Assignment No. 08

Title: Implementation of A* (A-star) Algorithm using Graph Search

---

1. Aim

To implement the **A*** (A-star) algorithm for solving AI search problems using the **Graph Search** method.

---

2. Objectives

- To understand the working of heuristic search in Artificial Intelligence.
- To explore the use of **A*** algorithm in pathfinding and graph traversal problems.
- To analyze the efficiency of informed search strategies compared to uninformed search methods.

---

3. Theory

The **A*** algorithm is an **informed search strategy** that combines the strengths of **Uniform Cost Search (UCS)** and **Greedy Best-First Search**.
It uses both:

- The **actual cost** to reach a node → *g(n)*
- The **estimated cost** from that node to the goal → *h(n)*

Evaluation Function

$$f(n) = g(n) + h(n)$$

Where:

- **g(n):** Cost from the start node to current node *n*.
- **h(n):** Heuristic estimate of the cost from *n* to goal.
- **f(n):** Estimated total cost of the path through *n*.

Applications

- Pathfinding in maps (e.g., GPS navigation).
- Game AI (for NPC movement).
- Robot navigation and motion planning.

---

4. Algorithm (Steps of A*)

1. Initialize the **open list** with the start node.
2. Initialize the **closed list** as empty.
3. Repeat until the goal is found or the open list becomes empty:
   - Select the node with the **lowest f(n)** from the open list.
   - If this node is the **goal**, return success and trace back the path.
   - Otherwise, expand the node:

        - For each successor, calculate *g(n)*, *h(n)*, and *f(n)*.
        - If the successor is **not** in open/closed lists, add it to the open list.
        - If it is already present with a **higher cost**, update its values.
    - Move the expanded node to the **closed list**.
4. If the open list becomes empty and the goal is not found → **return failure**.

---

5. Python Implementation

```
# A* Algorithm Implementation

from queue import PriorityQueue

# Example Graph
graph = {
   'S': {'A': 1, 'B': 4},
   'A': {'B': 2, 'C': 5, 'D': 12},
   'B': {'C': 2},
   'C': {'D': 3, 'G': 7},
   'D': {'G': 2},
   'G': {}
}

# Heuristic values
heuristics = {
   'S': 7, 'A': 6, 'B': 4,
   'C': 2, 'D': 1, 'G': 0
}

def a_star_search(start, goal):
   open_list = PriorityQueue()
   open_list.put((0, start))
   g_cost = {start: 0}
   parent = {start: None}

   while not open_list.empty():
      _, current = open_list.get()

      if current == goal:
         path = []
```

```
        while current:
            path.append(current)
            current = parent[current]
        path.reverse()
        return path, g_cost[goal]

    for neighbor, cost in graph[current].items():
        new_g = g_cost[current] + cost
        if neighbor not in g_cost or new_g < g_cost[neighbor]:
            g_cost[neighbor] = new_g
            f = new_g + heuristics[neighbor]
            open_list.put((f, neighbor))
            parent[neighbor] = current

    return None, float('inf')

# Run A*
start, goal = 'S', 'G'
path, total_cost = a_star_search(start, goal)

print("Path found:", path)
print("Total cost:", total_cost)
```

---

6. Sample Output

Path found: ['S', 'A', 'B', 'C', 'D', 'G']
Total cost: 10

---

7. Observations

- The **A*** algorithm expands fewer nodes than BFS or DFS because it uses heuristics for guidance.
- Its **optimality** depends on how accurate and **admissible** the heuristic function is.
- In the given example, the algorithm successfully finds the **shortest path** with **minimum total cost**.
