Manas Jha

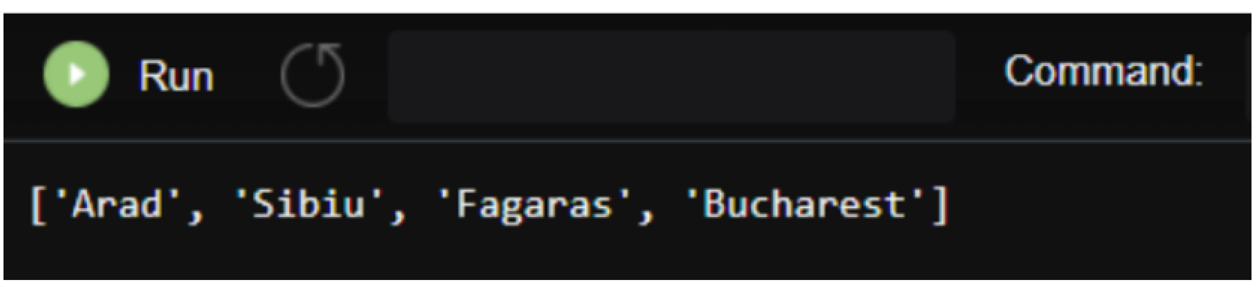
RA1911003010643

Aim- (A) Developing Best first search Algorithm for real world problems

```
from queue import Queue

romaniaMap = {
'Arad': ['Sibiu', 'Zerind', 'Timisoara'],
'Zerind': ['Arad', 'Oradea'],
'Oradea': ['Zerind', 'Sibiu'],
'Sibiu': ['Arad', 'Oradea', 'Fagaras', 'Rimnicu'],
'Timisoara': ['Arad', 'Lugoj'],
'Lugoj': ['Timisoara', 'Mehadia'],
'Mehadia': ['Lugoj', 'Drobeta'],
'Drobeta': ['Mehadia', 'Craiova'],
'Craiova': ['Drobeta', 'Rimnicu', 'Pitesti'],
'Rimnicu': ['Sibiu', 'Craiova', 'Pitesti'],
'Fagaras': ['Sibiu', 'Bucharest'],
'Pitesti': ['Rimnicu', 'Craiova', 'Bucharest'],
'Bucharest': ['Fagaras', 'Pitesti', 'Giurgiu', 'Urziceni'],
'Giurgiu': ['Bucharest'],
'Urziceni': ['Bucharest', 'Vaslui', 'Hirsova'],
'Hirsova': ['Urziceni', 'Eforie'],
'Eforie': ['Hirsova'],
'Vaslui': ['Iasi', 'Urziceni'],
'Iasi': ['Vaslui', 'Neamt'],
'Neamt': ['Iasi']
}
```

```
def bfs(startingNode, destinationNode):

# For keeping track of what we have visited

visited = {}

# keep track of distance

distance = {}

# parent node of specific graph

parent = {}


bfs_traversal_output = []

# BFS is queue based so using 'Queue' from python built-in

queue = Queue()


# travelling the cities in map

for city in romaniaMap.keys():

# since intially no city is visited so there will be nothing in visited list

visited[city] = False

parent[city] = None

distance[city] = -1


# starting from 'Arad'

startingCity = startingNode

visited[startingCity] = True

distance[startingCity] = 0

queue.put(startingCity)


while not queue.empty():

u = queue.get() # first element of the queue, here it will be 'arad'

bfs_traversal_output.append(u)
```

```
# explore the adjust cities adj to 'arad'
for v in romaniaMap[u]:
if not visited[v]:
visited[v] = True
parent[v] = u
distance[v] = distance[u] + 1
queue.put(v)

# reaching our destination city i.e 'bucharest'
g = destinationNode
path = []
while g is not None:
path.append(g)
g = parent[g]

path.reverse()
# printing the path to our destination city
print(path)
# print(distance)
# Starting City & Destination City
bfs('Arad', 'Bucharest')
```




```
# graph class
class Graph:
```

```
# init class
def __init__(self, graph_dict=None, directed=True):
    self.graph_dict = graph_dict or {}
    self.directed = directed
    if not directed:
        self.make_undirected()

# create undirected graph by adding symmetric edges
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

# add link from A and B of given distance, and also add the inverse link if the graph is undirected
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

# get neighbors or a neighbor
def get(self, a, b=None):
    links = self.graph_dict.setdefault(a, {})
    if b is None:
        return links
    else:
        return links.get(b)

# return list of nodes in the graph
```

```
    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)

# node class
class Node:

    # init class
    def __init__(self, name:str, parent:str):
        self.name = name
        self.parent = parent
        self.g = 0 # distance to start node
        self.h = 0 # distance to goal node
        self.f = 0 # total cost

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

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

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

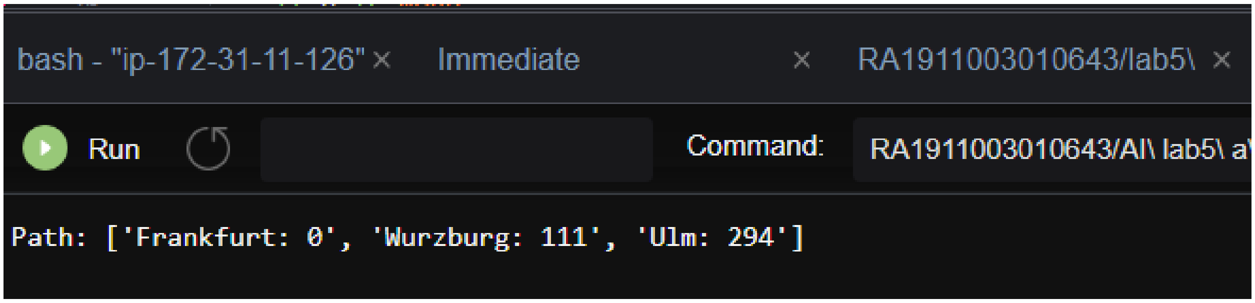
```
# A* search
def astar_search(graph, heuristics, start, end):

    # lists for open nodes and closed nodes
    open = []
    closed = []

    # a start node and an goal node
    start_node = Node(start, None)
    goal_node = Node(end, None)

    # add start node
    open.append(start_node)

    # loop until the open list is empty
    while len(open) > 0:


        open.sort()                          # sort open list to get the node with the lowest cost first
        current_node = open.pop(0)           # get node with the lowest cost
        closed.append(current_node)          # add current node to the closed list

        # check if we have reached the goal, return the path
        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)    # get neighbours

        # loop neighbors
        for key, value in neighbors.items():
            neighbor = Node(key, current_node)      # create neighbor node
            if(neighbor in closed):                 # check if the neighbor is in the closed list
                continue

            # calculate full path cost
            neighbor.g = current_node.g + graph.get(current_node.name, neighbor.name)
            neighbor.h = heuristics.get(neighbor.name)
            neighbor.f = neighbor.g + neighbor.h

            # check if neighbor is in open list and if it has a lower f value
            if(add_to_open(open, neighbor) == True):

                # everything is green, add neighbor to open list
                open.append(neighbor)

    # return None, no path is found
    return None

# check if a neighbor should be added to open list
def add_to_open(open, neighbor):
    for node in open:
        if (neighbor == node and neighbor.f > node.f):
```

```
            return False
    return True

# create a graph
graph = Graph() # user-based input for edges will be updated in the upcoming days
# create graph connections (Actual distance)
graph.connect('Frankfurt', 'Wurzburg', 111)
graph.connect('Frankfurt', 'Mannheim', 85)
graph.connect('Wurzburg', 'Nurnberg', 104)
graph.connect('Wurzburg', 'Stuttgart', 140)
graph.connect('Wurzburg', 'Ulm', 183)
graph.connect('Mannheim', 'Nurnberg', 230)
graph.connect('Mannheim', 'Karlsruhe', 67)
graph.connect('Karlsruhe', 'Basel', 191)
graph.connect('Karlsruhe', 'Stuttgart', 64)
graph.connect('Nurnberg', 'Ulm', 171)
graph.connect('Nurnberg', 'Munchen', 170)
graph.connect('Nurnberg', 'Passau', 220)
graph.connect('Stuttgart', 'Ulm', 107)
graph.connect('Basel', 'Bern', 91)
graph.connect('Basel', 'Zurich', 85)
graph.connect('Bern', 'Zurich', 120)
graph.connect('Zurich', 'Memmingen', 184)
graph.connect('Memmingen', 'Ulm', 55)
graph.connect('Memmingen', 'Munchen', 115)
graph.connect('Munchen', 'Ulm', 123)
graph.connect('Munchen', 'Passau', 189)
graph.connect('Munchen', 'Rosenheim', 59)
graph.connect('Rosenheim', 'Salzburg', 81)
```

```
graph.connect('Passau', 'Linz', 102)
graph.connect('Salzburg', 'Linz', 126)
# make graph undirected, create symmetric connections
graph.make_undirected()
# create heuristics (straight-line distance, air-travel distance)
heuristics = {}
heuristics['Basel'] = 204
heuristics['Bern'] = 247
heuristics['Frankfurt'] = 215
heuristics['Karlsruhe'] = 137
heuristics['Linz'] = 318
heuristics['Mannheim'] = 164
heuristics['Munchen'] = 120
heuristics['Memmingen'] = 47
heuristics['Nurnberg'] = 132
heuristics['Passau'] = 257
heuristics['Rosenheim'] = 168
heuristics['Stuttgart'] = 75
heuristics['Salzburg'] = 236
heuristics['Wurzburg'] = 153
heuristics['Zurich'] = 157
heuristics['Ulm'] = 0
# run the search algorithm
path = astar_search(graph, heuristics, 'Frankfurt', 'Ulm')
```

```
print("Path:", path)
```



Date

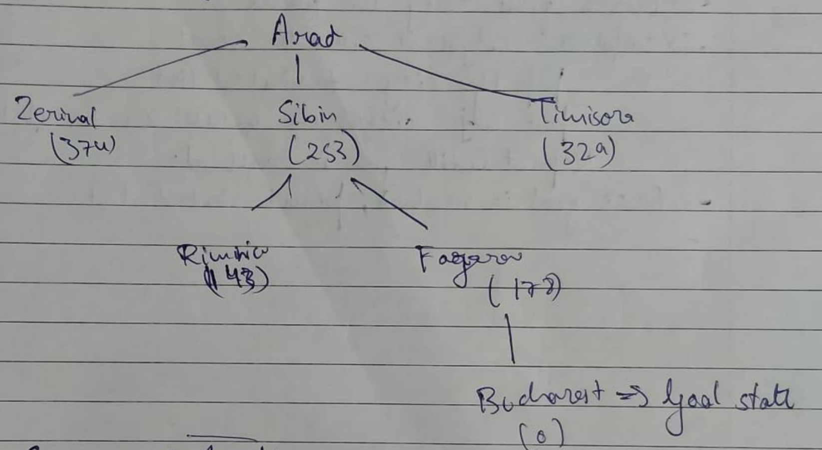
Algorithm:

→ create an empty queue.
→ create lists visited, distance and parent
→ Travel through the cities in map
→ Initially visited list is empty
→ Push first city in queue
→ explore adjacent cities to start
pop the city to visited list
put adj. cities in queue acc. to distance
Repeat until goal is reached.
→ Once goal is reached, print visited list.

Problem Formulation: Developing best first search and A* algorithm for real world problems.

Given a city map. Find the path from Arad to Bucharest using BFS and A* algorithm.

Problem Solving:



Queue → Arad

Remove A and put in visited set

Sibiu | Timisora | Zerind (Put in order)

Remove Sibiu put in visited set

Fagares | Rimnic | Timisora | Zerind

Remove Fagaras and put in visited set

Now Bucharest is a neighbour of Fagares, here we return.

Getting the path from visited set

∴ Path is Arad → Sibiu → Fagaras → Bucharest