Akshat Agarwal
RA1911003010646

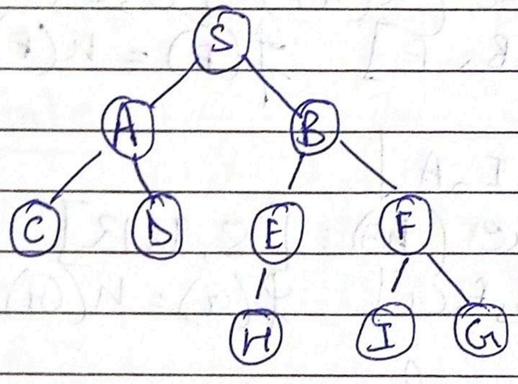
Artificial Intelligence
Lab 5

Aim :- To develop Best first Search & A* for real life problems.

1) BFS

Problem formulation :

Given a graph, Starting node and h(n), use the evaluation function to decide which is the most promising node for reaching to the dest^n and explore it till reaches the destination. Display the path & cost fn.



Node	h(n)
A	12
B	4
C	7
D	3
E	8
F	2
G	0
H	4
I	9
S	13

Initial State

Open : [S]

Closed : [ ]

Final State

Open : [I, E, A]

Closed : [S, B, F, G]

Path : S → B → F → G

Cost : 2+1+3+0
= 6

Date / /
Page

Problem Solving

- Open : [S]
  Priority Queue (h(n)) : [13]
  Closed : [ ]   f(S) = h(S) = 13

- Open : [B, A]
  Priority Queue (h(n)) : [4, 12]
  Closed : [S]   f(B) = h(B) = 4

- Open : [F, E, A]
  Priority Queue (h(n)) : [2, 8, 12]
  Closed : [S, B]

- Open : [G, F, I, A]
  Priority Queue (h(n)) : [0, 8, 9, 12]
  Closed : [S, B, F]   f(F) = h(F) = 2

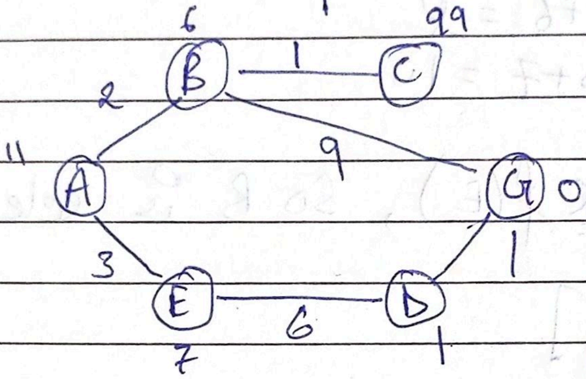
- Open : [E, I, A]
  Priority Queue (h(n)) : [8, 9, 12]
  Closed : [S, B, F, G]   f(G) = h(G) = 0

Goal State Reached

2] A Star

Problem Formulation:

Given a graph with the numbers written on edge representing the distance between the nodes while the numbers written on nodes representing heuristic values.
Find the most cost-effective path to reach from start A to final state G using A*



Initial State

→ A

Open : [A]
closed : [ ]

Final State

open : [ ]
Closed : [A, E, D, G]

Path: A → E → D → G
Cost: 3+6+1+0
= 10

↳ Since GoalState is reached
⇒ open : [ ]
Closed : [A, E, D, G]

Date / /
Page
STUDY BUDDIES

Problem Solving

- Open : [A]
  Closed : [ ]

  $g(A) = 0$
  $h(A) = 11$
  $f(A) = 11$

- A has two nodes B & E.
  $f(B) = 2+6 = 8$
  $f(E) = 3+7 = 10$

  $f(B) < f(E)$, so B is selected

- Open : [B, E]
  Closed : [A]

- B has two nodes C & G
  $f(C) = 2+1+99 = 102$
  $f(G) = 2+9+0 = 11$

  But $f(G) > f(E)$
  ∴ We explore from E

- Open : [E]
  Closed : [A]

- E has only one node D
  $f(D) = 3+6+1 = 10$
  open : [D]
  Closed : [A, E]

- D has only one node G
  $f(G) = 3+6+1+0 = 10$

  Open : [G]
  Closed : [A, E,

Akshat Agarwal
RA1911003010646

Artificial Intelligence
Lab - 5

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

BEST FIRST SEARCH

Code :

```
class Graph:
    # Initialize the 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 an 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 a 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 a 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)
# This class represent a node
class Node:
    # Initialize the 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.position, self.f))
# Best-first search
def best_first_search(graph, heuristics, start, end):

    # Create lists for open nodes and closed nodes
    open = []
    closed = []
    # Create a start node and an goal node
    start_node = Node(start, None)
    goal_node = Node(end, None)
    # Add the start node
    open.append(start_node)

    # Loop until the open list is empty
    while len(open) > 0:
        # Sort the open list to get the node with the lowest cost
first
        open.sort()
        # Get the node with the lowest cost
```

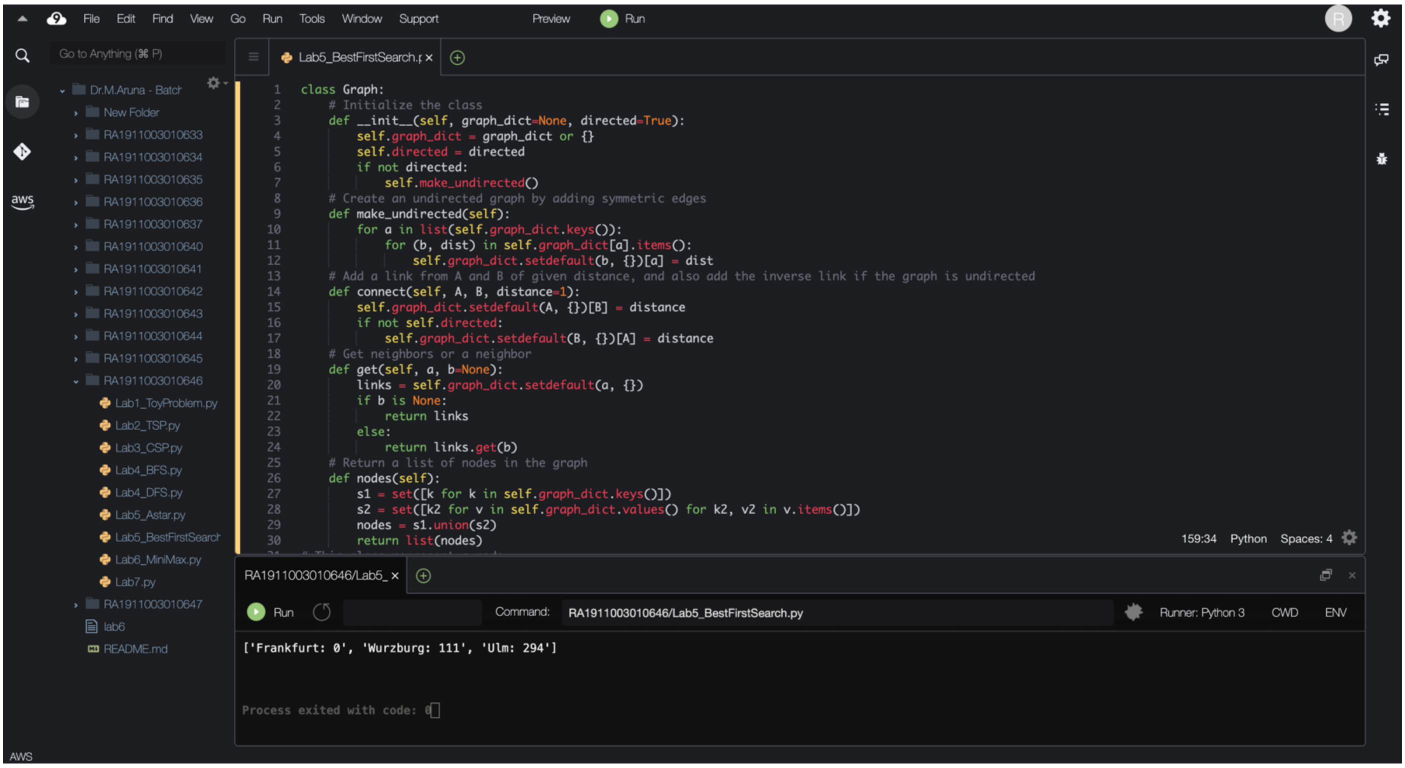
```
        current_node = open.pop(0)
        # Add the current node to the closed list
        closed.append(current_node)

        # 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 reversed path
            return path[::-1]
        # Get neighbours
        neighbors = graph.get(current_node.name)
        # Loop neighbors
        for key, value in neighbors.items():
            # Create a neighbor node
            neighbor = Node(key, current_node)
            # Check if the neighbor is in the closed list
            if(neighbor in closed):
                continue
            # Calculate cost to goal
            neighbor.g = current_node.g +
graph.get(current_node.name, neighbor.name)
            neighbor.h = heuristics.get(neighbor.name)
            neighbor.f = 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
# The main entry point for this module
def main():
    # Create a graph
    graph = Graph()
    # 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 search algorithm
    path = best_first_search(graph, heuristics, 'Frankfurt', 'Ulm')
    print(path)
    print()
# Tell python to run main method
if __name__ == "__main__": main()
```

Output :



A STAR SEARCH

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


            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 first
                    #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 :