# A\* Algorithm in Python - Documentation

## 1. Introduction

The A\* (A-Star) algorithm is one of the most popular pathfinding algorithms used in Artificial Intelligence and Computer Science. It is widely used for finding the shortest path between two nodes in a graph — especially in maps, games, and network routing. A\* combines the strengths of Dijkstra’s Algorithm (guaranteed shortest path) and Greedy Best-First Search (speed through heuristic).

## 2. Working Principle

The algorithm maintains two lists:  
- Open List: Nodes to be explored.  
- Closed List: Nodes already explored.  
  
For each node n, A\* calculates: f(n) = g(n) + h(n)  
Where:  
- g(n) = actual cost from start node to n  
- h(n) = heuristic cost (estimated distance from n to goal)  
  
The node with the lowest f(n) is selected next.

## 3. Python Implementation

Below is the full Python implementation of the A\* Algorithm:

from collections import deque  
  
class Graph:  
 def \_\_init\_\_(self, adjacency\_list):  
 self.adjacency\_list = adjacency\_list  
  
 def get\_neighbors(self, v):  
 return self.adjacency\_list[v]  
  
 def h(self, n):  
 H = {'A': 1, 'B': 1, 'C': 1, 'D': 1}  
 return H[n]  
  
 def a\_star\_algorithm(self, start\_node, stop\_node):  
 open\_list = set([start\_node])  
 closed\_list = set([])  
 g = {start\_node: 0}  
 parents = {start\_node: start\_node}  
  
 while len(open\_list) > 0:  
 n = None  
 for v in open\_list:  
 if n is None or g[v] + self.h(v) < g[n] + self.h(n):  
 n = v  
 if n is None:  
 print('Path does not exist!')  
 return None  
 if n == stop\_node:  
 reconst\_path = []  
 while parents[n] != n:  
 reconst\_path.append(n)  
 n = parents[n]  
 reconst\_path.append(start\_node)  
 reconst\_path.reverse()  
 print('Path found: {}'.format(reconst\_path))  
 return reconst\_path  
 for (m, weight) in self.get\_neighbors(n):  
 if m not in open\_list and m not in closed\_list:  
 open\_list.add(m)  
 parents[m] = n  
 g[m] = g[n] + weight  
 else:  
 if g[m] > g[n] + weight:  
 g[m] = g[n] + weight  
 parents[m] = n  
 if m in closed\_list:  
 closed\_list.remove(m)  
 open\_list.add(m)  
 open\_list.remove(n)  
 closed\_list.add(n)  
 print('Path does not exist!')  
 return None  
  
adjacency\_list = {  
 'A': [('B', 1), ('C', 3), ('D', 7)],  
 'B': [('D', 5)],  
 'C': [('D', 12)],  
 'D': []  
}  
  
graph1 = Graph(adjacency\_list)  
graph1.a\_star\_algorithm('A', 'D')

## 4. Example Execution

Input: Start Node = A, Goal Node = D

Output: Path found: ['A', 'B', 'D']

## 5. Dry Run (Step-by-Step)

Step 1: Open={A}, Closed={} -> Current Node=A  
Step 2: Open={B, C, D}, Closed={A} -> Current Node=B  
Step 3: Open={C, D}, Closed={A, B} -> Current Node=D (Goal Found)  
Final Path: A → B → D

## 6. Time and Space Complexity

Time Complexity: O(E)  
Space Complexity: O(V)  
Where E = number of edges, V = number of vertices.

## 7. Advantages of A\*

• Finds the optimal shortest path.  
• Efficient for large graphs.  
• Easy to adapt for various applications.

## 8. Limitations

• Requires a good heuristic function.  
• May consume more memory for large graphs.

## 9. Applications

• GPS Navigation Systems  
• Game AI (NPC movement)  
• Robotics Path Planning  
• Network Routing