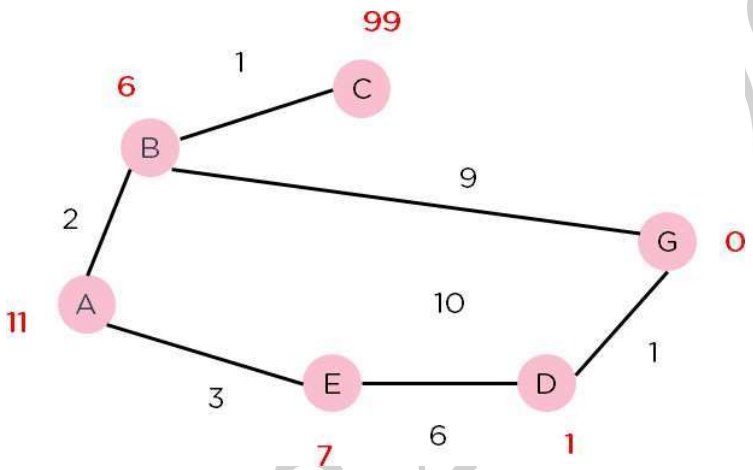
EX.No : **DATE :**

A* SEARCH ALGORITHM

A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.
All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all paths transverse gives you the cost of that route.
Initially, the Algorithm calculates the cost to all its immediate neighboring nodes,n, and chooses the one incurring the least cost. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If f(n) represents the final cost, then it can be denoted as :
f(n) = g(n) + h(n), where :
g(n) = cost of traversing from one node to another. This will vary from node to node
h(n) = heuristic approximation of the node's value. This is not a real value but an approximation cost.



CODE:

```
from collections import deque

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

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

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

    def a_star_algorithm(self, start, stop):
        open_lst = set([start])
        closed_lst = set([])
        poo = {}
        poo[start] = 0
        par = {}
        par[start] = start

        while len(open_lst) > 0:
            n = None
            for v in open_lst:
                if n == None or poo[v] + self.h(v) < poo[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            if n == stop:
                reconst_path = []

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

                reconst_path.append(start)

                reconst_path.reverse()
```

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


        for (m, weight) in self.get_neighbors(n):
            if m not in open_lst and m not in closed_lst:
                open_lst.add(m)
                par[m] = n
                poo[m] = poo[n] + weight
            else:
                if poo[m] > poo[n] + weight:
                    poo[m] = poo[n] + weight
                    par[m] = n

                    if m in closed_lst:
                        closed_lst.remove(m)
                        open_lst.add(m)
        open_lst.remove(n)
        closed_lst.add(n)

    print('Path does not exist!')
    return None

adjac_lis = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjac_lis)
graph1.a_star_algorithm('A', 'D')
```

OUTPUT:

```
Path found: ['A', 'B', 'D']
```

RESULT:

Thus, the code has been successfully executed, and the output has been verified successfully.