# A\* Algorithm (Dynamic Implementation) - Documentation

## Overview

This Python program implements the A\* (A-star) search algorithm, allowing users to dynamically input nodes, edges, weights, and heuristic values. The A\* algorithm is used to find the shortest path between two points in a graph, combining the advantages of Dijkstra’s algorithm (for path cost) and Greedy Best-First Search (for heuristic estimation).

## Key Features

• Dynamic input for nodes, edges, weights, and heuristic values.

• Uses adjacency list representation for graph storage.

• Implements the A\* algorithm for pathfinding between nodes.

• Displays both adjacency list and heuristic table before computation.

• Provides reconstructed path after successful search.

## Class and Function Explanation

1. Class: Graph

This class encapsulates all the functionality required for managing a graph, including adding edges, storing heuristics, and implementing the A\* algorithm.

2. add\_edge(node1, node2, weight)

Adds a directed edge from node1 to node2 with a given weight. If node1 does not already exist in the adjacency list, it initializes a new list entry for it.

3. set\_heuristic(node, value)

Stores a heuristic value for a given node, which represents the estimated cost to reach the goal.

4. get\_neighbors(v)

Returns all connected nodes and their respective edge weights for a given vertex 'v'.

5. h(n)

Fetches the heuristic value of node 'n'. If no value is set, returns a default value of 1.

6. a\_star\_algorithm(start\_node, stop\_node)

This is the main function implementing the A\* search algorithm. It uses the following structures:  
- open\_list: a set of nodes that are discovered but not yet explored.  
- closed\_list: a set of nodes that have already been explored.  
- g: a dictionary storing the current shortest distance from the start node to each node.  
- parents: a dictionary used to reconstruct the final path once the goal is reached.

Algorithm Steps:  
1. Start with the start\_node in the open\_list.  
2. Pick the node with the lowest f(n) = g(n) + h(n) value.  
3. For each neighbor, calculate potential new costs and update paths if a shorter route is found.  
4. Move fully explored nodes to the closed\_list.  
5. Continue until the stop\_node is reached or all nodes are explored.

## Path Reconstruction

When the stop\_node is found, the algorithm reconstructs the path by tracing back through the parents dictionary from the goal node to the start node, then reverses the list to display the correct order.

## User Input Flow

1. The program first asks the user to enter the number of edges.  
2. For each edge, the user specifies source node, destination node, and weight.  
3. Next, the user enters the number of nodes for which heuristic values will be set.  
4. The program asks for start and end nodes to begin the A\* search.  
5. Finally, it prints the adjacency list, heuristic dictionary, and the resulting path.

## Sample Input and Output

Example Input:  
Enter number of edges: 3  
From node: A  
To node: B  
Enter weight from A to B: 1  
From node: A  
To node: C  
Enter weight from A to C: 3  
From node: B  
To node: D  
Enter weight from B to D: 5  
  
Enter number of nodes (for heuristic values): 4  
Node name: A  
Heuristic value for A: 1  
Node name: B  
Heuristic value for B: 1  
Node name: C  
Heuristic value for C: 1  
Node name: D  
Heuristic value for D: 1  
  
Enter Start Node: A  
Enter End Node: D

Output:  
🔹 Adjacency List: {'A': [('B', 1), ('C', 3)], 'B': [('D', 5)], 'C': []}  
🔹 Heuristics: {'A': 1, 'B': 1, 'C': 1, 'D': 1}  
✅ Path found: ['A', 'B', 'D']

## Conclusion

This program dynamically demonstrates the working of the A\* search algorithm. By taking user input, it shows how the algorithm explores nodes, updates path costs, and finally reconstructs the most optimal route between the start and end nodes.