

# Artificial Intelligence

## Week 5

### Topic:

**Implementation of Best First Search**

**AND A\* Search algorithms**

## **Aim:**

To implement BFS and A-star algorithm.

## **Problem Statement:**

Developing Best first search and A\* Algorithm for real world problems

## **Best First Search:**

Best first search is a traversal technique that decides which node is to be visited next by checking which node is the most promising one and then check it. For this it uses an evaluation function to decide the traversal.

This best first search technique of tree traversal comes under the category of heuristic search or informed search technique.

The cost of nodes is stored in a priority queue. This makes implementation of best-first search is same as that of breadth First search. We will use the priorityqueue just like we use a queue for BFS.

## **A\* ALGORITHM:**

It is a searching algorithm that is used to find the shortest path between an initial and a final point.

It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A\* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It still remains a widely popular algorithm for graph traversal.

It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

## Algorithms :

### Steps For Best First Search Algorithm:

1. Create 2 empty lists: OPEN and CLOSED
2. Start from the initial node (say N) and put it in the 'ordered' OPEN list
3. Repeat the next steps until GOAL node is reached

If OPEN list is empty, then EXIT the loop returning 'False'

1. Select the first/top node (say N) in the OPEN list and move it to the CLOSED list. Also capture the information of the parent node
2. If N is a GOAL node, then move the node to the Closed list and exit the loop returning 'True'. The solution can be found by backtracking the path
3. If N is not the GOAL node, expand node N to generate the 'immediate' next nodes linked to node N and add all those to the OPEN list
4. Reorder the nodes in the OPEN list in ascending order according to an evaluation function  $f(n)$

### Steps for A\* Search Algorithm :

- Make an open list containing starting node
  - If it reaches the destination node :
  - Make a closed empty list
  - If it does not reach the destination node, then consider a node with the lowest f-score in the open list

We are finished

- Else :

Put the current node in the list and check its neighbors

- For each neighbor of the current node :
  - If the neighbor has a lower g value than the current node and is in the closed list:

Replace neighbor with this new node as the neighbor's parent

- Else If (current g is lower and neighbor is in the open list):

Replace neighbor with the lower g value and change the neighbor's parent to the current node.

- Else If the neighbor is not in both lists:

Add it to the open list and set its g

Code :

Best First Search :

```
from queue import PriorityQueue
v = 5
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, 5)
addege(0, 2, 1)
addege(2, 3, 2)
addege(1, 4, 1)
addege(3, 4, 2)
source = 0
target = 4
best_first_search(source, target, v)
```

## A\* Search :

```
def aStarAlgo(start_node, stop_node):
    print("\nA* Algorithm \n")

    open_set = set(start_node)
    closed_set = set()
    g = {}
    parents = {}

    g[start_node] = 0

    parents[start_node] = start_node

    while len(open_set) > 0:
        n = None

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

        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)

        if n == None:
            print('Path does not exist!')
            return None

        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
```

```

        open_set.remove(n)
        closed_set.add(n)

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

def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 99,
        'D': 1,
        'E': 7,
        'G': 0,

    }

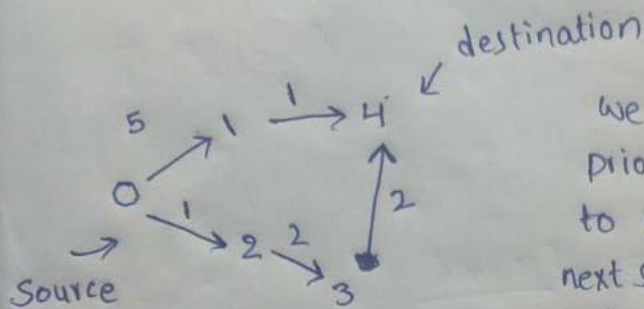
    return H_dist[n]

Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1), ('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],

}
aStarAlgo('A', 'G')
```

## Manual Calculation :

### Best first Search:



We use priority Queue to know what is the next smallest distance node.

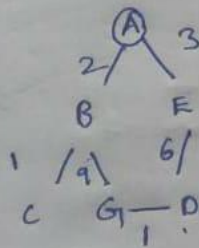
$\therefore 0 \rightarrow 1$  having  $5 > 0 \rightarrow 2$

We choose

$0 \rightarrow 2 \xrightarrow{2} 3 \xrightarrow{2} 4$  ← we reach destination so we stop.  
 ( $\because$  we have only one path)

### A\* Search: $f(n) = g(n) + h(n)$

| node | $h(n)$ |
|------|--------|
| A    | 11     |
| B    | 6      |
| C    | 99     |
| D    | 1      |
| E    | 7      |
| G    | 0      |



$$A \rightarrow B = 2 + 6 = 8 \quad (\text{we choose this})$$

$$A \rightarrow E = 3 + 6 = 9$$

$$A \rightarrow B \rightarrow C = 3 + 99 = 102$$

$$A \rightarrow B \rightarrow G = 2 + 9 + 0 = 11$$

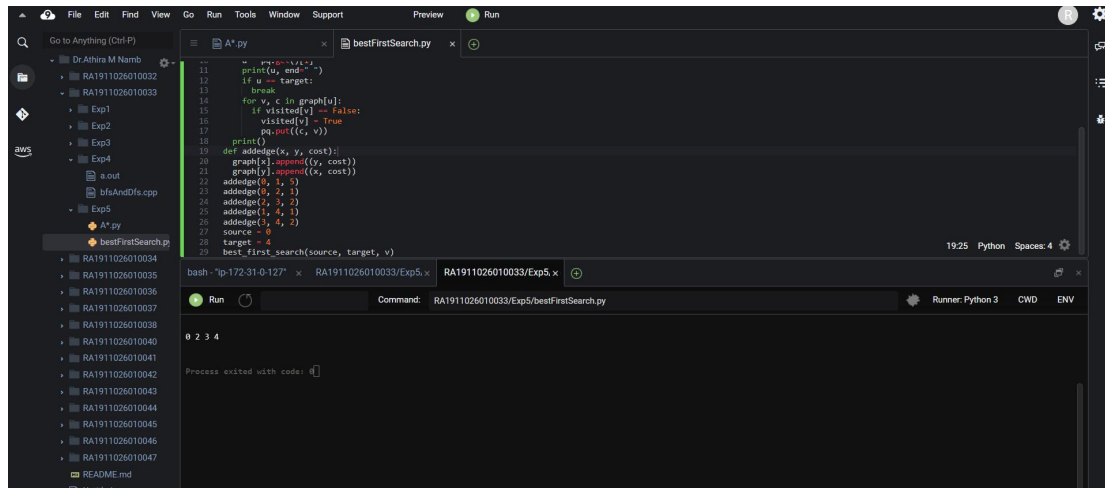
$$A \rightarrow E \rightarrow D = 3 + 6 + 1 = 10$$

$A \rightarrow E \rightarrow D \rightarrow G$  is the best path

$$A \rightarrow E \rightarrow D \rightarrow G = 3 + 6 + 1 + 0 = 10$$

Output:

Best First Search :

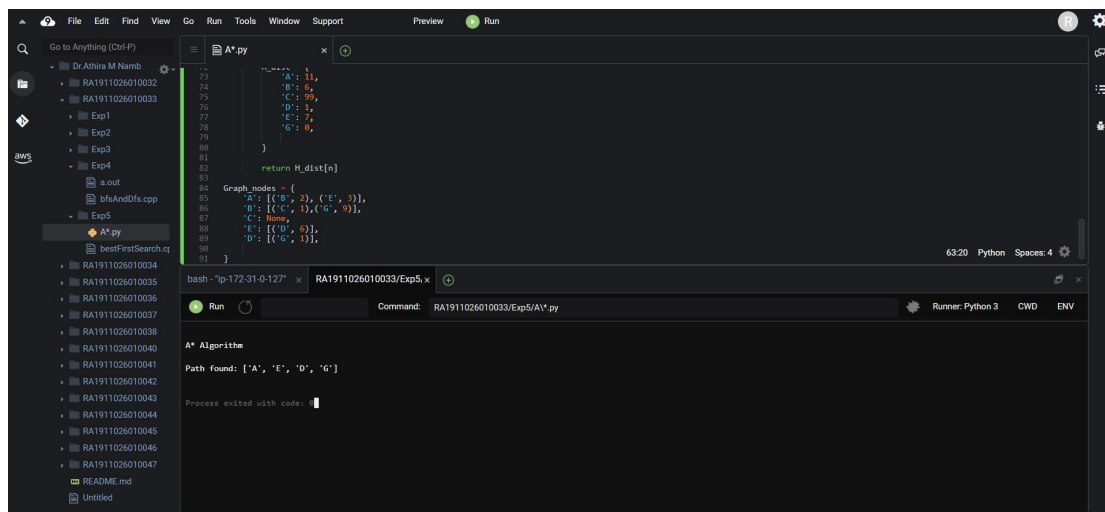


```
11 print(u, end=" ")
12 if u == target:
13     break
14 for v, c in graph[u]:
15     if visited[v] == False:
16         visited[v] = True
17         pq.put((c, v))
18 print()
19 def addedge(x, y, cost):
20     graph[x].append((y, cost))
21     graph[y].append((x, cost))
22 addedge(0, 1, 5)
23 addedge(0, 2, 3)
24 addedge(2, 3, 2)
25 addedge(1, 4, 3)
26 addedge(3, 4, 2)
27 source = 0
28 target = 4
29 best_first_search(source, target, v)
```

0 2 3 4

Process exited with code: 0

A\* Search :



```
73 A: 11
74 B: 0
75 C: 99
76 D: 3
77 E: 7
78 G: 0
79
80
81
82 return H_dist[n]
83
84
85 Graph_nodes = {
86     'A': [('B', 2), ('E', 3)],
87     'B': [('C', 1), ('G', 9)],
88     'C': None,
89     'E': [('D', 6)],
90     'D': [('G', 1)],
91 }
```

A\* Algorithm

Path found: ['A', 'E', 'D', 'G']

Process exited with code: 0



**Observation :**

Best First search time complexity is  $O(n \log n)$  A\* search function works on

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

Here  $g(n)$  is the actual distance between the start point and any vertex

$H(n)$  is the heuristic value for the vertex.

Now if  $g(n) = 0$  then the A\* search becomes best first search

**Result :**

Implementation of Best first Search and A\* Search is successfully implemented.