EX.No : 5

DATE :

A* SEARCH ALGORITHM

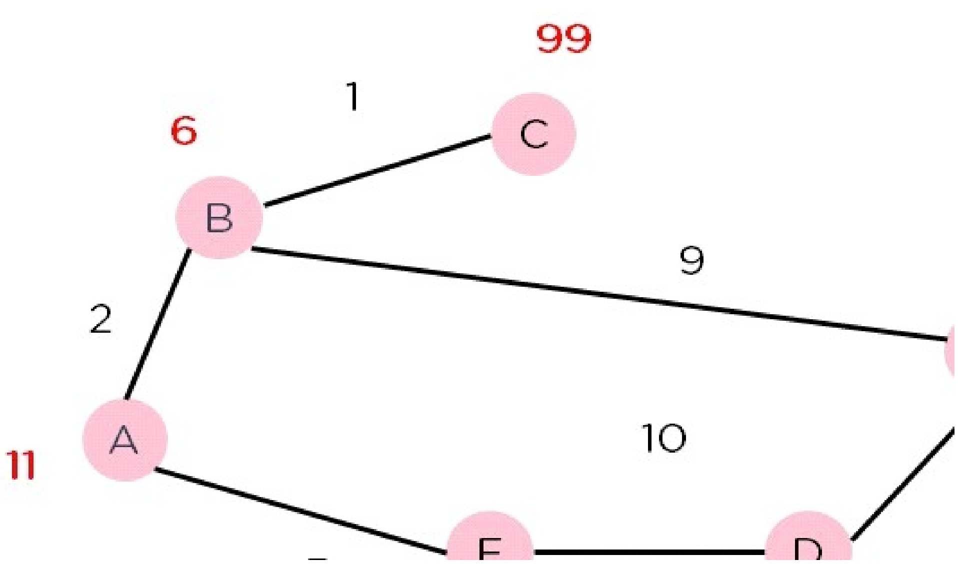
A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.

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.



AIM :

To implement an A* search algorithm using Python.

<u>CODE:</u>

+ Code + Text

```
# Experiment 4 - a star informed search
import heapq

def a_star(graph, start, goal, h):
    # Priority queue to store (f, node)
    open_set = []
    heapq.heappush(open_set, (h[start], start))

    # Store the cost from start to each node
    g = {node: float('inf') for node in graph}
    g[start] = 0

    # Store the estimated total cost from start to goal through each node
    f = {node: float('inf') for node in graph}
    f[start] = h[start]

    # Track the path
    came_from = {}

    # Closed set to track visited nodes
    closed_set = set()

    while open_set:
        # Pop the node with the lowest f value
        _, current = heapq.heappop(open_set)

        # If the goal is reached, reconstruct the path
        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]  # Return reversed path

        closed_set.add(current)

        # Explore neighbors
        for neighbor, cost in graph[current]:
```

```
+ Code   + Text
        # Explore neighbors
        for neighbor, cost in graph[current]:
            if neighbor in closed_set:
                continue

            tentative_g = g[current] + cost

            if tentative_g < g[neighbor]:
                came_from[neighbor] = current
                g[neighbor] = tentative_g
                f[neighbor] = g[neighbor] + h[neighbor]
                if neighbor not in open_set:
                    heapq.heappush(open_set, (f[neighbor], neighbor))

    return None  # If no path found

# Example usage

# Define the graph as an adjacency list
graph = {
    'A': [('B', 1), ('C', 3)],
    'B': [('A', 1), ('D', 1), ('E', 5)],
    'C': [('A', 3), ('F', 2)],
    'D': [('B', 1)],
    'E': [('B', 5), ('F', 2)],
    'F': [('C', 2), ('E', 2), ('G', 1)],
    'G': [('F', 1)]
}

# Define the heuristic function (straight-line distance to goal, G)
heuristic = {
    'A': 7,
    'B': 6,
    'C': 2,
    'D': 5,
    'E': 3,
    'F': 1,
    'G': 0  # Goal
}
```

```
# Call A* algorithm
start_node = 'A'
goal_node = 'G'
path = a_star(graph, start_node, goal_node, heuristic)

# Output the result
if path:
    print("Path found:", path)
else:
    print("No path found")
```

OUTPUT:

```
Path found: ['A', 'C', 'F', 'G']
```

RESULT :

Thus the output is successfully executed and verified