1. **Implement A\* Search algorithm**

def aStarAlgo(start\_node, stop\_node):

open\_set = set(start\_node)

closed\_set = set()

g = {} #store distance from starting node

parents = {} # parents contains an adjacency map of all nodes

#distance of starting node from itself is zero

g[start\_node] = 0

#start\_node is root node i.e it has no parent nodes

#so start\_node is set to its own parent node

parents[start\_node] = start\_node

while len(open\_set) > 0:

n = None

#node with lowest f() is found

for v in open\_set:

if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

n = v

if n == stop\_node or Graph\_nodes[n] == None:

pass

else:

for (m, weight) in get\_neighbors(n):

#nodes 'm' not in first and last set are added to first

#n is set its parent

if m not in open\_set and m not in closed\_set:

open\_set.add(m)

parents[m] = n

g[m] = g[n] + weight

#for each node m,compare its distance from start i.e g(m) to the

#from start through n node

else:

if g[m] > g[n] + weight:

#update g(m)

g[m] = g[n] + weight

#change parent of m to n

parents[m] = n

#if m in closed set,remove and add to open

if m in closed\_set:

closed\_set.remove(m)

open\_set.add(m)

if n == None:

print('Path does not exist!')

return None

# if the current node is the stop\_node

# then we begin reconstructin the path from it to the start\_node

if n == stop\_node:

path = []

while parents[n] != n:

path.append(n)

n = parents[n]

path.append(start\_node)

path.reverse()

print('Path found: {}'.format(path))

return path

# remove n from the open\_list, and add it to closed\_list

# because all of his neighbors were inspected

open\_set.remove(n)

closed\_set.add(n)

print('Path does not exist!')

return None

#define fuction to return neighbor and its distance

#from the passed node

def get\_neighbors(v):

if v in Graph\_nodes:

return Graph\_nodes[v]

else:

return None

#for simplicity we ll consider heuristic distances given

#and this function returns heuristic distance for all nodes

def heuristic(n):

H\_dist = {

'A': 11,

'B': 6,

'C': 5,

'D': 7,

'E': 3,

'F': 6,

'G': 5,

'H': 3,

'I': 1,

'J': 0

}

return H\_dist[n]

#Describe your graph here

Graph\_nodes = {

'A': [('B', 6), ('F', 3)],

'B': [('A', 6), ('C', 3), ('D', 2)],

'C': [('B', 3), ('D', 1), ('E', 5)],

'D': [('B', 2), ('C', 1), ('E', 8)],

'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],

'F': [('A', 3), ('G', 1), ('H', 7)],

'G': [('F', 1), ('I', 3)],

'H': [('F', 7), ('I', 2)],

'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],

}

aStarAlgo('A', 'J')

**OUTPUT**

Path found: ['A', 'F', 'G', 'I', 'J']

**Explanation**

This code implements the A\* algorithm to find the shortest path between two nodes in a graph. Let's break down the code line by line:

1. `def aStarAlgo(start\_node, stop\_node):`: This line defines a function `aStarAlgo` that takes two arguments: `start\_node` (the starting node) and `stop\_node` (the destination node).

2. `open\_set = set(start\_node)`: This line initializes an open set with the starting node.

3. `closed\_set = set()`: This line initializes a closed set, which will contain nodes that have already been evaluated.

4. `g = {}`: This line initializes a dictionary `g` to store the distance from the starting node to each node.

5. `parents = {}`: This line initializes a dictionary `parents` to store the parent node for each node in the path.

6. `g[start\_node] = 0`: This line initializes the distance from the starting node to itself as 0.

7. `parents[start\_node] = start\_node`: This line sets the parent of the starting node to itself.

8. `while len(open\_set) > 0:`: This line starts a loop that continues until the open set is empty.

9. `for v in open\_set:`: This line iterates over all nodes in the open set.

10. `if n == stop\_node or Graph\_nodes[n] == None:`: This line checks if the current node is the destination node or if it has no neighbors.

11. `for (m, weight) in get\_neighbors(n):`: This line iterates over the neighbors of the current node and their corresponding weights.

12. `if m not in open\_set and m not in closed\_set:`: This line checks if the neighbor is neither in the open set nor in the closed set.

13. `else:`: This line is the `else` block for the above `if` condition, which handles the case when the neighbor is already in either the open set or the closed set.

14. `if n == None:`: This line checks if the current node is `None`, indicating that no path exists.

15. `if n == stop\_node:`: This line checks if the current node is the destination node.

16. `path = []`: This line initializes an empty list to store the path from the destination node to the starting node.

17. `while parents[n] != n:`: This line starts a loop to reconstruct the path from the destination node to the starting node.

18. `path.append(n)`: This line appends the current node to the path.

19. `path.append(start\_node)`: This line appends the starting node to the path.

20. `path.reverse()`: This line reverses the order of nodes in the path to get the correct path from the starting node to the destination node.

21. `open\_set.remove(n)`: This line removes the current node from the open set.

22. `closed\_set.add(n)`: This line adds the current node to the closed set.

23. `print('Path does not exist!')`: This line prints a message indicating that no path exists if the loop completes without finding a solution.

24. `return None`: This line returns `None` if no solution is found.

25. `def get\_neighbors(v): ...`: This block defines a function `get\_neighbors` that returns the neighbors of a given node.

26. `def heuristic(n): ...`: This block defines a heuristic function that returns the heuristic distance for a given node.

27. `Graph\_nodes = {...}`: This block defines the graph structure, where each node is mapped to its neighbors and their corresponding weights.

28. `aStarAlgo('A', 'J')`: This line calls the `aStarAlgo` function with the starting node `'A'` and the destination node `'J'` to find the shortest path between them.

This code efficiently finds the shortest path between two nodes in a graph using the A\* algorithm.