ABHINAV RANJAN
RA1911003010003
CSE A1 SECTION
SRMIST , KTR

# AI LAB EXP 5 - BEST FIRST SEARCH AND A* ALGORITHM

## AIM :

To implement best first search and A* algorithm in AI.

## REQUIREMENTS :

1. Knowledge of the concepts of best first search algorithm
2. Knowledge of the concepts of A* algorithm
3. AWS to execute code

## ALGORITHM:

### BEST FIRST SEARCH

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

a. If OPEN list is empty, then EXIT the loop returning ‘False’
b. 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

c. 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
d. 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
e. Reorder the nodes in the OPEN list in ascending order according to an evaluation function f(n)

## A* ALGORITHM

1. Initialise the open list
2. Initialise the closed list
   put the starting node on the open
   list (you can leave its f at zero)
3. while the open list is not empty
   a) find the node with the least f on
      the open list, call it "q"
   b) pop q off the open list
   c) generate q's 8 successors and set their
      parents to q

   d) for each successor
      i) if successor is the goal, stop search

      ii) else, compute both g and h for successor
        successor.g = q.g + distance between
                      successor and q
        successor.h = distance from goal to
        successor (This can be done using many
        ways, we will discuss three heuristics-
        Manhattan, Diagonal and Euclidean
        Heuristics)

        successor.f = successor.g + successor.h

iii) if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor

iV) if a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip this successor otherwise, add the node to the open list
end (for loop)

e) push q on the closed list
end (while loop)

## SOURCE CODE:

### 1. BEST FIRST SEARCH

```
from collections import defaultdict
from queue import PriorityQueue

class Graph:

    def __init__(self):
        self.graph = defaultdict(list)


    def addEdge(self,u,v):
        self.graph[u].append([v])

    def BFS(self, start, goal, hn):
        q = PriorityQueue()
        path = {}
        visited = []
        q.put((hn[start], start, start))

        while q:
```

```
            top = q.get()
            visited.append(top[1])
            if top[1] not in path:
                path[top[1]] = top[2]

            if top[1] == goal:
                res = []
                i = goal
                while i != start:
                    res.append(i)
                    i = path[i]
                res.append(start)
                res.reverse()
                return res

            for node in self.graph[top[1]]:
                if node not in visited:
                    q.put((hn[node[0]], node[0], top[1]))
g = Graph()
print ("Best First Search: ")
g.addEdge("Home", "School");
g.addEdge("Home", "Garden");
g.addEdge("Home", "Bank");
g.addEdge("School", "Post_Office");
g.addEdge("School", "Railway_Station");
g.addEdge("Railway_Station", "University");
g.addEdge("Bank", "Police_Station");
g.addEdge("Police_Station", "University");
g.addEdge("Garden", "Railway_Station");

hn = {'Home': 0 , 'School': 50, 'Post_Office': 59, 'Railway_Station':
75, 'Garden': 40, 'Bank': 45,
            'Police_Station': 60, 'University': 28}

start = 'Home'
end = 'University'
```

```
path = g.BFS(start, end, hn)
print("Shortest path of traversal => ", path)
```

## 2. A*

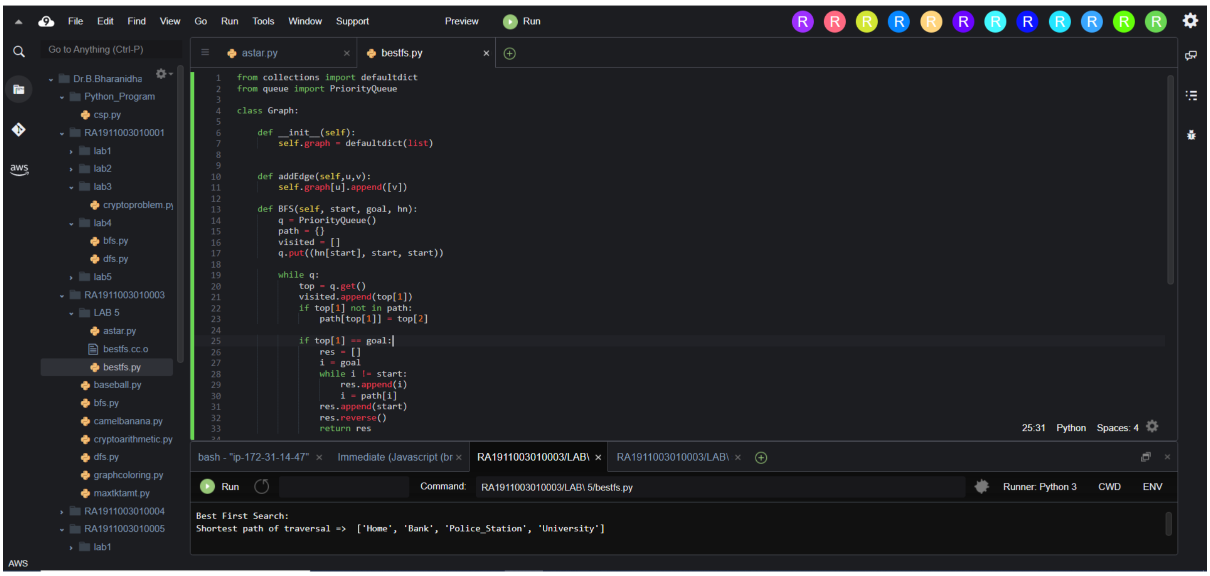
```
class Graph:
    def __init__(self, graph_dict=None, directed=True):
        self.graph_dict = graph_dict or {}
        self.directed = directed
        if not directed:
            self.make_undirected()
    def make_undirected(self):
        for a in list(self.graph_dict.keys()):
            for (b, dist) in self.graph_dict[a].items():
                self.graph_dict.setdefault(b, {})[a] = dist
    def connect(self, A, B, distance=1):
        self.graph_dict.setdefault(A, {})[B] = distance
        if not self.directed:
            self.graph_dict.setdefault(B, {})[A] = distance
    def get(self, a, b=None):
        links = self.graph_dict.setdefault(a, {})
        if b is None:
            return links
        else:
            return links.get(b)
    def nodes(self):
        s1 = set([k for k in self.graph_dict.keys()])
        s2 = set([k2 for v in self.graph_dict.values() for k2, v2 in v.items()])
        nodes = s1.union(s2)
        return list(nodes)
class Node:

    def __init__(self, name:str, parent:str):
        self.name = name
        self.parent = parent
        self.g = 0
```

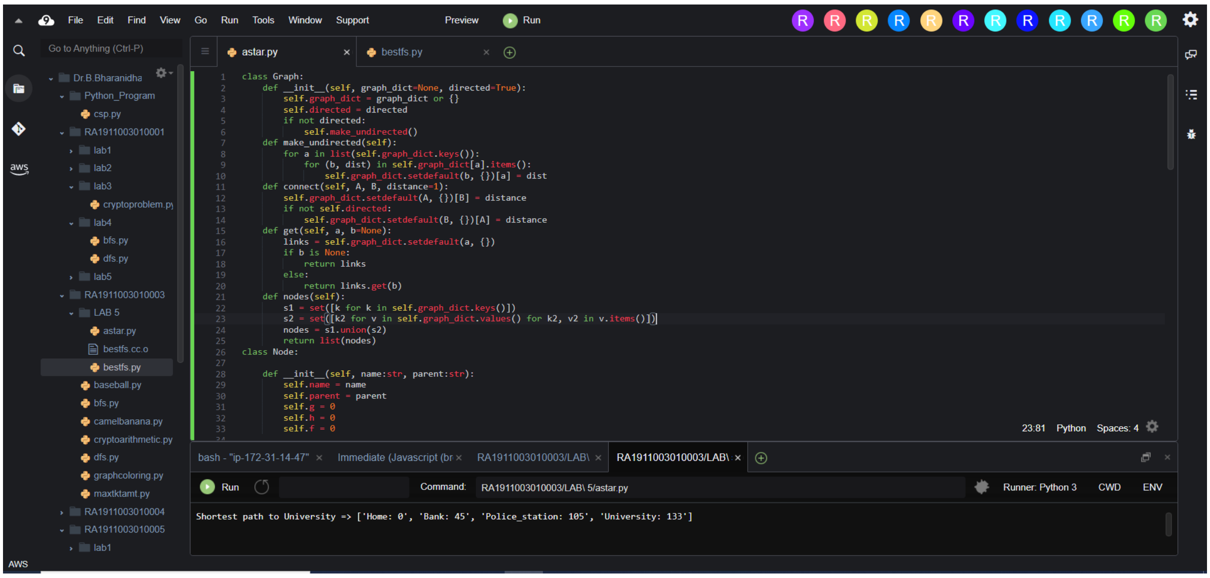
```
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.name == other.name

    def __lt__(self, other):
         return self.f < other.f

    def __repr__(self):
        return ('({0},{1})'.format(self.name, self.f))

def astar_search(graph, heuristics, start, end):


    open = []
    closed = []

    start_node = Node(start, None)
    goal_node = Node(end, None)

    open.append(start_node)


    while len(open) > 0:

        open.sort()

        current_node = open.pop(0)

        closed.append(current_node)
        if current_node == goal_node:
            path = []
            while current_node != start_node:
                path.append(current_node.name + ': ' + str(current_node.g))
                current_node = current_node.parent
            path.append(start_node.name + ': ' + str(start_node.g))
```

```
            return path[::-1]

        neighbors = graph.get(current_node.name)

        for key, value in neighbors.items():

            neighbor = Node(key, current_node)

            if(neighbor in closed):
                continue

            neighbor.g = current_node.g + graph.get(current_node.name,
neighbor.name)
            neighbor.h = heuristics.get(neighbor.name)
            neighbor.f = neighbor.g + neighbor.h

            if(add_to_open(open, neighbor) == True):

                open.append(neighbor)

    return None

def add_to_open(open, neighbor):
    for node in open:
        if (neighbor == node and neighbor.f > node.f):
            return False
    return True

def main():

    graph = Graph()

    graph.connect('Home', 'School', 50)
    graph.connect('School', 'Post_office', 59)
    graph.connect('School', 'Railway_station', 75)
    graph.connect('Railway_station', 'University', 40)
```

```
    graph.connect('Home', 'Garden', 40)
    graph.connect('Garden', 'Railway_station', 72)
    graph.connect('Home', 'Bank', 45)
    graph.connect('Bank', 'Police_station', 60)
    graph.connect('Police_station', 'University', 28)
    graph.make_undirected()

    heuristics = {}
    heuristics['Post_office'] = 109
    heuristics['School'] = 50
    heuristics['Railway_station'] = 112
    heuristics['Garden'] = 40
    heuristics['Police_station'] =105
    heuristics['Bank'] = 45
    heuristics['University'] = 133
    heuristics['Home'] = 0

    path = astar_search(graph, heuristics, 'Home', 'University')
    print("Shortest path to University =>", path)
    print()

if __name__ == "__main__": main()
```

## SCREENSHOT OF OUTPUTS:






# RESULT:

Thus we have successfully implemented best first search and A* algorithm in AI.