# Program 1 : Implement A* Search algorithm.

# A* Algorithm Explanation

The A* algorithm is a widely used pathfinding algorithm in computer science and artificial intelligence. It's designed to find the shortest path between two nodes or points in a graph while taking into account both the cost to reach a node from the start and a heuristic estimate of the cost to reach the goal from that node. Here's an explanation of how the A* algorithm works:

**Real-life Examples**

- Maps
- Games

**Formula for A* Algorithm**

```
h(n) = heuristic_value
g(n) = actual_cost
f(n) = actual_cost + heursitic_value
```

```
f(n) = g(n) + h(n)
```

## Initialization

- Create an open set: This set contains nodes that need to be explored.
- Create a closed set: This set contains nodes that have already been explored.
- Initialize a dictionary to keep track of the parent node for each node.
- Initialize a dictionary to store the cost of reaching each node from the start node.
- Add the start node to the open set.
- Set the cost to reach the start node as 0.
- Set the parent of the start node as itself.

## Main Loop

- While the open set is not empty, continue the following steps.
- Select a node from the open set with the lowest combined cost. The combined cost is the sum of the cost to reach that node from the start and a heuristic estimate of the cost to reach the goal from that node.
- If the selected node is the goal node, the path is found, and the algorithm terminates.

## Exploration

- Explore the neighbors of the selected node.
- For each neighbor, calculate the cost to reach that neighbor from the start node through the selected node and update the parent node if this path is shorter.
- Calculate the heuristic estimate for the neighbor, which is an approximation of the cost to reach the goal from that neighbor.
- Add the neighbor to the open set if it's not there already, and update its cost and parent if the path through the current node is shorter.

## Termination

- If the open set becomes empty, and the goal node hasn't been reached, there is no path from the start node to the goal node. The algorithm terminates.

## Path Reconstruction

- If the goal node is reached, the algorithm traces back from the goal node to the start node using the parent information to reconstruct the shortest path.

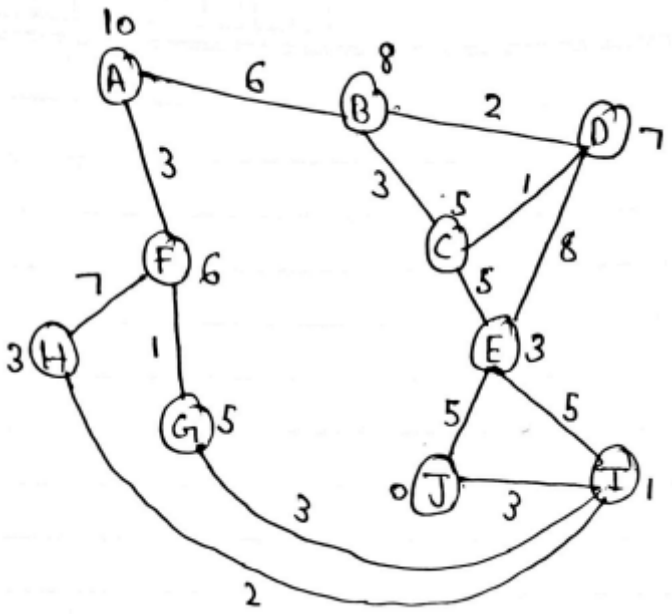
## Result

- The result is the shortest path from the start node to the goal node, along with the associated cost.

## Key Concepts

- **Heuristic Function**: A* uses a heuristic function to estimate the cost from a node to the goal. This heuristic helps guide the search toward the goal efficiently. The heuristic should be admissible, meaning it never overestimates the true cost.
- **Open Set and Closed Set**: The open set contains nodes that need to be explored, while the closed set contains nodes that have already been explored. This ensures that nodes are not revisited unnecessarily.
- **Cost Function**: A* uses a cost function to track the cost of reaching each node from the start node.

A* is a versatile algorithm and is widely used in applications like pathfinding in games, navigation systems, and various optimization problems where finding the most efficient path is crucial. It's known for its optimality (it finds the shortest path) under certain conditions and efficiency when an appropriate heuristic is used.




```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    parents = { }
    g = dict()
    g[start_node] = 0
    parents[start_node] = start_node

    while len(open_set) > 0:
        n = None

        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_neighbours(n):
                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:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)

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

        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

        open_set.remove(n)
        closed_set.add(n)

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

def get_neighbours(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None

def heuristic(n):
    H_dist = {
        'A' : 10,
        'B' : 8,
        'C' : 5,
        'D' : 7,
        'E' : 3,
        'F' : 6,
        'G' : 5,
        'H' : 3,
        'I' : 1,
        'J' : 0
    }
    return H_dist[n]


Graph_nodes = {
    'A' : [('B', 6),('F', 3)],
    'B' : [('C', 3),('D', 2)],
    'C' : [('D', 1),('E', 5)],
    'D' : [('C', 1),('E', 8)],
    'E' : [('I', 5),('J', 5)],
    'F' : [('G', 1),('H', 7)],
    'G' : [('I', 3)],
    'H' : [('I', 2)],
    'I' : [('E', 5),('J', 3)],
}
aStarAlgo('A','J')
```

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

# Bit detailed :

def aStarAlgo(start_node, stop_node):: This defines a function called aStarAlgo that takes two parameters, start_node and stop_node, which represent the start and stop nodes for finding the shortest path.

open_set = set(start_node): Initializes an open set with the starting node. The open set is a set of nodes that need to be explored.

closed_set = set(): Initializes an empty closed set. The closed set contains nodes that have been explored.

parents = {}: Initializes an empty dictionary to keep track of the parent nodes of each node during the search.

g[start_node] = 0: Initializes a dictionary g with the cost from the start node to itself, which is 0.

parents[start_node] = start_node: Sets the parent of the start node to itself.

while len(open_set) > 0 :: Starts a while loop that continues as long as there are nodes in the open set to explore.

n = None: Initializes a variable n to None. This will be used to keep track of the node with the lowest cost.

for v in open_set:: Iterates over all nodes in the open set.

if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):: Checks if n is None or if the cost from the start to the current node v plus its heuristic is less than the cost from the start to n plus the heuristic of n. This is part of the A* algorithm's logic for selecting the node with the lowest estimated total cost.

n = v: If the above condition is true, updates n to the current node v.

if n == stop_node or Graph_nodes[n] == None:: Checks if n is the stop node or if there are no neighbors for n in the graph.

pass: If the above condition is true, it does nothing and continues to the next iteration.

else:: If n is not the stop node and it has neighbors, it proceeds to explore its neighbors.

for (m, weight) in get_neighbors(n):: Iterates over the neighbors of node n and their corresponding edge weights.

if m not in open_set and m not in closed_set:: Checks if the neighbor m is not in the open set or the closed set. If so, it means this is a newly discovered node.

open_set.add(m): Adds the neighbor m to the open set.

parents[m] = n: Sets the parent of m to be n.

g[m] = g[n] + weight: Updates the cost to reach m via n by adding the edge weight.

The code continues to update costs and explore nodes according to the A* algorithm logic.

if n == None:: After the while loop, if n is still None, it means there is no path from the start to the stop node.

if n == stop_node:: If n is the stop node, it means a path has been found.

path.reverse(): Reverses the path because it was built from the stop node to the start node, and we want it in the correct order.

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

open_set.remove(n): Removes n from the open set.

closed_set.add(n): Adds n to the closed set.

print('Path does not exist!'): If none of the conditions in the while loop is met, it means there is no path from the start to the stop node.

def get_neighbors(v):: Defines a function to get the neighbors of a node.

def heuristic(n):: Defines a heuristic function that estimates the cost from a node to the stop node.

H_dist = { ... }: Initializes a dictionary that contains heuristic values for each node in the graph.

return H_dist[n]: Returns the heuristic value for the given node n.

Graph_nodes = { ... }: Defines the graph with nodes and their neighbors and edge weights.

aStarAlgo('A', 'J'): Finally, it calls the aStarAlgo function with the start node 'A' and the stop node 'J' to find the shortest path in the graph.