

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 на графах

Студент гр. 8383

Бессуднов Г. И.

Преподаватель

Фирсов М. А.

Санкт-Петербург

2020

Цель работы.

Разработать программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом A^* , а также при помощи жадного алгоритма.

Постановка задачи.

Метод A^* :

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

Вар. 2. В A^* эвристическая функция для каждой вершины задаётся неотрицательным числом во входных данных.

Жадный алгоритм:

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

Вар. 2. В A^* эвристическая функция для каждой вершины задаётся неотрицательным числом во входных данных.

Описание алгоритма A^* .

В начале работы в очередь с приоритетом (далее просто очередь) помещаются все ребра, исходящие из начальной вершины. Приоритет у очереди следующий: она сортирует ребра по возрастанию значения $F(x)$ в конечной вершине. Значение $F(x)$ вершины складывается из эвристической функции для нее и пройденного расстояния от начала. Далее берется самое первое ребро в очереди и в очередь добавляются все ребра, которые выходят из

конечной вершины текущего. Алгоритм заканчивает работу, если конечная вершина текущего ребра есть финишная вершина.

Описание жадного алгоритма.

На каждом шаге алгоритма просматриваются все исходящие ребра из вершины, начиная с текущей. Среди них выбирается ребро с наименьшим весом и по нему происходит переход, оно помечается как просмотренное. Алгоритм заканчивает работу, когда было выбрано ребро, ведущее в точку назначения. Если у вершины нет непросмотренных ребер, или она не имеет исходящих ребер, то алгоритм делает шаг назад и заново просматривает родительскую точку.

Описание структур данных.

`enum Colors { Black, White };` - перечисление, необходимое для обозначения просмотренных точек. Черный - точка не просмотрена, белый - просмотрена.

```
class Vertex {
public:
    char name; //имя точки
    double heuristic; //эвристическое значение точки
    double pathLength; //длина пути до точки
    double cost; //эвристическая оценка
    Colors color; //цвет точки

    bool operator< (const Vertex &v) const;

    void evaluateCost();
};
```

Класс для хранения информации о точках. Содержит функцию `void evaluateCost()` необходимую для подсчета общей стоимости передвижения в точку.

```
typedef struct Edge {
    Vertex start; //начальная точка
```

```

Vertex end; //конечная точка
double weight; //вес
}

```

Класс для хранения информации о ребре.

```

struct edgeComp_Q {
    bool operator() (Edge *e1, Edge *e2);
};

```

Структура-компаратор для очереди с приоритетом. По ней определяется приоритет в очереди. Содержит метод `bool operator() (Edge *e1, Edge *e2)` принимающий два ребра `e1`, `e2` и сравнивающий их. Возвращает `true`, если `e1` больше `e2` и `false` в противном случае.

```

bool edgeComp_V(const Edge &e1, const Edge &e2)

```

Функция-компаратор для сортировки вектора ребер. Принимает два ребра `e1`, `e2` и сравнивает их. Возвращает `true`, если `e1` больше `e2` и `false` в противном случае.

`class Graph` - класс для хранения графа и работы с ним. Содержит следующие переменные:

`std::map<Vertex, std::vector<Edge>>::iterator graphIt;` - итератор для графа

`std::map<char, double>::iterator heuristicsIt;` - итератор для эвристической карты

`std::vector<Edge>::iterator edgeIt;` - итератор для вектора ребер

`std::map<Vertex, std::vector<Edge>> graph;` - граф

`std::map<char, double> heuristics;` - карта эвристик

`std::priority_queue<Edge*, std::vector<Edge*>, edgeComp_Q> edgeQueue;` - очередь с приоритетом

`std::map<char, char> path;` - карта для заполнения пути

`std::string pathName;` - строка с путем

`Vertex start;` - начальная вершина

`Vertex end;` - конечная вершина

`HANDLE hConsole;` - управление консолью

Так же содержит следующие важные функции:

`void handfillHeuristics()` - функция, позволяющая вручную заполнить эвристики.

`void autofillHeuristics()` - функция, позволяющая автоматически заполнить эвристики.

`void generatePath()` - функция для создания строки с путем в графе.

`void greedPath()` - функция нахождения пути по жадному алгоритму.

`void aStarPath()` - функция нахождения пути по алгоритму A*.

`std::map<Vertex, std::vector<Edge>> graph;` - структура для хранения графа. Граф хранится как список смежности. В структуре `map` параметр `Vertex` отвечает за вершину, а вектор, состоящий из `Edge` - это ребра, смежные с этой вершиной.

Сложность алгоритмов по времени.

Сложность зависит от эвристической функции. Сложность может стать полиномиальной, если удовлетворено следующее условие:

$$|h(x) - h^*(x)| \leq O(\log(h^*(x)))$$

В худшем случае сложность будет расти экспоненциально.

Сложность жадного алгоритма по времени можно оценить как

$$O(|E|).$$

В худшем случае проверяются все E ребер.

Сложность алгоритмов по памяти.

Сложность жадного алгоритма по памяти можно оценить как

$$O(|V|^2).$$

Так как в худшем случае в путь будет добавлено столько вершин, сколько ребер в графе + 1.

Сложность A* алгоритма по памяти можно оценить как

$$O(|E|).$$

Так как в худшем случае в очередь с приоритетом будут положены все ребра графа.

Тестирование.

Пример вывода результата для A*

```

A STAR ALGORITHM INFORMATION

HEURISTICS
dot:  value:
a      4
b      3
c      2
d      1
e      0

Current edge: a<--->d (5)      dot: d (cost: 6, pathLength: 5)
QUEUE:
Current edge: a<--->d (5)      dot: d (cost: 6, pathLength: 5)
Current edge: a<--->b (3)      dot: b (cost: 6, pathLength: 3)

GRAPH:
a<--->b
a<--->d
b<--->c
c<--->d
d<--->e
<----->

Current edge: d<--->e (1)      dot: e (cost: 6, pathLength: 6)
QUEUE:
Current edge: d<--->e (1)      dot: e (cost: 6, pathLength: 6)
Current edge: a<--->b (3)      dot: b (cost: 6, pathLength: 3)

GRAPH:
a<--->b
a<--->d
b<--->c
c<--->d
d<--->e
<----->

ade

```

Пример вывода результата для тех же входных данных для жадного алгоритма:

```

GREED ALGORITHM INFORMATION

Current edge: a<--->b
a<--->b (3)
a<--->d (5)
b<--->c (1)
c<--->d (1)
d<--->e (1)
<----->

Current edge: b<--->c
a<--->b (3)
a<--->d (5)
b<--->c (1)
c<--->d (1)
d<--->e (1)
<----->

Current edge: c<--->d
a<--->b (3)
a<--->d (5)
b<--->c (1)
c<--->d (1)
d<--->e (1)
<----->

Current edge: d<--->e
a<--->b (3)
a<--->d (5)
b<--->c (1)
c<--->d (1)
d<--->e (1)
<----->

abcde

```

№	Входные данные	Результат A*	Результат ЖА
1	a b a b 1.0 * a 1.0 b 1.0	ab	ab
2	a c a a 3.0 a b 2.5 b c 0.2 a c 4.0 * 3.0	abc	abc

	1.1 90.0 0.0001		
3	a l a b 1 a f 3 b c 5 b g 3 f g 4 c d 6 d m 1 g e 4 e h 1 e n 1 n m 2 g i 5 i j 6 i k 1 j l 5 m j 3 * 1.0 3.0 1.5 4.3 2.5 10.6 3.8 7.4	abgenmjl	abgenmjl

	8.2 6.3 9.1 11.1 0.3 5.0		
4	a i a d 3.0 d i 1.0 a c 2.0 c h 11.0 h i 1.0 a b 1.0 b e 10.0 e g 3.0 e f 2.0 * 11 10 9 8 7 6 5 4 3	adi	achi

Вывод.

В ходе выполнения лабораторной работы был реализован алгоритм A^* и жадный алгоритм для нахождения минимального пути в графе от одной вершины к другой.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include "pch.h"
#include <iostream>
#include <vector>
#include <stack>
#include <string>
#include <map>
#include <algorithm>
#include <queue>
#include <windows.h>

//перечисление цветов
enum Colors { Black, White };

//класс вершины
class Vertex {
public:
    char name; //имя точки
    double heuristic; //эвристическое значение точки
    double pathLength; //длина пути до точки
    double cost; //эвристическая оценка
    Colors color; //цвет точки
    Vertex() : color(Colors::Black) {}
    Vertex(char a) : name(a), color(Colors::Black) {}

    bool operator< (const Vertex &v) const {
        return this->name < v.name;
    }

    void evaluateCost() {
        cost = pathLength + heuristic;
    }
};

//класс ребра
typedef struct Edge {
    Vertex start; //начальная точка
    Vertex end; //конечная точка
    double weight; //вес

    Edge() {}
```

```

    Edge(Vertex start, Vertex end, double weighth) : start(start),
    end(end), weighth(weighth) {}

    bool operator< (const Edge &e) const {
        return this->weighth < e.weighth;
    }

}Edge;

//компаратор для очереди ребер
struct edgeComp_Q {
    bool operator() (Edge *e1, Edge *e2) {
        if (e1->end.cost == e2->end.cost) {
            return e1->end.heuristic > e2->end.heuristic;
        }
        return (e1->end.cost > e2->end.cost);
    }
};

//компаратор для вектора ребер
bool edgeComp_V(const Edge &e1, const Edge &e2){
    if (e1.weighth == e2.weighth) {
        return e1.end.name < e2.end.name;
    }
    return e1.weighth < e2.weighth;
}

//класс графа
class Graph {
private:
    std::map<Vertex, std::vector<Edge>>::iterator graphIt; //итератор
для графа
    std::map<char, double>::iterator heuristicsIt; //итератор для
эвристической карты
    std::vector<Edge>::iterator edgeIt; // итератор для вектора ребер

    std::map<Vertex, std::vector<Edge>> graph; //граф
    std::map<char, double> heuristics; //карта эвристик

    std::priority_queue<Edge*, std::vector<Edge*>, edgeComp_Q>
edgeQueue; //очередь с приоритетом
    std::map<char, char> path; //карта для заполнения пути
    std::string pathName; //строка с путем

```

```

Vertex start; //начальная вершина
Vertex end; //конечная вершина

HANDLE hConsole; //управление консолью

//функция для печати карты эвристик
void printHeuristics() {
    std::cout << "HEURISTICS\ndot:\tvalue:" << std::endl;
    for (heuristicsIt = heuristics.begin(); heuristicsIt !=
heuristics.end(); heuristicsIt++) {
        std::cout << heuristicsIt->first << "\t" <<
heuristicsIt->second << std::endl;
    }

    std::cout << "\n" << std::endl;
}

void printGreedMessage() {
    SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN | 0);
    std::cout << "GREED ALGORITHM INFORMATION\n\n" << std::endl;
    SetConsoleTextAttribute(hConsole, 7 | 0);
}

//функция для печати информации об одной итерации ЖА
void printGreedIteration(Edge &current) {
    SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN | 0);
    std::cout << "Current edge: " << current.start.name << "<--->"
<< current.end.name << std::endl;
    SetConsoleTextAttribute(hConsole, 7 | 0);

    for (graphIt = graph.begin(); graphIt != graph.end();
graphIt++) {
        for (auto &edge : graphIt->second) {
            if (edge.end.name == current.end.name &&
edge.start.name == current.start.name) {
                SetConsoleTextAttribute(hConsole,
FOREGROUND_GREEN | 0);
                std::cout << edge.start.name << "<--->" <<
edge.end.name << " (" << edge.weight << ")" << std::endl;
                SetConsoleTextAttribute(hConsole, 7 | 0);
            } else {

```

```

        std::cout << edge.start.name << "<--->" <<
edge.end.name << " (" << edge.weight << ")" << std::endl;
    }
}

std::cout << "<----->\n" << std::endl;
}

void printAStarMessage() {
    SetConsoleTextAttribute(hConsole, FOREGROUND_BLUE | 0);
    std::cout << "A STAR ALGORITHM INFORMATION\n\n" << std::endl;
    SetConsoleTextAttribute(hConsole, 7 | 0);
}

//функция для печати информации об одной итерации A*
void printAStarIteration(std::priority_queue<Edge*,
std::vector<Edge*>, edgeComp_Q> queue, Edge current) {
    SetConsoleTextAttribute(hConsole, FOREGROUND_BLUE | 0);
    std::cout << "Current edge: " << current.start.name << "<--->"
<< current.end.name << " (" << current.weight << ") "
        << "\tdot: " << current.end.name << " (cost: " <<
current.end.cost << ", pathLength: "
        << current.end.pathLength << ")" << std::endl;
    SetConsoleTextAttribute(hConsole, 7 | 0);

    std::cout << "QUEUE:" << std::endl;
    Edge *cur;
    while (!queue.empty()) {
        cur = queue.top();
        queue.pop();
        std::cout << "Current edge: " << cur->start.name << "<--
->" << cur->end.name << " (" << cur->weight << ") "
            << "\tdot: " << cur->end.name << " (cost: " << cur-
>end.cost << ", pathLength: "
            << cur->end.pathLength << ")" << std::endl;
    }
    std::cout << "\n" << std::endl;

    std::cout << "GRAPH:" << std::endl;
    for (graphIt = graph.begin(); graphIt != graph.end();
graphIt++) {
        for (auto &edge : graphIt->second) {

```



```

        for (heuristicsIt = heuristics.begin(); heuristicsIt !=
heuristics.end(); heuristicsIt++) {
            if (heuristicsIt->second < minHeuristic) {
                minHeuristic = heuristicsIt->second;
            }
        }

        for (heuristicsIt = heuristics.begin(); heuristicsIt !=
heuristics.end(); heuristicsIt++) {
            heuristicsIt->second += minHeuristic;
        }

        printHeuristics();
    }

public:
    Graph() {

        hConsole = GetStdHandle(STD_OUTPUT_HANDLE);

        std::cout << "Enter start: ";
        std::cin >> start.name;
        std::cout << "Enter end: ";
        std::cin >> end.name;
        Vertex v1;
        Vertex v2;
        double weigth;
        std::cout << "Enter edges (from, to, weigth)\nTo end input
press *" << std::endl;
        while (true) {
            std::cin >> v1.name;
            if (v1.name == '*') break;
            std::cin >> v2.name >> weigth;
            graph[v1].push_back(Edge(v1, v2, weigth));
            graph[v2];
        }

        handfillHeuristics();

    }

    void greedPath();
    void aStarPath();
};

```

```

//функция восстановления пути
void Graph::generatePath() {
    pathName.clear();
    char currentSymbol;
    currentSymbol = end.name;

    while (currentSymbol != start.name) {
        pathName += currentSymbol;
        currentSymbol = path[currentSymbol];
    }

    pathName += currentSymbol;

    std::reverse(pathName.begin(), pathName.end());

    std::cout << pathName << std::endl;
}

void Graph::greedPath() {

    printGreedMessage();

    //сначала сортируем для каждой вершины исходящие из нее ребра
    for (auto &node : graph) {
        std::sort(node.second.begin(), node.second.end(), edgeComp_V);
    }

    Edge current;
    current.end = start;

    for (;;) {

        //условие выхода из цикла
        if (current.end.name == end.name) {
            generatePath();
            return;
        }

        //поиск минимального непросмотренного ребра
        for (edgeIt = graph[current.end].begin(); edgeIt <
graph[current.end].end(); edgeIt++) {
            if (edgeIt->end.color == Colors::Black) {

```

```

        edgeIt->end.color = Colors::White;
        current.start = current.end;
        current.end = edgeIt->end;
        path[current.end.name] = current.start.name;
        break;
    }
}

//если ребро не было найдено, то делаем шаг назад
if (edgeIt == graph[current.end].end()) {
    current.end.name = path[current.end.name];
}

printGreedIteration(current);
}
}

void Graph::aStarPath() {

    printAStarMessage();

    std::map<char, Colors> visitedVerts;
    Edge *current;

    //создаем карту с непросмотренными вершинами
    for (graphIt = graph.begin(); graphIt != graph.end(); graphIt++) {
        visitedVerts[graphIt->first.name] = Colors::Black;
    }
    visitedVerts[start.name] = Colors::White;

    //заполняем очередь с приоритетами ребрами, исходящими из первой
    //вершины
    for (auto &edge : graph[start]) {
        edge.end.heuristic = heuristics[edge.end.name];
        edge.end.pathLength = edge.weight;
        edge.end.evaluateCost();
        edgeQueue.push(&edge);
    }

    while (!edgeQueue.empty()) {
        //снимаем верхнее ребро из очереди
        current = edgeQueue.top();
        printAStarIteration(edgeQueue, *current);
    }
}

```

```

        edgeQueue.pop();
        //если уже побывали в точке, в которую ведет ребро, то ничего
не делаем
        if (visitedVerts[current->end.name] == Colors::White) {
            continue;
        }

        visitedVerts[current->end.name] = Colors::White;
        path[current->end.name] = current->start.name;
        //условие выхода
        if (current->end.name == end.name) {
            generatePath();
            return;
        }
        //добавляем в очередь все ребра, ведущие в непросмотренные
вершины и исходящие из вершины,
        //в которую ведет текущее ребро
        for (auto &edge : graph[current->end.name]) {
            if (visitedVerts[edge.end.name] == Colors::Black) {
                edge.end.heuristic = heuristics[edge.end.name];
                edge.end.pathLength = current->end.pathLength +
edge.weigth;

                edge.end.evaluateCost();
                edgeQueue.push(&edge);
            }
        }
    }
}

int main() {
    Graph graph;
    graph.greedPath();
    graph.aStarPath();
    return 0;
}

```