# A\* Algorithm (Word Code Explanation)

This document provides a full, step-by-step explanation of the A\* algorithm implementation in the provided Python code.

## 1. Class Definition

The code defines a class named 'Graph' which represents a weighted graph using an adjacency list. This adjacency list stores each node and its neighboring nodes along with the edge weights.

## 2. \_\_init\_\_ Method

The '\_\_init\_\_' method initializes the graph with the given adjacency list. It stores the adjacency list in an instance variable 'self.adjacency\_list'.

## 3. neighbors(v) Method

This method returns all neighbors of a given vertex 'v'. It simply looks up and returns the list of connected nodes from the adjacency list.

## 4. h(n) Heuristic Function

This function defines the heuristic value (estimated cost to reach the goal) for each node. In this code, all heuristic values are set to 1 for simplicity. In real scenarios, heuristic values represent the 'distance' or 'guess' of how close each node is to the goal.

## 5. a\_algo(start\_node, stop\_node) Method

This method implements the A\* (A-star) pathfinding algorithm, which finds the shortest path between two nodes by combining the actual cost (g) and the heuristic (h).

Here’s how it works step by step:

* 1. Initialize 'open\_list' with the start node. This list keeps track of nodes to explore.
* 2. Initialize 'closed\_list' as empty. This list keeps track of nodes already explored.
* 3. Initialize dictionary 'g' with the start node cost as 0. It stores the actual path cost from the start node.
* 4. Initialize 'parents' dictionary to keep track of the shortest known path to each node.
* 5. While 'open\_list' is not empty, find the node 'n' with the smallest f(n) = g(n) + h(n).
* 6. If no node is found, there’s no path, and the function returns None.
* 7. If the current node 'n' is the goal node, the path is reconstructed by backtracking through the 'parents' dictionary.
* 8. Otherwise, for each neighbor (m, weight) of 'n':
* - If the neighbor isn’t in open\_list or closed\_list, add it to open\_list and record its cost.
* - If the new path cost is smaller, update the cost and parent for that neighbor.
* 9. Move 'n' from open\_list to closed\_list after exploring its neighbors.
* 10. Continue the process until the goal node is found or all possibilities are exhausted.

## 6. Example Execution

In the example adjacency list, the graph is defined as:  
A → B (1), C (3), D (5)  
B → D (4)  
C → D (10)  
  
When we call a\_algo('A', 'D'), the algorithm finds the path A → B → D with a total cost of 5, since A→B (1) and B→D (4) gives the smallest total distance.

## 7. Summary

The A\* algorithm is an efficient pathfinding method that combines both actual path cost (g) and heuristic cost (h) to make intelligent decisions while finding the shortest route. In this implementation, the heuristic is static (set to 1), but in practical use, it would vary depending on the node’s distance from the goal.