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))

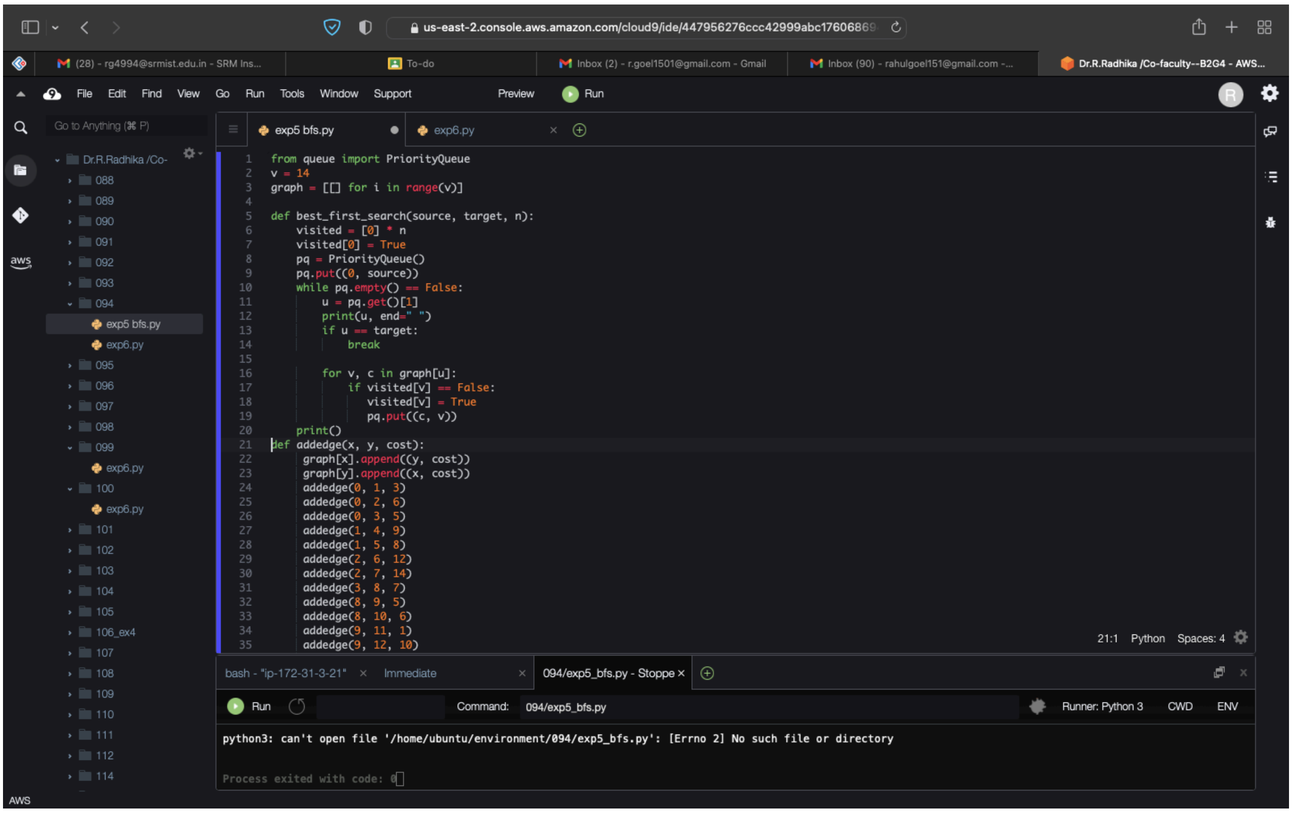
addedge(0, 1, 3) addedge(0, 2, 6) addedge(0, 3, 5) addedge(1, 4, 9) addedge(1, 5, 8) addedge(2, 6, 12) addedge(2, 7, 14) addedge(3, 8, 7) addedge(8, 9, 5) addedge(8, 10, 6) addedge(9, 11, 1) addedge(9, 12, 10) addedge(9, 13, 2)

source = 0
target = 9 best_first_search(source, target, v)

Output:

A Best First Search*

Description:



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

#ditance 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]

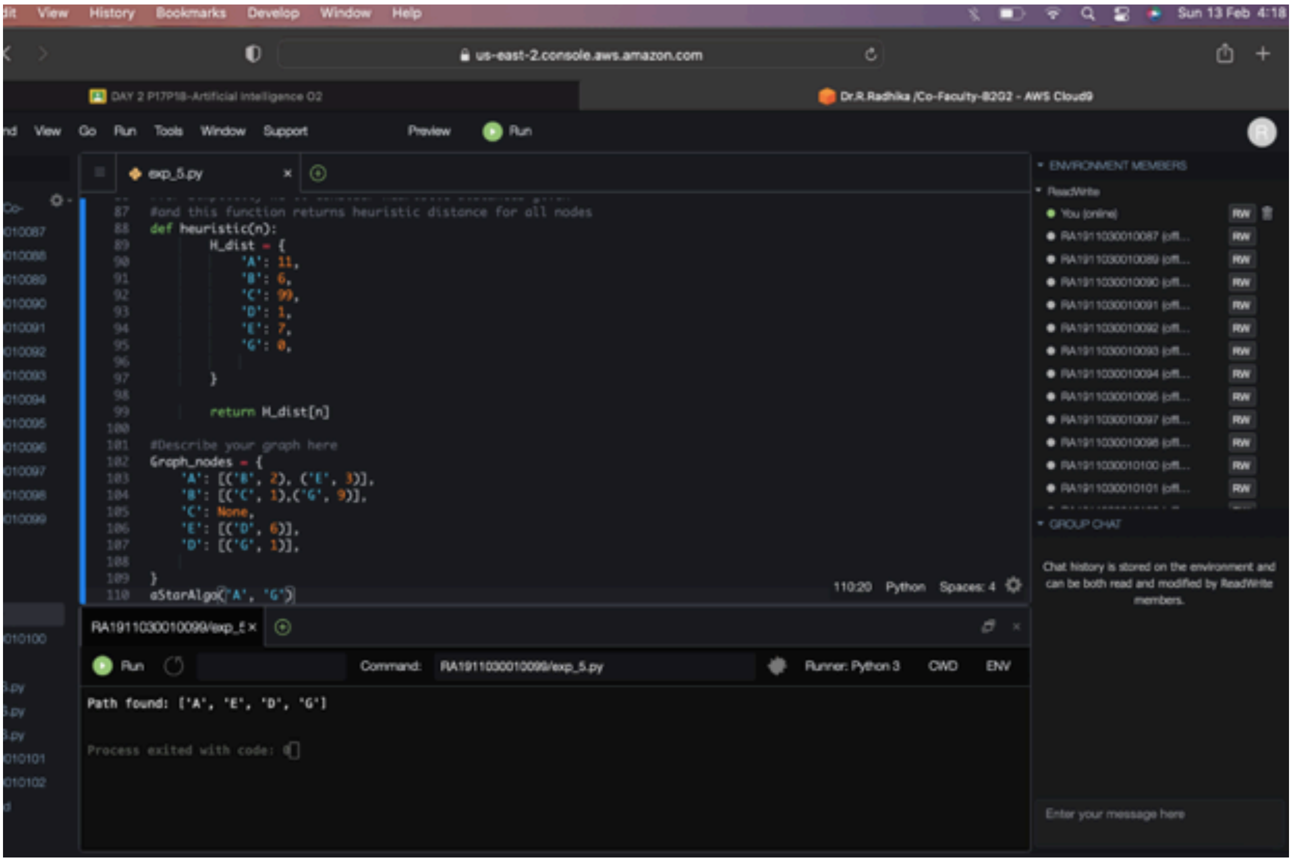
#Describe your graph here Graph_nodes = {

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

'D': [('G', 1)],

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

Output:



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