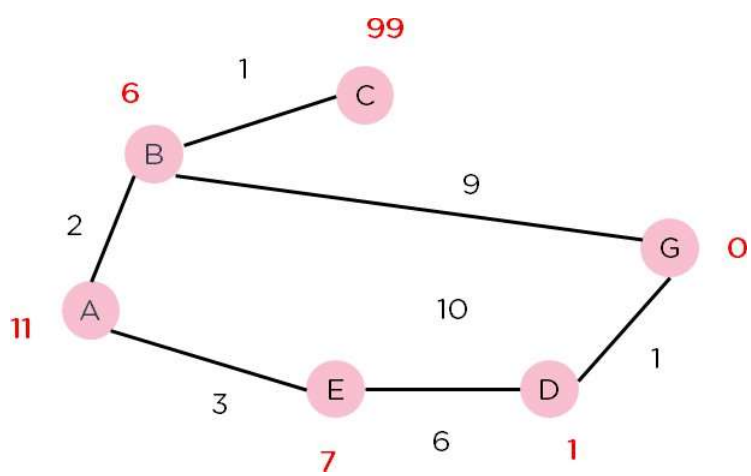
**EX.No :** **DATE :**

# 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:

```python
# Defining the graph from the second image
graph = {
    'A': [('B', 2), ('E', 3)],
    'B': [('A', 2), ('C', 1)],
    'C': [('B', 1), ('G', 9)],
    'D': [('E', 6), ('G', 1)],
    'E': [('A', 3), ('D', 6)],
    'G': [('C', 9), ('D', 1)]
}

# Heuristic values (red numbers next to each node, assuming G is the goal)
heuristic = {
    'A': 11,
    'B': 6,
    'C': 99,  # High heuristic to simulate distant node
    'D': 1,
    'E': 7,
    'G': 0   # G is the goal
}

# A* algorithm implementation remains the same as before
def a_star(graph, start, goal):
    from queue import PriorityQueue

    # Priority queue to store (f(n), node)
    open_list = PriorityQueue()
    open_list.put((0, start))

    # Dictionary to store the cost to reach each node
    g_costs = {start: 0}

    # Dictionary to store the path (parent nodes)
    came_from = {start: None}

    # Dictionary for storing the total estimated cost
    f_costs = {start: heuristic[start]}

    while not open_list.empty():
        current = open_list.get()[1]  # Get the node with the lowest f(n)

        # If we've reached the goal, reconstruct the path
        if current == goal:
            return reconstruct_path(came_from, current)

        # Explore neighbors
        for neighbor, cost in graph[current]:
            tentative_g_cost = g_costs[current] + cost

            # If the new path to neighbor is shorter or neighbor not visited
            if neighbor not in g_costs or tentative_g_cost < g_costs[neighbor]:
                g_costs[neighbor] = tentative_g_cost
                f_cost = tentative_g_cost + heuristic[neighbor]
                f_costs[neighbor] = f_cost
                open_list.put((f_cost, neighbor))
                came_from[neighbor] = current

    return None  # Return None if no path is found

# Reconstruct the path from start to goal
def reconstruct_path(came_from, current):
    path = []
    while current:
        path.append(current)
        current = came_from[current]
    path.reverse()
    return path

# Test the algorithm with 'A' as the start node and 'G' as the goal node
start_node = 'A'
goal_node = 'G'
path = a_star(graph, start_node, goal_node)
print("Shortest path:", path)
```

## OUTPUT:

```
Shortest path: ['A', 'E', 'D', 'G']
```

## RESULT:

Thus Program is Executed Successfully And Output is Verified.