LAB-C:WRITE A PROGRAM TO SOLVE THE PATH PROBLEM OF THE MAP GIVEN IN YOUR BOOK USING
A.BestFS.
B.A* SEARCH.

A: BEST FIRST SEARCH

OBJECTIVE: To implement map problem using BFS.

THEORY:

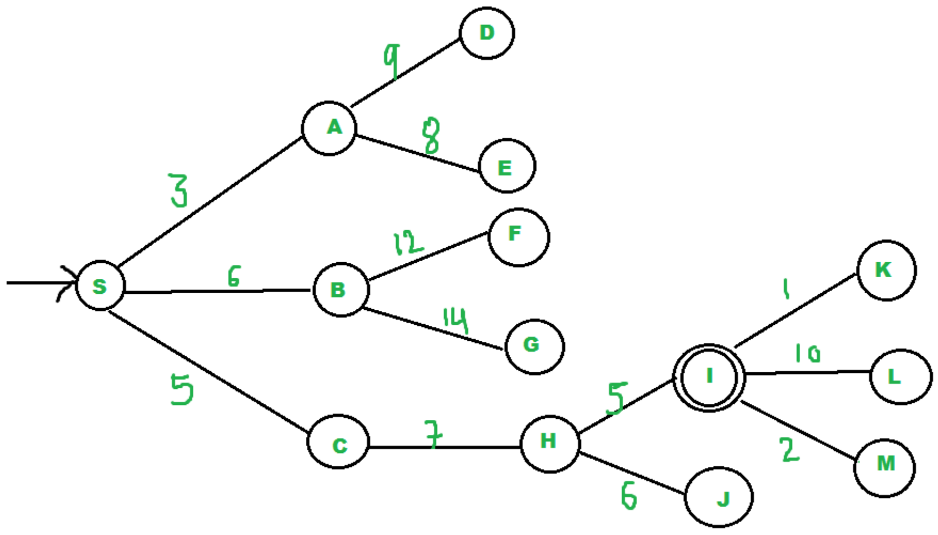
If we consider searching as a form of traversal in a graph, an uninformed search algorithm would blindly traverse to the next node in a given manner without considering the cost associated with that step. An informed search, like BFS, on the other hand, would use an evaluation function to decide which among the various available nodes is the most promising (or 'BEST') before traversing to that node. BFS uses the concept of a Priority queue and heuristic search. To search the graph space, the BFS method uses two lists for tracking the traversal. An 'Open' list that keeps track of the current 'immediate' nodes available for traversal and a 'CLOSED' list that keeps track of the nodes already traversed.

In BFS and DFS, when we are at a node, we can consider any of the adjacent as the 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.
Best First Search falls under the category of Heuristic Search or Informed Search

Implementation of Best First Search:
We use a priority queue or heap to store the costs of nodes that have the lowest evaluation function value. So the implementation is a variation of BFS, we just need to change Queue to PriorityQueue.

```
// Pseudocode for Best First Search
Best-First-Search(Graph g, Node start)
   1) Create an empty PriorityQueue
      PriorityQueue pq;
   2) Insert "start" in pq.
      pq.insert(start)
   3) Until PriorityQueue is empty
        u = PriorityQueue.DeleteMin
        If u is the goal
           Exit
        Else
           Foreach neighbor v of u
              If v "Unvisited"
                  Mark v "Visited"
                  pq.insert(v)
           Mark u "Examined"
End procedure
```

Illustration:
Let us consider the example below:



- We start from source “S” and search for goal “I” using given costs and Best First search.

- pq initially contains S
    - We remove s from and process unvisited neighbors of S to pq.
    - pq now contains {A, C, B} (C is put before B because C has lesser cost)

- We remove A from pq and process unvisited neighbors of A to pq.
    - pq now contains {C, B, E, D}

- We remove C from pq and process unvisited neighbors of C to pq.
    - pq now contains {B, H, E, D}

- We remove B from pq and process unvisited neighbors of B to pq.
    - pq now contains {H, E, D, F, G}
- We remove H from pq.
- Since our goal “I” is a neighbor of H, we return.

ALGORITHM:

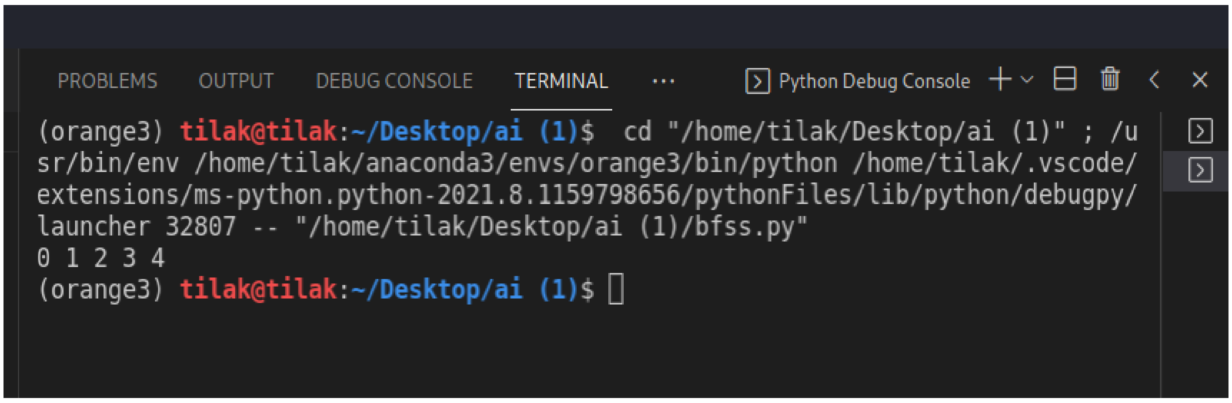
1. Create 2 empty lists: OPEN and CLOSED
2. Start from the initial node (say N) and put it in the ‘ordered’ OPEN list
3. Repeat the next steps until the GOAL node is reached
    a. If the OPEN list is empty, then EXIT the loop returning ‘False’
    b. Select the first/top node (say N) in the OPEN list 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)
4. This algorithm will traverse the shortest path first in the queue. The time complexity of the algorithm is given by O(n*logn).

PROGRAM:

```
from queue import PriorityQueue
v = 8
```

```
graph = [[] for i in range(v)]

# Function For Implementing Best First Search
# Gives output path having lowest cost

def best_first_search(actual_Src, target, n):
visited = [False] * n
pq = PriorityQueue()
pq.put((0, actual_Src))
visited[actual_Src] = True
while pq.empty() == False:
u = pq.get()[1]
# Displaying the path having lowest cost
print(u, end=" ")
if u == target:
break


for v, d in graph[u]:
if visited[v] == False:
visited[v] = True
pq.put((d, v))
print()
def addedge(a, b, cost):
graph[a].append((b, cost))
graph[b].append((a, cost))
addedge(0, 1, 7)
addedge(0, 2, 15)
addedge(1, 3, 28)
addedge(3, 4, 17)
addedge(2, 3, 9)
source = 0
target = 4
best_first_search(source, target, v)
```

OUTPUT:



```
(orange3) tilak@tilak:~/Desktop/ai (1)$  cd "/home/tilak/Desktop/ai (1)" ; /u
sr/bin/env /home/tilak/anaconda3/envs/orange3/bin/python /home/tilak/.vscode/
extensions/ms-python.python-2021.8.1159798656/pythonFiles/lib/python/debugpy/
launcher 32807 -- "/home/tilak/Desktop/ai (1)/bfss.py"
0 1 2 3 4
(orange3) tilak@tilak:~/Desktop/ai (1)$ 
```

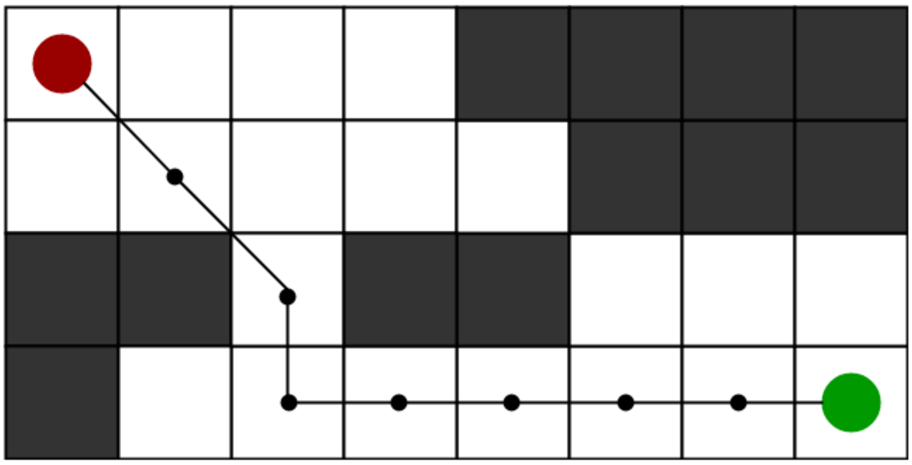
CONCLUSION:In this way we implemented Best First Search in map problem   successfully

B:A* SEARCH.

OBJECTIVE: To implement map problem using A*.

THEORY:

Motivation
To approximate the shortest path in real-life situations, like- in maps, games where there can be many hindrances.
We can consider a 2D Grid having several obstacles and we start from a source cell (colored red below) to reach towards a goal cell (colored green below)



Why A* Algorithm?

This Algorithm is the advanced form of the BFS algorithm (Breadth-first search), which searches for the shorter path first than, the longer paths. It is a complete as well as an **optimal** solution for solving path and grid problems.
Optimal – find the least cost from the starting point to the ending point. Complete – It means that it will find all the available paths from start to end.

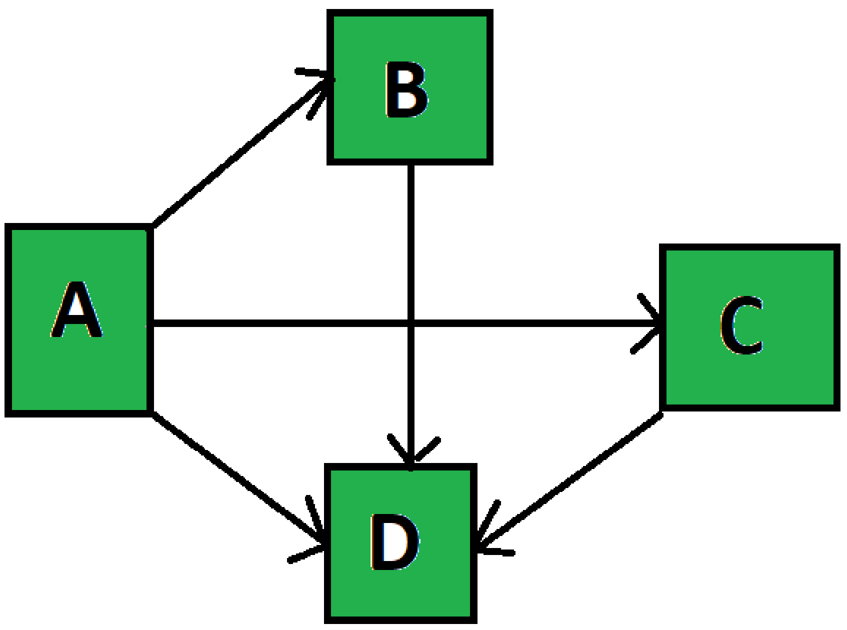
Basic concepts of A*

The A* algorithm basically reaches the optimum result by calculating the positions of all the other nodes between the starting node and the ending node. In addition, it is faster than Dijkstra's algorithm due to the heuristic function[2].
$f(n) = g(n) + h(n)$
Where
g (n) : The actual cost path from the start node to the current node.
h (n) : The actual cost path from the current node to goal node.
f (n) : The actual cost path from the start node to the goal node.
For the implementation of A* algorithm we have to use two arrays namely OPEN and CLOSE.

OPEN:
An array that contains the nodes that have been generated but have not been yet examined till yet.

CLOSE:
An array which contains the nodes which are examined.

A* algorithm is best when it comes to finding paths from one place to another. It always makes sure that the founded path is the most efficient. This is the implementation of A* on a graph structure



Advantages of A* Algorithm in Python

- It is fully complete and optimal.
- This is the best one of all the other techniques. We use to solve all the complex problems through this algorithm.
- The algorithm is optimally efficient, i.e., there is no other optimal algorithm that is guaranteed to expand fewer nodes than A*.

Disadvantages of A* Algorithm in Python

- This algorithm is complete if the branching factor is finite of the algorithm and every action has a fixed cost.
- The speed execution of A* search is highly dependant on the accuracy of the heuristic algorithm that is used to compute h (n) and is a bit slower than other algorithms.
- It is having complex problems.

Pseudo-code of A* algorithm

```
let openList equal empty list of nodes
let closedList equal empty list of nodes
put startNode on the openList (leave it's f at zero)
while openList is not empty
```

```
let currentNode equal the node with the least f value
remove currentNode from the openList
add currentNode to the closedList
if currentNode is the goal
    You've found the exit!
let children of the currentNode equal the adjacent nodes
for each child in the children
    if child is in the closedList
        continue to beginning of for loop
    child.g = currentNode.g + distance b/w child and current
    child.h = distance from child to end
    child.f = child.g + child.h
    if child.position is in the openList's nodes positions
        if child.g is higher than the openList node's g
            continue to beginning of for loop
    add the child to the openList
```

ALGORITHM:

1: Firstly, Place the starting node into OPEN and find its f (n) value.
2: Then remove the node from OPEN, having the smallest f (n) value. If it is a goal node, then stop and return to success.
3: Else remove the node from OPEN, and find all its successors.
4: Find the f (n) value of all the successors, place them into OPEN, and place the removed node into CLOSE.
5: Goto Step-2.
6: Exit.

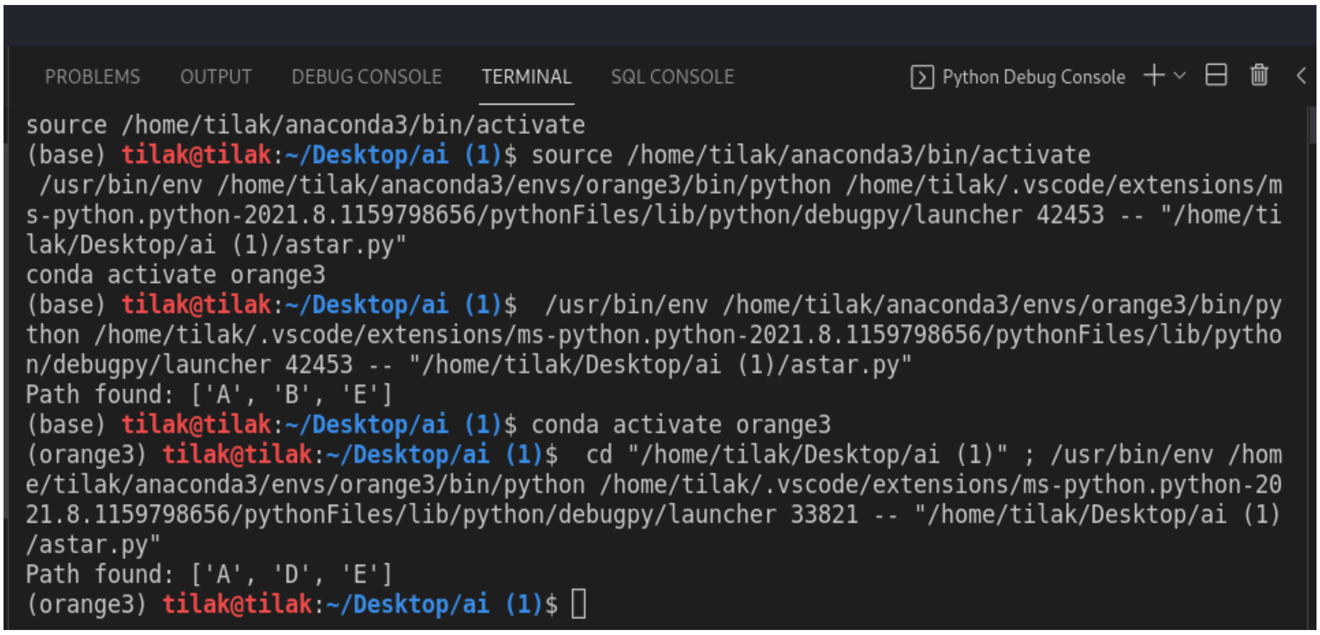
PROGRAM:

```
from collections import deque


class Graph:
        def __init__(self, adjacency_list):
                self.adjacency_list = adjacency_list


def get_neighbors(self, v):
        return self.adjacency_list[v]


def h(self, n):
        H = {
                'A': 4,
                'B': 6,
                'C': 1,
                'D': 3,
                'E':12
        }
        return H[n]


def a_star_algorithm(self, start_node, stop_node):
        """ open_list is a list of nodes which have been visited, but who's neighbors
```

```
haven't all been inspected, starts off with the start node
closed_list is a list of nodes which have been visited
and who's neighbors have been inspected"""

open_list = set([start_node])
closed_list = set([])
# g contains current distances from start_node to all other nodes
# the default value (if it's not found in the map) is +infinity
g = {}


g[start_node] = 0


# parents contains an adjacency map of all nodes
parents = {}
parents[start_node] = start_node


while len(open_list) > 0:
n = None


# find a node with the lowest value of f() - evaluation function
for v in open_list:
if n == None or g[v] + self.h(v) < g[n] + self.h(n):
n = v


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:
reconst_path = []


while parents[n] != n:
reconst_path.append(n)
n = parents[n]


reconst_path.append(start_node)


reconst_path.reverse()


print('Path found: {}'.format(reconst_path))
return reconst_path
```

```
            # for all neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
            # if the current node isn't in both open_list and closed_list
            # add it to open_list and note n as it's parent
            if m not in open_list and m not in closed_list:
            open_list.add(m)
            parents[m] = n
            g[m] = g[n] + weight


            # otherwise, check if it's quicker to first visit n, then m
            # and if it is, update parent data and g data
            # and if the node was in the closed_list, move it to open_list
            else:
            if g[m] > g[n] + weight:
            g[m] = g[n] + weight
            parents[m] = n
            if m in closed_list:
            closed_list.remove(m)
            open_list.add(m)
            open_list.remove(n)
            closed_list.add(n)
            print('Path does not exist!')
            return None
    if __name__ == '__main__':
    adjac_lis = {
    'A': [('B', 2),('C', 6), ('D', 7)],
    'B': [('E', 16)],
    'C': [('D', 7)],
    'D': [('E',2)]


    }
    graph1 = Graph(adjac_lis)
    graph1.a_star_algorithm('A', 'E')
```

OUTPUT:



```
source /home/tilak/anaconda3/bin/activate
(base) tilak@tilak:~/Desktop/ai (1)$ source /home/tilak/anaconda3/bin/activate
 /usr/bin/env /home/tilak/anaconda3/envs/orange3/bin/python /home/tilak/.vscode/extensions/m
s-python.python-2021.8.1159798656/pythonFiles/lib/python/debugpy/launcher 42453 -- "/home/ti
lak/Desktop/ai (1)/astar.py"
conda activate orange3
(base) tilak@tilak:~/Desktop/ai (1)$  /usr/bin/env /home/tilak/anaconda3/envs/orange3/bin/py
thon /home/tilak/.vscode/extensions/ms-python.python-2021.8.1159798656/pythonFiles/lib/pytho
n/debugpy/launcher 42453 -- "/home/tilak/Desktop/ai (1)/astar.py"
Path found: ['A', 'B', 'E']
(base) tilak@tilak:~/Desktop/ai (1)$ conda activate orange3
(orange3) tilak@tilak:~/Desktop/ai (1)$  cd "/home/tilak/Desktop/ai (1)" ; /usr/bin/env /hom
e/tilak/anaconda3/envs/orange3/bin/python /home/tilak/.vscode/extensions/ms-python.python-20
21.8.1159798656/pythonFiles/lib/python/debugpy/launcher 33821 -- "/home/tilak/Desktop/ai (1)
/astar.py"
Path found: ['A', 'D', 'E']
(orange3) tilak@tilak:~/Desktop/ai (1)$ 
```

Explanation:

In this code, we have made the class named Graph, where multiple functions perform different operations. There is written with all the functions what all operations that function is performing. Then some conditional statements will perform the required operations to get the minimum path for traversal from one node to another node. Finally, we will get the output as the shortest path to travel from one node to another.

CONCLUSION:

A* in Python is a powerful and beneficial algorithm with all the potential. However, it is only as good as its heuristic function, which is highly variable considering a problem’s nature. It has found its applications in software systems in machine learning and search optimization to game development.