EX.NO: 5 **DATE: 06 - 09 - 2024**

A* SEARCH ALGORITHM

AIM:

To implement a A* heuristic algorithm to find the least-cost path in a graph using node weights and heuristic approximations for efficient traversal.

A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.

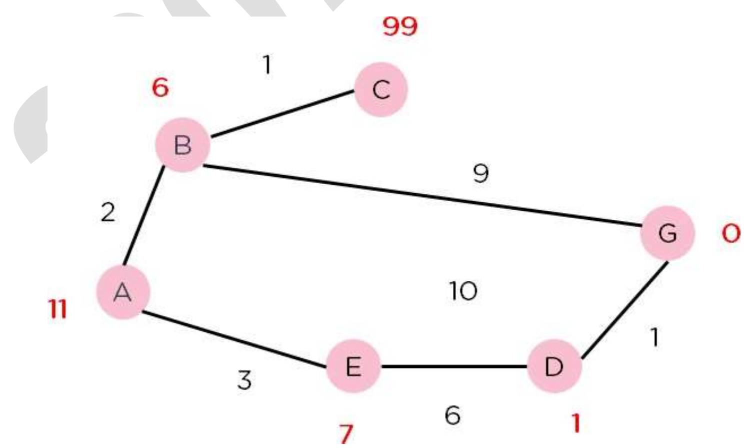
All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all paths transverse gives you the cost of that route.

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.



PROGRAM:

```
import heapq

class Node:
    def _init_(self, name, parent=None, g=0, h=0):
        self.name = name
```

```
        self.parent = parent
        self.g = g
        self.h = h
        self.f = g + h

    def _lt_(self, other):
        return self.f < other.f

def a_star(graph, start, goal, h_func):
    open_list = []
    heapq.heappush(open_list, Node(start, None, 0, h_func(start, goal)))
    closed_list = set()

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.name == goal:
            path = []
            while current_node:
                path.append(current_node.name)
                current_node = current_node.parent
            return path[::-1]

        closed_list.add(current_node.name)

        for neighbor, cost in graph[current_node.name]:
            if neighbor in closed_list:
                continue

            g_new = current_node.g + cost
            h_new = h_func(neighbor, goal)
            f_new = g_new + h_new

            neighbor_node = Node(neighbor, current_node, g_new, h_new)
            heapq.heappush(open_list, neighbor_node)

    return None
```

```
graph = {
    'A': [('D', 1), ('C', 3)],
    'B': [('A', 1), ('D', 1), ('E', 3)],
    'C': [('A', 3), ('F', 2)],
    'D': [('B', 1)],
    'E': [('B', 3), ('F', 1)],
    'F': [('C', 2), ('E', 1)]
}

def heuristic(node, goal):
    return 0

start_node = 'D'
goal_node = 'F'
path = a_star(graph, start_node, goal_node, heuristic)

if path:
    print(f"Path found: {path}")
else:
    print("No path found")
```

OUTPUT:

RESULT:

Thus , the heuristic algorithm successfully identifies an efficient, least-cost path in the graph by evaluating node weights and heuristic estimates.