**Name: Usman Ul Haq**

**Roll# BSAIM-035**

**AI LAB TASKs**

**Documentation**

**Write code for A\* Algorithm**

This document provides an explanation of an implementation of the *A search algorithm*\*, which is used to find the shortest path between two nodes in a graph. The algorithm combines the benefits of **Dijkstra’s Algorithm (cost-based search)** and **Greedy Best-First Search (heuristic-based search)** to efficiently determine the optimal path.

**Key Concepts**

* **Graph Representation**
  + The graph is represented as an adjacency list, where each node stores a list of its neighboring nodes along with the cost to travel to them.
* *A Algorithm Principles*\*
  + The algorithm uses two key values for each node:
    - **g(n):** The actual cost from the start node to the current node.
    - **h(n):** A heuristic estimate of the cost from the current node to the goal.
    - **f(n) = g(n) + h(n):** The total estimated cost, which is used to decide which node to explore next.

**Code Explanation**

**1. Graph Class**

* The Graph class is used to create and manage the graph.
* The add\_edge(node, neighbor, cost) method adds a directed edge between two nodes with a given cost.

***2. A Algorithm Implementation*\***

* **Initialization**
  + The algorithm starts with an **open list**, which keeps track of nodes to be explored, and a **closed set**, which stores nodes that have already been visited.
  + The open list is initialized with the start node and its heuristic value.
* **Main Loop**
  + The open list is sorted based on f(n) = g(n) + h(n), ensuring the node with the lowest estimated cost is processed first.
  + The node with the lowest f(n) is removed from the open list and added to the closed set.
  + If this node is the goal, the algorithm reconstructs and returns the shortest path along with the total cost.
* **Exploring Neighbors**
  + The algorithm iterates through the current node’s neighbors and calculates their g(n) + h(n).
  + If a neighbor has not been visited before, it is added to the open list along with its updated cost.

**Graph Example Used in the Code**

The following graph is created in the program:

A

/ \

B C

| | \

D D E

\ | |

\ | |

F

* Nodes: **A, B, C, D, E, F**
* Edges and Costs:
  + A → B (1)
  + A → C (4)
  + B → D (2)
  + C → D (1)
  + C → E (3)
  + D → F (5)
  + E → F (2)

**Heuristic Values (h(n))**

| **Node** | **Heuristic (h(n))** |
| --- | --- |
| A | 6 |
| B | 4 |
| C | 4 |
| D | 2 |
| E | 3 |
| F | 0 (Goal) |

The heuristic values represent an estimated cost from each node to the goal **(F)**.

**Execution & Output**

* The A\* algorithm is executed to find the shortest path from **A** to **F**.
* The shortest path found is:

css

CopyEdit

A → B → D → F

* **Total cost:** **8**

***Advantages of A Search*\***

✅ **Guaranteed to find the shortest path** (if the heuristic is admissible)  
✅ **More efficient than Dijkstra’s algorithm** when a good heuristic is available  
✅ **Avoids unnecessary node expansion**

**Limitations**

⚠️ **Performance depends on the quality of the heuristic function**  
⚠️ **If the heuristic is inaccurate, the algorithm may not perform optimally**  
⚠️ **Sorting the open list at each step can be inefficient (Using a priority queue would improve performance)**

**Conclusion**

The A\* algorithm is an efficient pathfinding method widely used in AI applications such as **robotics, game development, and navigation systems**. This implementation demonstrates how A\* can be applied to a weighted graph to find the most cost-effective route.