TASK:3A

```
def aStarAlgo(start_node, stop_node):
    open_set = set([start_node])
    closed_set = set()
    g = {}  # store distance from starting node
    parents = {}  # parents contain an adjacency map of all nodes

    # distance of starting node from itself is zero
    g[start_node] = 0
    # start_node is the root node, so 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 the lowest f() is found
        for v in open_set:
            if n is None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v

        if n == stop_node or n is None or Graph_nodes.get(n) is None:
            break
        else:
            for m, weight in get_neighbors(n):
                # nodes 'm' not in open_set and closed_set are added to open_set
                # n is set as 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 is in closed_set, remove and add to open_set
                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)

    # remove n from the open_set and add it to closed_set
    # because all of its neighbors were inspected
    open_set.remove(n)
    closed_set.add(n)

if n is None:
    print('Path does not exist!')
    return None

# if the current node is the stop_node,
# then we begin reconstructing 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:', path)
```

```
        return path

    print('Path does not exist!')
    return None


# define function to return neighbors and their distances from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return []

# heuristic distances for 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.get(n, 0)


# 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)],
    'J': []  # Make sure J is in the graph as well
}

print("Following is the A* Algorithm:")
aStarAlgo('A', 'J')
```

OUTPUT:

Output

```
Following is the A* Algorithm:
Path found: ['A', 'F', 'G', 'I', 'J']

=== Code Execution Successful ===
```

TASK:3B

```
def aStarAlgo(start_node, stop_node):
    open_set = set([start_node])
    closed_set = set()
    g = {}  # store distance from starting node
    parents = {}  # parents contain an adjacency map of all nodes

    # distance of starting node from itself is zero
    g[start_node] = 0
    # start_node is the root node, so 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 the lowest f() is found
        for v in open_set:
            if n is None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v

        if n == stop_node or n is None or n not in Graph_nodes:
            break

        for m, weight in get_neighbors(n):
            # nodes 'm' not in open_set and closed_set are added to open_set
            # n is set as 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
            else:
```

```
        # for each node m, compare its distance from start i.e g(m)
        # to the from start through n node
        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 is in closed_set, remove and add to open_set
          if m in closed_set:
            closed_set.remove(m)
            open_set.add(m)

  # remove n from the open_set and add it to closed_set
  # because all of its neighbors were inspected
  open_set.remove(n)
  closed_set.add(n)

if n is None:
  print('Path does not exist!')
  return None

# if the current node is the stop_node,
# then we begin reconstructing 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:', path)
        return path

    print('Path does not exist!')
    return None


# define function to return neighbors and their distances from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return []


# 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': 99,
        'D': 1,
        'E': 7,
        'G': 0
    }
    return h_dist.get(n, float('inf'))


# Describe your graph here
Graph_nodes = {
```

```
    'A': [('B', 2), ('E', 3)],
    'B': [('A', 2), ('C', 1), ('G', 9)],
    'C': [('B', 1)],
    'D': [('E', 6), ('G', 1)],
    'E': [('A', 3), ('D', 6)],
    'G': [('B', 9), ('D', 1)]
}

print("Following is the A* Algorithm:")
aStarAlgo('A', 'G')
```

OUTPUT:

```
Output
Following is the A* Algorithm:
Path found: ['A', 'E', 'D', 'G']

=== Code Execution Successful ===
```