Experiment No: 4 A* Search

Aim:

To implement the A* search algorithm in Python. The A* algorithm is used to find the shortest path from a start node to a goal node in a graph, using both the actual cost from the start node and a heuristic estimate to the goal.

Algorithm:

1) Initialize the graph: Represent the graph as a dictionary where each node has a list of tuples representing its neighbouring nodes and the cost to reach them.
2) Heuristic Function: Define a heuristic function h(n) that estimates the cost from node n to the goal node. The choice of heuristic depends on the problem domain. For simplicity, assume we are given this heuristic function.
3) Define the A* search Function:
   - Initialize two sets: open_set (nodes to be evaluated) and closed_set (nodes already evaluated).
   - use a priority queue (min-heap) to pick the node with the lowest $f(n) = g(n) + h(n)$, where $g(n)$ is the cost from the start node to n and $h(n)$ is the heuristic estimate from n to the goal.
   - Initialize the g values (costs from the start) to infinity for all nodes except the start node, which is initialized to 0.
   - Initialize the f values (estimated total cost) using the heuristic for the start node.
   - Track the parent of each node to reconstruct the path after reaching the goal.

4) A* search Process:

- While open_set is not empty:
  - Select the node n in open_set with the lowest f(n).
  - If n is the goal node, reconstruct the path and return it.
  - Move n from open_set to closed_set.
  - For each neighbour of n, calculate the tentative g value. If this is lower than the current g value for the neighbor, update the g value and calculate the new f value. If the neighbor is not in open_set, add it.
- If the goal node is not reached and open_set is empty, return that there is no solution.

Program:

```
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 neighbours
    for neighbors, cost in graph[current]:
        if neighbour 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:

Example Output:

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

Result:

Thus, the program for A* search is successfully executed and the output is verified