритма A*.** Функция полного пути f(V)=g(V)+h(V) вычисляется как минимальный уже построенный путь g(V), и эвристическая функция h(V). От выбора этой самой эвристической функции зависит сложность алгоритма A*. В худшем случае количество обрабатываемых вершин растет экспоненциально в зависимости от длины оптимального пути. Но если выполнено следующее неравенство, сложность становится полиномиальной:

$|h(V) - h^*(V)| \leq O(\log h^*(x))$, где $h^*(V)$ – точная оценка длины пути из текущей вершины V в конечную.

**Свойства:**

Чтобы A* был оптимален, выбранная функция h(v) должна быть допустимой эвристической функцией.

Говорят, что эвристическая оценка h(v) ***допустима***, если для любой вершины v значение h(v) меньше или равно весу кратчайшего пути от v до цели.

Допустимая оценка является оптимистичной, потому что она предполагает, что стоимость решения меньше, чем оно есть на самом деле.

Второе, более сильное условие — функция h(v) должна быть монотонной. Эвристическая функция h(v) называется ***монотонной*** (или преемственной), если для любой вершины v1 и ее потомка v2 разность h(v1) и h(v2) не превышает фактического веса ребра c(v1, v2) от v1 до v2, аэвристическая оценка целевого состояния равна нулю.

**Выполнение работы:**

В программе реализована структура Vertex для хранения информации о вершинах графа, содержащая:

Переменные:

char name — имя вершины, bool used — посещали ли мы эту вершину, float f — значение функции f в алгоритме A*, float h — значение эвристики, float g — пройденное расстояние от начала, float goalDistance — расстояние до цели, float minCost — расстояние до следующей

вершины в кратчайшем пути, Vertex* parent — родитель данной вершины в кратчайшем пути, std::vector <std::pair<Vertex*, float>> edge — хранит множество ребер в формате: <вершина, куда идет ребро; стоимость>

Методы:

viod makeEdge(Vertex* v, float cost) — добавляет ребро между данной вершиной и вершиной v со стоимостью cost.

В программе были использованы следующие функции:

Vertex* castomFind(std::vector <Vertex*> graph, char v) — для поиска вершины в графе по имени;

Vertex* addVertex (std::vector <Vertex*> &graph, char v) — для добавления вершины в граф.

int heuristic(char first, char second) — для вычисления эвристической функции для вершины.

bool findInQ(std::vector <Vertex*> Q, Vertex v) — для поиска вершины в множестве рассматриваемых вершин.

std::vector <char> greedy(std::vector <Vertex*> &graph, char start, char finish) — реализация жадного алгоритма поиска кратчайшего пути.

bool aStar(std::vector <Vertex*> &graph, char start, char finish) — реализация алгоритма A*.

void checkHeuristic(std::vector <Vertex*> graph, char start, char finish) — для проверки эвристики на допустимость и монотонность.

**Описание алгоритма работы программы:**

Программа получает на вход 2 вершины startV и finishV, затем ребра - <исходящая вершина> <входящая вершина> <вес> до окончания ввода (ctrl+Z). Сначала считываются названия вершин в переменные типа Vertex, которые

добавляются в вектор graph, только в том случае, если вершин с такими же именами еще нет в векторе. Затем добавляется соответствующее ребро: с помощью метода makeEdge в множество ребер для исходящей вершины добавляется указатель на входящую вершину и стоимость. Таким образом формируем вектор graph. Далее с помощью функции aStar() ищем наименьший путь в графе: пока начальная вершина не равна конечной. Составляем множество вершин, которые требуется рассмотреть, выбираем вершину с наименьшим значением эвристической функции и просматриваются её соседи. Для каждого из соседей обновляется расстояние g, значение эвристической функции f и он добавляется в множество. Таким образом, вершины с минимальным значением эвристической функции f записываются в parent. Затем пробегаемся и собираем путь с конца, записываем результат в строку и переворачиваем.

С помощью функции checkHeuristic исследуем эвристику на допустимость (проверяем, что для любой вершины v значение h(v) меньше или равно весу кратчайшего пути от v до цели.) и монотонность (проверяем, что для любой вершины v1 и ее потомка v2 разность h(v1) и h(v2) не превышает фактического веса ребра c(v1,v2) от v1 до v2, а эвристическая оценка целевого состояния равна нулю).

**Тестирование**

1) Входные данные:

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

Выходные данные:

A star way:
ade

Greedy way:
abcde

Heuristic is valid!

Heuristic is monotonic!

2) Входные данные:

a g

a b 3.0

a c 1.0

b d 2.0

b e 3.0

d e 4.0

e a 3.0

e f 2.0

a g 8.0

f g 1.0

Выходные данные:

A star way:
ag

Greedy way:
abdefg

Heuristic is valid!

Heuristic is not monotonic!

3) Входные данные:

a d

a b 1.0

b c 9.0

c d 3.0

a d 9.0

a e 1.0

e d 3.0

Выходные данные:

A star way:
aed

Greedy way:
abcd

Heuristic is not valid!

Heuristic is not monotonic!

4) Входные данные:

a g

a b 3.0

a c 1.0

b d 2.0

b e 3.0

d e 4.0

e a 1.0

e f 2.0

a g 8.0

f g 1.0

Выходные данные:

A star way:
ag

Greedy way:
abdeag

Heuristic is valid!

Heuristic is not monotonic!

**Вывод.**

Были применены на практике знания о построение жадного алгоритма

поиска пути в графе и алгоритма A* – «A звездочка». Реализована программа, которая считывает граф и находит в нем путь от стартовой вершины к конечной с помощью жадного алгоритма и алгоритма A*.

## Приложение А
## Исходный код программы

Название файла: main.cpp

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

#define INF 1000000

struct Vertex
{
public:
    char name;
    Vertex* parent = nullptr;
    bool used;
    float f, h, g, goalDistance, minCost;
    std::vector <std::pair<Vertex*, float>> edge;

    Vertex(char nameV)
    {
        name = nameV;
        f = 0;
        h = 0;
        g = 0;
        goalDistance = 0;
        minCost = 0;
        used = false;
    }

    void makeEdge(Vertex* v, float cost)
    {
        edge.push_back(std::make_pair(v,cost));
    }
};
```

```
Vertex* castomFind(std::vector <Vertex*> graph, char v)
{
    for(int i = 0; i < graph.size(); i++)
    {
        if(graph[i]->name == v)
        {
            return graph[i];
        }
    }
    return nullptr;
}

Vertex* addVertex (std::vector <Vertex*> &graph, char v)
{
    Vertex* ptr = castomFind(graph, v);
    Vertex* ver;
    if(ptr == nullptr)
    {
        ver = new Vertex(v);
        graph.push_back(ver);
        return graph.back();
    }
    return ptr;
}

int heuristic(char first, char second)
{
    return abs(first - second);
}

bool findInQ(std::vector <Vertex*> Q, Vertex v)
{
    for(int i = 0; i < Q.size(); i++)
    {
        if(Q[i]->name == v.name)
        {
```

```
            return true;
        }
    }
    return false;
}

std::vector <char> greedy(std::vector <Vertex*> &graph, char start,
char finish)
{
    std::vector <char> greedyWay;
    greedyWay.push_back(start);

    Vertex* ptr = castomFind(graph, start);
    Vertex* newPtr = nullptr;
    int minWay = INF;

    while(ptr->name != finish || greedyWay.empty())
    {
        for(auto e : ptr->edge)
        {
            if(e.second < minWay && e.first->used == false)
            {
                minWay = e.second;
                newPtr = e.first;
            }
        }
        if(minWay == INF)
        {
            ptr->used = true;
            greedyWay.pop_back();
            ptr = castomFind(graph, greedyWay.back());
            continue;
        }
        greedyWay.push_back(newPtr->name);
        ptr = newPtr;
        ptr->used = true;
```

```
        minWay = INF;

    }
    return greedyWay;
}

bool aStar(std::vector <Vertex*> &graph, char start, char finish)
{
    //a star
    std::vector <Vertex*> Q;
    Q.push_back(castomFind(graph, start));
    Q[0]->g = 0.0;
    Q[0]->f = Q[0]->g + Q[0]->h;
    int index, minF = INF, tentativeScore;
    while(!Q.empty())
    {
        for(int i = 0; i < Q.size(); i++)
        {
            if(Q[i]->f <= minF)
            {
                index = i;
                minF = Q[i]->f;
            }
        }
        minF = INF;

        if(Q[index]->name == finish)
        {
            //---------------
            Vertex* ptr = castomFind(graph, finish);
            Vertex* startPtr = castomFind(graph, start);
            while(ptr != startPtr)
            {
                ptr->parent->goalDistance = ptr->goalDistance + ptr-
>parent->minCost;
                ptr = ptr->parent;
```

```
            }
            //---------------
            return true;
        }

        Q[index]->used = true;
        for(int i = 0; i < Q[index]->edge.size(); i++)
        {
            tentativeScore = Q[index]->g + Q[index]->edge[i].second;
            if(Q[index]->edge[i].first->used == true &&
tentativeScore >= Q[index]->edge[i].first->g)
            {
                continue;
            }
            if(Q[index]->edge[i].first->used == false ||
tentativeScore < Q[index]->edge[i].first->g)
            {
                Q[index]->edge[i].first->parent = Q[index];
                //----------
                Q[index]->minCost = Q[index]->edge[i].second;
                //----------
                Q[index]->edge[i].first->g = tentativeScore;
                Q[index]->edge[i].first->f = Q[index]-
>edge[i].first->g + Q[index]->edge[i].first->h;
                if(findInQ(Q, *Q[index]->edge[i].first) == false)
                {
                    Q.push_back(Q[index]->edge[i].first);
                }
            }
        }
        Q.erase(Q.begin() + index);
    }
    //a star

    for(auto i : graph)
    {
        i->used = false;
```

```
    }

    return false;
}

void checkHeuristic(std::vector <Vertex*> graph, char start, char
finish)
{
    //valid

    for(auto i : graph)
    {
        aStar(graph, i->name, finish);
    }

    bool flag = true;

    for(auto ptr : graph)
    {
        if(ptr->goalDistance == 0 && ptr->name != finish)
        {
            ptr->goalDistance = INF;
        }
        if(ptr->goalDistance < ptr->h)
        {
            flag = false;
            break;
        }
    }

    if(flag == true)
    {
        std::cout<<"\nHeuristic is valid!\n";
    }
    else
    {
```

```
        std::cout<<"\nHeuristic is not valid!\n";
    }


    //monotone

    if(castomFind(graph, finish)->h == 0)
    {
        for(auto v1 : graph)
        {
            for(int i = 0; i < v1->edge.size(); i++)
            {
                if(v1->h - v1->edge[i].first->h > v1-
>edge[i].second)
                {
                    std::cout<<"\nHeuristic is not monotonic!\n";
                    goto exit;
                }
            }
        }
        std::cout<<"\nHeuristic is monotonic!\n";
    }
    else
    {
        std::cout<<"\nHeuristic is not monotonic!\n";
    }
exit:
    ;

}


int main()
{
    char startV, finishV, firstV, secondV;
    float cost;
```

```
std::vector <Vertex*> graph;

std::cin>>startV>>finishV;

while(std::cin>>firstV)
{
    std::cin>>secondV>>cost;

    Vertex* ptr1 = addVertex(graph, firstV);
    Vertex* ptr2 = addVertex(graph, secondV);

    ptr1->makeEdge(ptr2, cost);
}

for(int i = 0; i < graph.size(); i++)
{
    graph[i]->h = heuristic(graph[i]->name, finishV);
}


std::cout<<"A star way:\n";

bool pathAStar = aStar(graph, startV, finishV);

if(pathAStar == true)
{
    Vertex* ptr = castomFind(graph, finishV);
    std::string way;

    Vertex* startPtr = castomFind(graph, startV);

    while(ptr != startPtr)
    {
        way+=ptr->name;
        ptr = ptr->parent;
    }
```

```
    way+=ptr->name;

    reverse(way.begin(), way.end());
    std::cout<<way<<"\n\n";
}
else
{
    std::cout<<"No path.\n\n";
}

for(auto i : graph)
{
    i->used = false;
}

std::cout<<"Greedy way:\n";

std::vector <char> greedyWay = greedy(graph, startV, finishV);

if(!greedyWay.empty())
{
    for(auto i : greedyWay)
    {
        std::cout<<i;
    }
    std::cout<<"\n";
}
else
{
    std::cout<<"No path.\n";
}

for(auto i : graph)
{
    i->used = false;
}
```

```
    checkHeuristic(graph, startV, finishV);

    for(int i = 0; i < graph.size(); i++)
    {
        delete graph[i];
    }

    return 0;
}
```