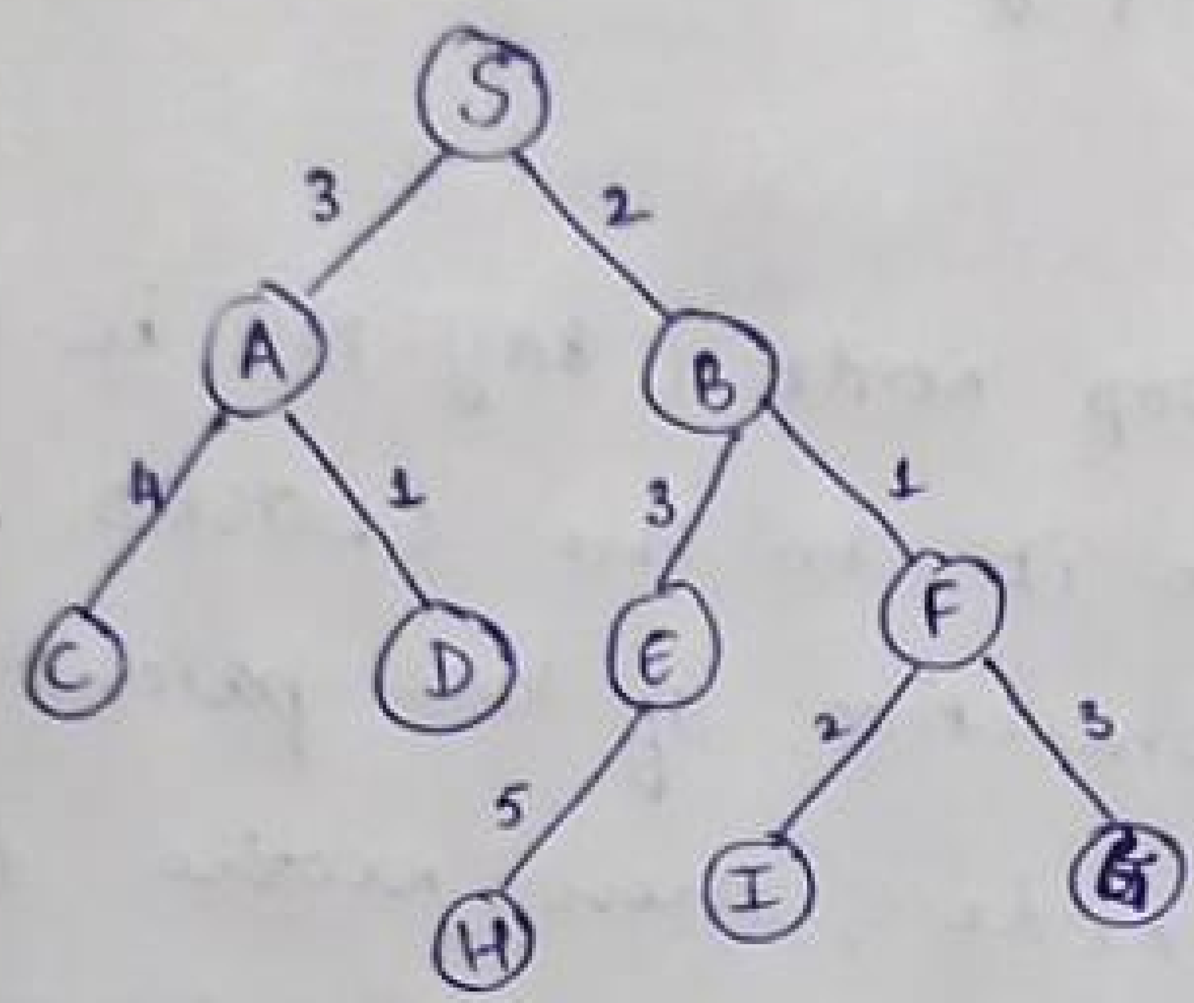
EXP 5 : Best First Search & A* algo

Problem Formulation : Given a graph, a starting node & h(n), use the evaluation function to decide which is the most promising node for reaching to the destination & explore till it reaches the destination.



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

Problem Solving

→ open : [S]
priority queque (h(n)): [13]
closed: [] f(s) = h(s) = 13

→ open : [B, A]
priority queue [4, 12]
closed [S] f(B) : 4

→ open : [F, E, A]
priority queue : [2, 8, 12]
closed : [S, B]

→ open : [G, E, I, A]
priority queue : [0, 8, 9, 12]
closed : [S, B, F] f(F) = 2

→ open : [E, I, A]
priority queue : [8, 9, 12]
Closed : [S, B, F, G]
f(G) = 0

Goal state reached.

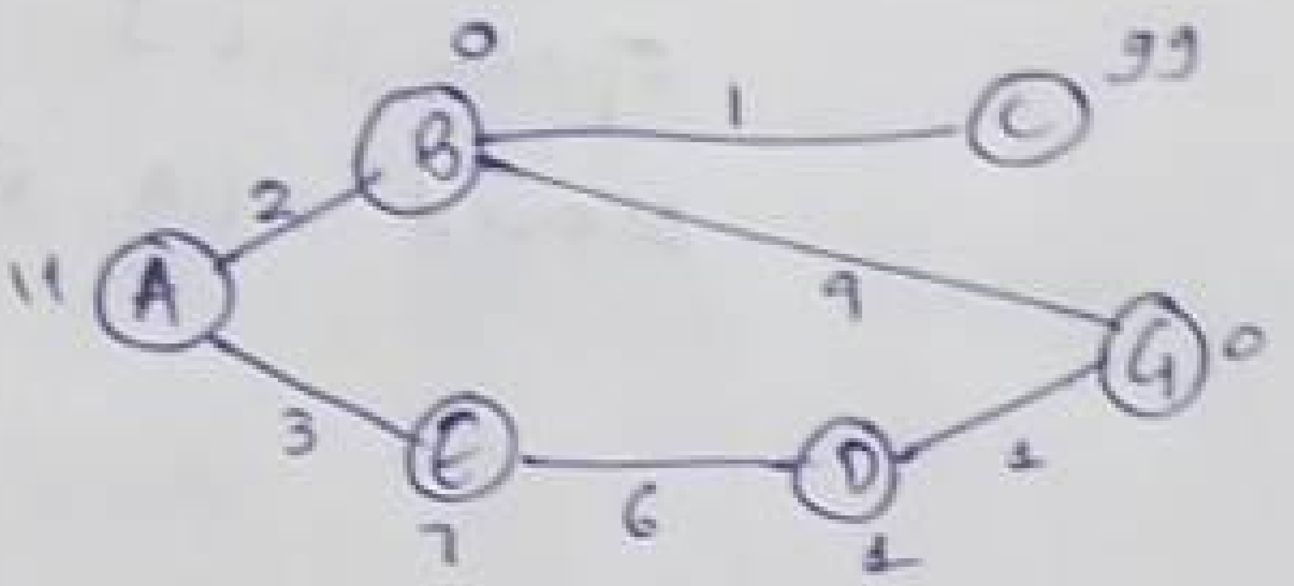
Algorithm:

1. Start
2. create 2 empty lists : OPEN & CLOSED
3. Start from the initial node (say N) and put it in the 'ordered' OPEN list.
4. Repeat the next steps ~~up~~ until GOAL node is reached.
   (i) if OPEN list is empty, then EXIT the node returning FALSE.
   (ii) 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
   (iii) If N is a GOAL node, then move the node to the closed list & exit the loop returning 'TRUE'.
   The solution can be found by backtracking the path.
   (iv) If N is not the GOAL node, expand node N to generate the immediate next nodes linked to node N & add ~~all~~ all those ~~in~~ the in to the OPEN list.
   (v) Reorder the nodes in the OPEN list in ascending order according to an evaluation function f(n).
5. STOP

Problem Formulation A* algo

Given a graph with the numbers written on edge represents the distance between the nodes while the numbers written on nodes representing heuristic values.

Find the most cost effecting path to reach from start A to final state G using A* algorithm.



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

Problem Solving

1. open [A]

closed [ ]

g(A) = 0

h(A) = 11

f(A) = 1

A has 2 nodes B, E

f(B) = 2 + 6 = 8

f(E) = 3 + 7 = 10

f(B) < f(E)

2. open : [A , B]

closed : [A]

B has 2 nodes C, G

f(C) = 2 + 1 + 99 = 102

f(G) = 2 + 9 + 0 = 11

but f(G) > f(E)

∴ we explore path from B.

3. open : [G]

Closed : [A]

E has only node D

f(D) = 3 + 6 + 1 = 10.

4. open : [D]

closed : [A, E]

D has only one node G

f(G) = 3 + 6 + 1 + 0

= 10

5. open : [G]

closed : [A, E, D]

since goal state is reached

open : [ ]

closed : [A, E, D, G]

ALGORITHM :

1. Start
2. Firstly, add the beginning node to the open list.
3. The repeat the following step
   - In the open list, find the square with the lowest ~~of~~ f cost - & this denotes the current square.
   - Now we move to the closest square.
   - ~~Now~~ consider 8 squares adjacent to the current square & ignore it if it is on the closed list, or if its is not workable.
   - check if it is on the open list, if not, add it. You need to make the current square as this square's parent. You will now record the different costs of the square like F, G, H costs
   - If it is on the open list, use G cost to measure the better path. Lower the G cost, the better the path. If this path is better, make the current path as the parent square. Now you need

Scanned by TapScanner

to recalculate the other scores - the G &
F scores of this square.

- we stop :
  - if you find the path, you need to check the closed list & the target square to it.
  - there is no path if the open list is empty & you could not find the target square.

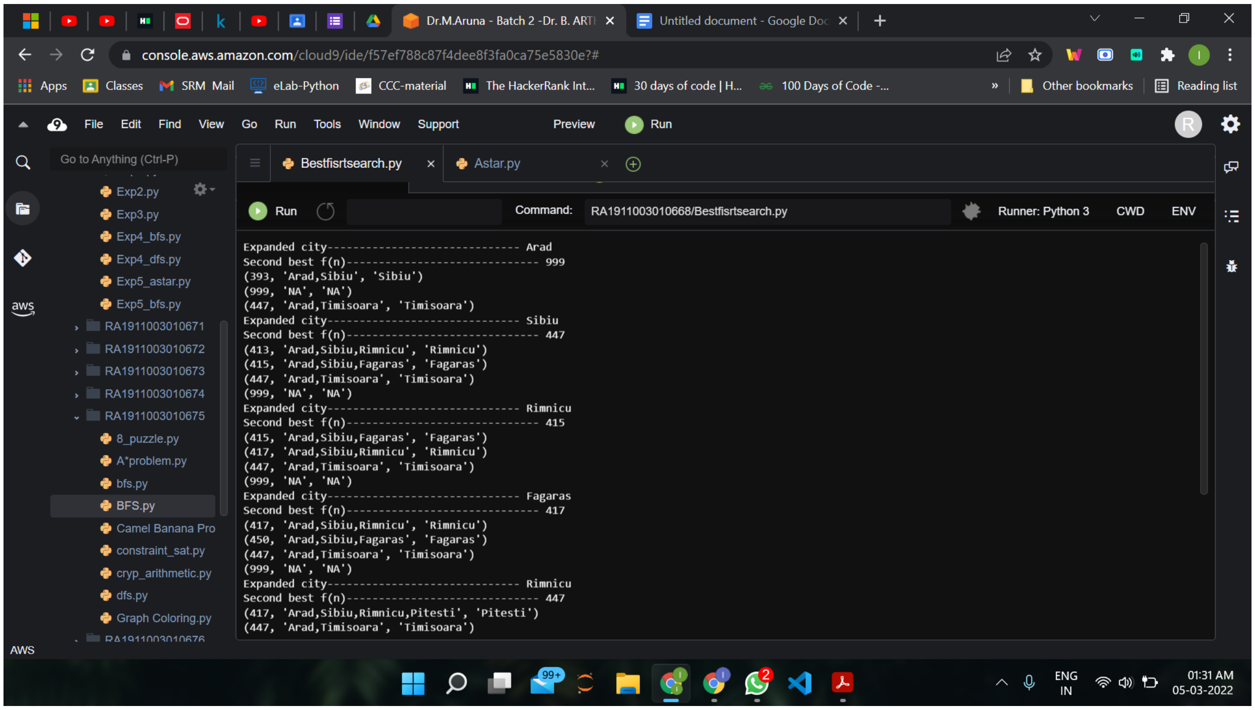
4. Save the path & work backwards to get the path.

Best First Search:

```
dict_hn={'Arad':336,'Bucharest':0,'Craiova':160,'Drobeta':242,'Eforie':161,
        'Fagaras':176,'Giurgiu':77,'Hirsova':151,'Iasi':226,'Lugoj':244,
        'Mehadia':241,'Neamt':234,'Oradea':380,'Pitesti':100,'Rimnicu':193,
        'Sibiu':253,'Timisoara':329,'Urziceni':80,'Vaslui':199,'Zerind':374}

dict_gn=dict(
Arad=dict(Zerind=75,Timisoara=118,Sibiu=140),
Bucharest=dict(Urziceni=85,Giurgiu=90,Pitesti=101,Fagaras=211),
Craiova=dict(Drobeta=120,Pitesti=138,Rimnicu=146),
Drobeta=dict(Mehadia=75,Craiova=120),
Eforie=dict(Hirsova=86),
Fagaras=dict(Sibiu=99,Bucharest=211),
Giurgiu=dict(Bucharest=90),
Hirsova=dict(Eforie=86,Urziceni=98),
Iasi=dict(Neamt=87,Vaslui=92),
Lugoj=dict(Mehadia=70,Timisoara=111),
Mehadia=dict(Lugoj=70,Drobeta=75),
Neamt=dict(Iasi=87),
Oradea=dict(Zerind=71,Sibiu=151),
Pitesti=dict(Rimnicu=97,Bucharest=101,Craiova=138),
Rimnicu=dict(Sibiu=80,Pitesti=97,Craiova=146),
Sibiu=dict(Rimnicu=80,Fagaras=99,Arad=140,Oradea=151),
Timisoara=dict(Lugoj=111,Arad=118),
Urziceni=dict(Bucharest=85,Hirsova=98,Vaslui=142),
Vaslui=dict(Iasi=92,Urziceni=142),
Zerind=dict(Oradea=71,Arad=75)
)
import queue as Q


start='Arad'
goal='Bucharest'
result=''

def get_fn(citystr):
   cities=citystr.split(',')
   hn=gn=0
   for ctr in range(0,len(cities)-1):
       gn=gn+dict_gn[cities[ctr]][cities[ctr+1]]
   hn=dict_hn[cities[len(cities)-1]]
   return(hn+gn)

def printout(cityq):
```

```
    for i in range(0,cityq.qsize()):
        print(cityq.queue[i])

def expand(cityq):
    global result
    tot,citystr,thiscity=cityq.get()
    nexttot=999
    if not cityq.empty():
        nexttot,nextcitystr,nextthiscity=cityq.queue[0]
    if thiscity==goal and tot<nexttot:
        result=citystr+'::'+str(tot)
        return
    print("Expanded city------------------------------",thiscity)
    print("Second best f(n)------------------------------",nexttot)
    tempq=Q.PriorityQueue()
    for cty in dict_gn[thiscity]:
            tempq.put((get_fn(citystr+','+cty),citystr+','+cty,cty))
    for ctr in range(1,3):
        ctrtot,ctrcitystr,ctrthiscity=tempq.get()
        if ctrtot<nexttot:
            cityq.put((ctrtot,ctrcitystr,ctrthiscity))
        else:
            cityq.put((ctrtot,citystr,thiscity))
            break
    printout(cityq)
    expand(cityq)
def main():
    cityq=Q.PriorityQueue()
    thiscity=start
    cityq.put((999,"NA","NA"))
    cityq.put((get_fn(start),start,thiscity))
    expand(cityq)
    print(result)
main()
```

Output:



A* algo:

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

Output: