

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ
по лабораторной работе
№2
по дисциплине «Построение и анализ алгоритмов»
Тема: Алгоритмы на графах

Студентка гр. 8304

Сани З. Б.

Преподаватель

Размочаева Н. В.

Санкт-Петербург

2020

Вариант 3.

Цель работы.

Научиться применять жадный алгоритм и алгоритм A* поиска пути в графе и оценивать их сложность.

Постановка задачи.

Вариант 3. Написать функцию, проверяющую эвристику на допустимость и монотонность.

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.

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.

Описание алгоритма.

Стартовая вершина помечается «открытой». Пока существуют открытые вершины:

- 1) Текущая вершина – открытая вершина, с наименьшей полной стоимостью.
- 2) Если текущая вершина конечная – алгоритм заканчивает работу.
- 3) Текущая вершина становится закрытой.
- 4) Для каждого незакрытого ребенка текущей вершины:
 - Рассчитывается функция пути для этой вершины.
 - Если вершина еще не открыта или рассчитанная функция меньше функции, рассчитанной для этой вершины ранее, рассчитанная функция становится функцией этой вершины, вершина-предок запоминается, как вершина, из которой совершен переход в ребенка. (необходимо для восстановления пути в будущем)

Если открытых вершин не осталось, а до конечной маршрут так и не был проложен, алгоритм заканчивает работу (пути не существует).

Анализ алгоритмов.

Временная сложность алгоритмов: $O(E + V^2) = O(V^2)$, где V – кол-во вершин, E – кол-во ребер. Обосновывается оценка тем, что в худшем случае будет совершен обход всех вершин и всех ребер. временная сложность в худшем случае экспоненциальная, в лучшем полиномиальная.

Сложность по памяти экспоненциальная.

Описание основных структур данных и функций.

Структуры.

Граф хранится в словаре `std::map<char, VertexInfo>`, где `VertexInfo` это структура, хранящая информацию о соседях и значение функции G .

```
struct VertexInfo {  
    std::vector<std::pair<char, int>> neighbors;  
    int length;  
};
```

Функции из описания алгоритма:

$$f(v)=g(v)+h(v)$$

- $g(v)$ — наименьшая стоимость пути в V из стартовой вершины,
- $h(v)$ — эвристическое приближение стоимости пути от V до конечной цели.

```
int G(char from, std::map<char, VertexInfo > & graphDict);  
int H(char from, char endVertex);  
int F(char from, char endVertex, std::map<char, VertexInfo >& graphDict);
```

Функция ищет для какой из открытых вершин будет минимальным путь.

```
char minVertex(std::set<char> & open, char endVertex, std::map<char, VertexInfo >&  
graphDict);
```

Функция возвращает понятный для пользователя путь в графе.

```
std::string constructPath (std::map<char, char> & from, char startVertex, char  
endVertex);
```

Функция ищущая алгоритмом А* минимальный путь.

```
void astarAlgo(char startVertex, char endVertex, std::map<char, VertexInfo >& my_  
graphDict);
```

monotone - функция используется для проверки данной эвристики на
МОНОТОННОСТЬ.

```
void monotone(std::map<char, VertexInfo > vertices, char endVertex);
```

Тестирование.

Результаты тестирования представлены в таблице 1.

Таблица 1 – Результаты тестирования

Ввод	Вывод
<pre>a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0</pre>	ade

a b a e 0 a c 0 e b 0	aeb
a b a d 3.0 a c 2.1 d b 1.0	adb

Вывод.

В ходе работы был построен и анализирован алгоритм A* на основе решения задачи о нахождении минимального пути в графе. Исходный код программы представлен в приложении А.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД

```
1 #include <iostream>
2 #include <map>
3 #include <vector>
4 #include <algorithm>
5 #include <set>
6
7 struct VertexInfo {
8     std::vector<std::pair<char, int>> neighbors;
9     int length;
10 };
11
12 bool cmp(const std::pair<char,int> &a, const std::pair<char,int> &b) {
13     if(a.second == b.second) return a.first > b.first;
14     return a.second < b.second;
15 }
16
17 int G(char from, std::map<char, VertexInfo> &graphDict) {
18     return graphDict[from].length;
19 }
20
21 int H(char from, char endvertex) {
22     return abs(from - endvertex);
23 }
24
25
26 int F(char from, char endvertex, std::map<char, VertexInfo> &graphDict){
27     return G(from, graphDict) + H(from, endvertex);
28 }
29
30 char minVertex(std::set<char> &open, char endvertex, std::map<char, VertexInfo> &graphDict) {
31     int min = INT_MAX;
32     char curr = 0;
33     for(char i: open)
34         if(F(i, endvertex, graphDict) < min){
35             min = F(i, endvertex, graphDict);
36             curr = i;
37         }
38     return curr;
39 }
40
41 std::string constructPath(std::map<char,char> &from,char startVertex, char endVertex) {
42     std::string path = {endVertex};
43     char curr = endVertex;
44     while (curr != startVertex) {
45         curr = from[curr];
46         path += curr;
47     }
48     std::reverse(path.begin(), path.end());
49     return path;
50 }
51
52 void astarAlgo(char startVertex, char endVertex, std::map<char, VertexInfo> &graphDict) {
53     char currentVertex = startVertex;
54     std::set<char> closed;
55     std::set<char> open = {startVertex};
56     std::map<char,char> from;
57
58     while(!open.empty()) {
59         currentVertex = minVertex(open,endVertex,graphDict);
60
61         if(currentVertex == endVertex){//endGame
62             std::cout << constructPath(from, startVertex, endVertex) << std::endl;
63             return;
64         }
65         open.erase(currentVertex);
66         closed.insert(currentVertex);
67
68         for(auto neighbour : graphDict[currentVertex].neighbors) {
69             bool tentative_is_better;
70             if(closed.find(neighbour.first) != closed.end())
71                 continue;
72             int tentative_g_score = G(currentVertex, graphDict) + neighbour.second;
73             if(open.find(neighbour.first) == open.end()){
74                 open.insert(neighbour.first);
75                 tentative_is_better = true;
76             }
77             else {
78                 tentative_is_better = tentative_g_score < G(neighbour.first, graphDict);
79             }
80
81             if(tentative_is_better) {
82                 from[neighbour.first] = currentVertex;
83                 graphDict[neighbour.first].length = tentative_g_score;
84             }
85         }
86     }
87     std::cout << "No way" << std::endl;
88 }
```

```

89 bool monotone(std::map<char, VertexInfo> vertices, char endVertex){
90     for(auto vertex : vertices){
91         char vertice1 = vertex.first;
92         for (int i = 0 ; i < vertex.second.neighbors.size(); ++i) {
93             char vertice2 = vertex.second.neighbors[i].first;
94             double weight = vertex.second.neighbors[i].second;
95             if(abs(endVertex-vertice1) - abs(endVertex-vertice2) > weight){
96                 return false;
97             }
98         }
99     }
100     return true;
101 }
102
103 int main() {
104     char startVertex, endVertex;
105
106     std::cin >> startVertex >> endVertex;
107     std::map<char, VertexInfo> graphDict;
108
109     char from, to;
110     float length;
111     while(std::cin >> from >> to >> length) {
112         if(length == -1)
113             break;
114         graphDict[from].neighbors.push_back({to,length});
115         std::sort(graphDict[from].neighbors.begin(),graphDict[from].neighbors.end(), cmp);
116     }
117
118     auto time = clock();
119     std::cout << "_____AStar Algorithm with Monotone function_____ " << std::endl;
120     std::cout << "Optima path : ";
121     astarAlgo(startVertex, endVertex, graphDict);
122     if(monotone(graphDict, endVertex)){
123         std::cout<< "The heuristic function of the graph is Monotone\n";
124     }else{
125         std::cout << "Not Monotone\n";
126     }
127
128     std::cout << " _____ " << std::endl;
129     std::cout << "Time taken : " << (double)(clock() - time) / CLOCKS_PER_SEC << std::endl;
130     std::cout << "Algorithm complexity G(V,E) :  $O(|V| * |V| + E) \sim O(V * V)$ " << std::endl;
131     std::cout << "where V - Vertices and E - edges" << std::endl;
132     return 0;
133 }
134
135

```


