

# EXPERIMENT 5 18CSC305J

**Name: Rahul Goel**

**Registration number: RA1911030010094**

**Date: 11.02.2022**

**AIM: To implement Best First Algorithm and A\* Algorithm using python.**

## ***BEST FIRST SEARCH***

### **Description:**

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore.

### **Algorithm:**

- Define a list, OPEN, consisting solely of a single node, the start node, s.
- IF the list is empty, return failure.

- Remove from the list the node  $n$  with the best score (the node where  $f$  is the minimum), and move it to a list, CLOSED.
- Expand node  $n$ .
- IF any successor to  $n$  is the goal node, return success and the solution (by tracing the path from the goal node to  $s$ ).
- FOR each successor node: 1.apply the evaluation function,  $f$ , to the node.  
2. IF the node has not been in either list, add it to OPEN.
- looping structure by sending the algorithm back to the second step.

**Code:**

```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]
def best_first_search(source, target, n):
    visited = [0] * n
    visited[0] = True
```

---

```
pq = PriorityQueue()
pq.put((0, source))
while pq.empty() == False:
```

```
u = pq.get()[1] print(u, end=" ") if u == target:  
break
```

```
for v, c in graph[u]:  
if visited[v] == False:
```

```
visited[v] = True
```

```
pq.put((c, v)) print()
```

```
def addedge(x, y, cost): graph[x].append((y,  
cost)) graph[y].append((x, cost))
```

```
addege(0, 1, 3) addege(0, 2, 6) addege(0,  
3, 5) addege(1, 4, 9) addege(1, 5, 8)
```

```
addege(2, 6, 12) addege(2, 7, 14)
```

```
addege(3, 8, 7) addege(8, 9, 5) addege(8,  
10, 6) addege(9, 11, 1) addege(9, 12, 10)
```

```
addege(9, 13, 2)
```

```
source = 0
```

```
target = 9 best_first_search(source, target, v)
```

**Output:**

## ***A\* Best First Search***

**Description:**

```
1 from queue import PriorityQueue
2 v = 14
3 graph = [[] for i in range(v)]
4
5 def best_first_search(source, target, n):
6     visited = [0] * n
7     visited[0] = True
8     pq = PriorityQueue()
9     pq.put((0, source))
10    while pq.empty() == False:
11        u = pq.get()[1]
12        print(u, end=" ")
13        if u == target:
14            break
15
16        for v, c in graph[u]:
17            if visited[v] == False:
18                visited[v] = True
19                pq.put((c, v))
20    print()
21
22 def addedge(x, y, cost):
23     graph[x].append((y, cost))
24     graph[y].append((x, cost))
25
26 addedge(0, 1, 3)
27 addedge(0, 2, 6)
28 addedge(0, 3, 5)
29 addedge(1, 4, 9)
30 addedge(1, 5, 8)
31 addedge(2, 6, 12)
32 addedge(2, 7, 14)
33 addedge(3, 8, 7)
34 addedge(8, 9, 5)
35 addedge(8, 10, 6)
36 addedge(9, 11, 1)
37 addedge(9, 12, 10)
```

```
python3: can't open file '/home/ubuntu/environment/094/exp5_bfs.py': [Errno 2] No such file or directory
Process exited with code: 0
```

A\* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

## Code:

```
def aStarAlgo(start_node, stop_node):
```

```
    open_set = set(start_node)
```

```
    closed_set = set()
```

```
    g = {} #store distance from starting node
```

parents = {}# parents contains an adjacency map of all

nodes

#distance of starting node from itself is zero

g[start\_node] = 0

#start\_node is root node i.e it has no parent nodes #so start\_node is set to its own parent node parents[start\_node] = start\_node

while len(open\_set) > 0: n = None

#node with lowest f() is found for v in open\_set:

if n == None or g[v] + heuristic(v) < g[n] + heuristic(n): n= v

---

first

if n == stop\_node or Graph\_nodes[n] == None:  
pass

else:

for (m, weight) in get\_neighbors(n):

#nodes 'm' not in first and last set are added to

#n is set its parent

if m not in open\_set and m not in closed\_set:

```
open_set.add(m) parents[m] = n
g[m] = g[n] + weight
```

```
#for each node m,compare its distance from
start i.e g(m) to the
```

```
#from start through n node else:
```

```
if g[m] > g[n] + weight: #update g(m)
g[m] = g[n] + weight #change parent of m to n
parents[m] = n
```

```
#if m in closed set,remove and add to open if m
in closed_set:
```

```
closed_set.remove(m) open_set.add(m)
```

```
if n == None:
```

```
print('Path does not exist!') return None
```

```
# if the current node is the stop_node
```

```
# then we begin reconstructin the path from it to
the start_node
```

```
if n == stop_node:
```

```
path = []
```

```
while parents[n] != n: path.append(n)
```

```
n = parents[n]
```

```
path.append(start_node) path.reverse()
```

```
print('Path found: {}'.format(path)) return path
# remove n from the open_list, and add it to
closed_list # because all of his neighbors were
inspected open_set.remove(n)
closed_set.add(n)

print('Path does not exist!') return None

#define fuction to return neighbor and its
distance #from the passed node
def get_neighbors(v):
if v in Graph_nodes: return Graph_nodes[v]
else:
return None

#for simplicity we ll consider heuristic distances
given #and this function returns heuristic
distance for all nodes def heuristic(n):
H_dist = { 'A': 11,
'B': 6, 'C': 99,
'D': 1, 'E': 7, 'G': 0,
}
return H_dist[n]

#Describe your graph here Graph_nodes = {
```

```

'A': [('B', 2), ('E', 3)], 'B': [('C', 1), ('G', 9)], 'C':
None,
'E': [('D', 6)],
'D': [('G', 1)],
}
aStarAlgo('A', 'G')

```

**Output:**

The screenshot shows the AWS Cloud9 IDE interface. The main editor displays a Python script named `exp_5.py` with the following code:

```

87 #and this function returns heuristic distance for all nodes
88 def heuristic(n):
89     H_dist = {
90         'A': 11,
91         'B': 6,
92         'C': 99,
93         'D': 1,
94         'E': 7,
95         'G': 0,
96     }
97
98     return H_dist[n]
99
100 #Describe your graph here
101 Graph_nodes = {
102     'A': [('B', 2), ('E', 3)],
103     'B': [('C', 1), ('G', 9)],
104     'C': None,
105     'E': [('D', 6)],
106     'D': [('G', 1)],
107 }
108
109 aStarAlgo('A', 'G')
110

```

Below the editor, the output of the script is shown in the console window:

```

Path found: ['A', 'E', 'D', 'G']
Process exited with code: 0

```

The right sidebar shows the 'ENVIRONMENT MEMBERS' section with a list of ReadWrite members and a 'GROUP CHAT' section.

**Result: Best first and A\* algorithm were successfully executed in python.**