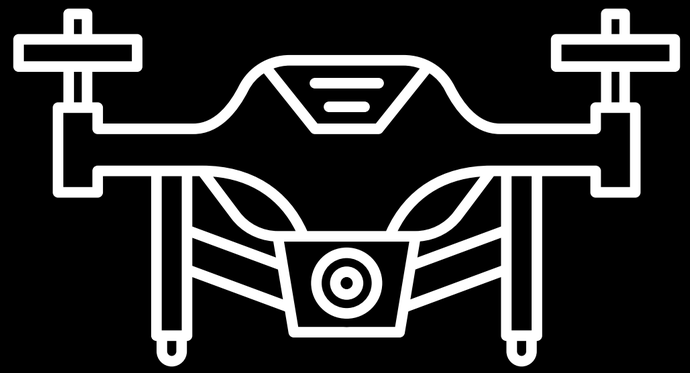
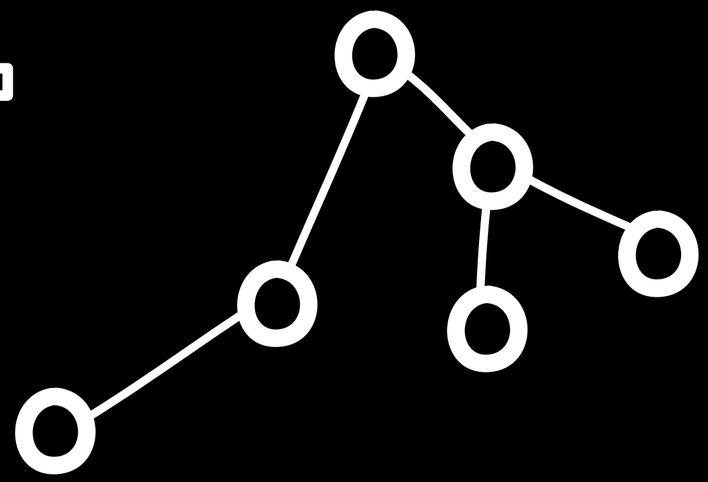
UAV
PATH PLANNING

Group Presentation





By Ayush , Priyanshu , Khushveer

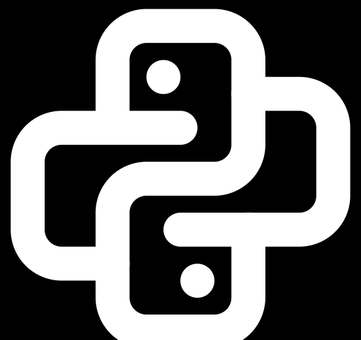
Essential Python Libraries

NumPy :

Numpy is a python library used for fast numerical computation and array based operations, it has functions in domain of linear algebra , fourier transforms and matrices

Pandas :

Pandas is a Python library built on top of NumPy, designed for data manipulation, analysis, and structured data handling. Provide DataFrame and series representation of tabular data.

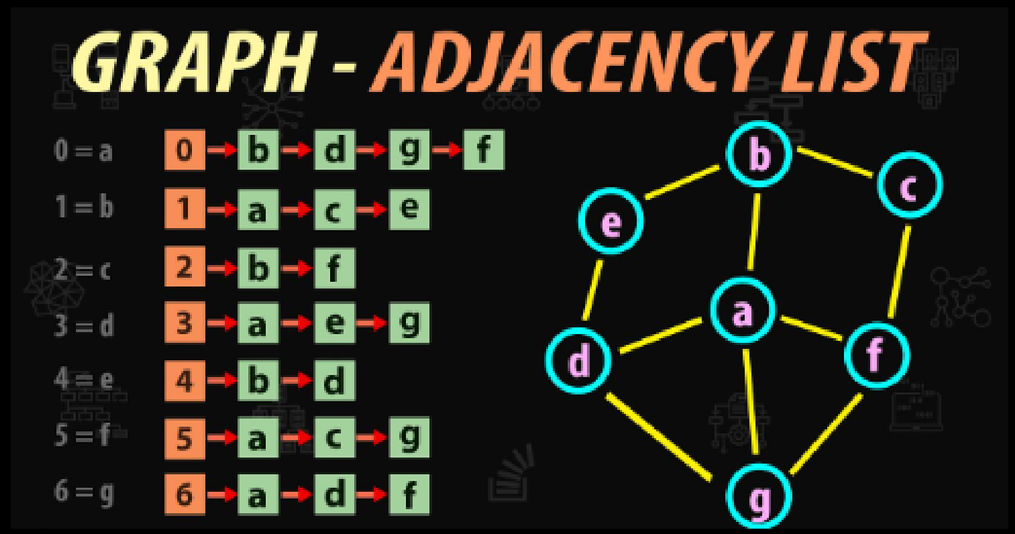


Matplotlib :

Matplotlib is a Python library for creating static and dynamic , interactive visualizations. It enables users to graphically represent data

Graphs !!!

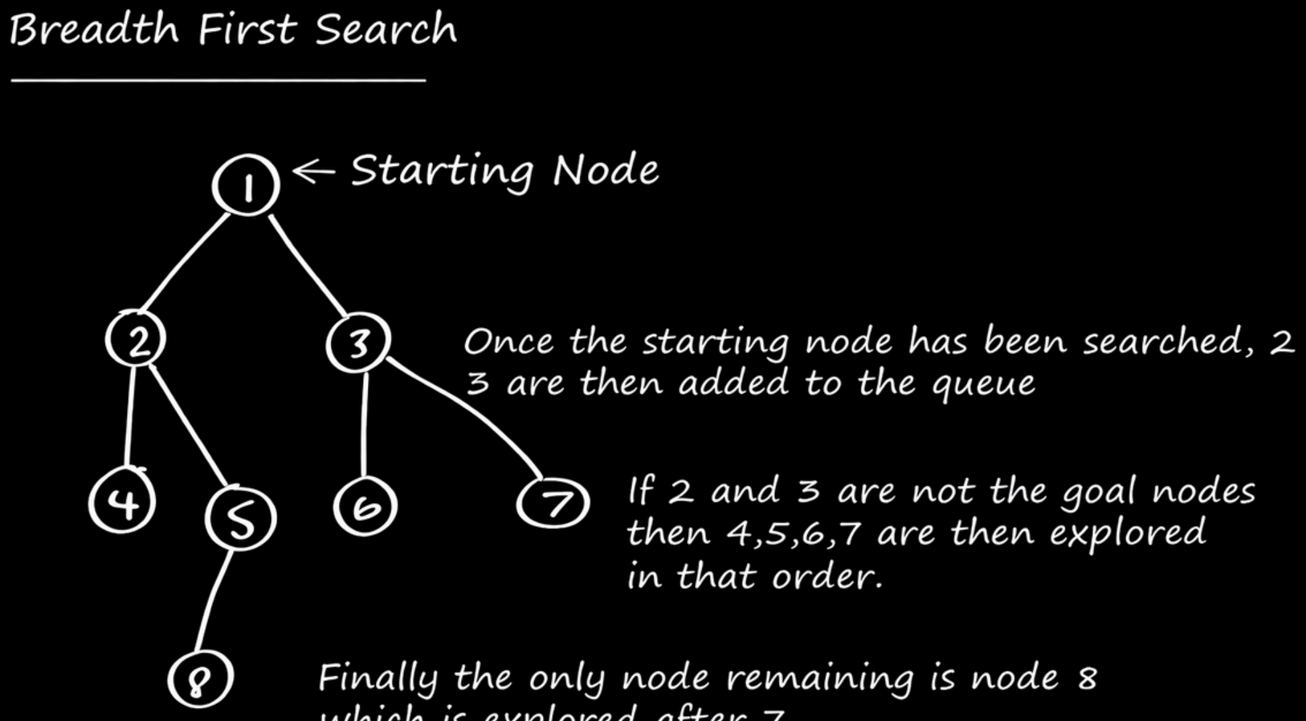
Graph Data Structure is a collection of nodes connected by edges. It's used to represent relationships between different entities. There are various ways to store graph , we will use the adjacency list representation



Graph Traversal

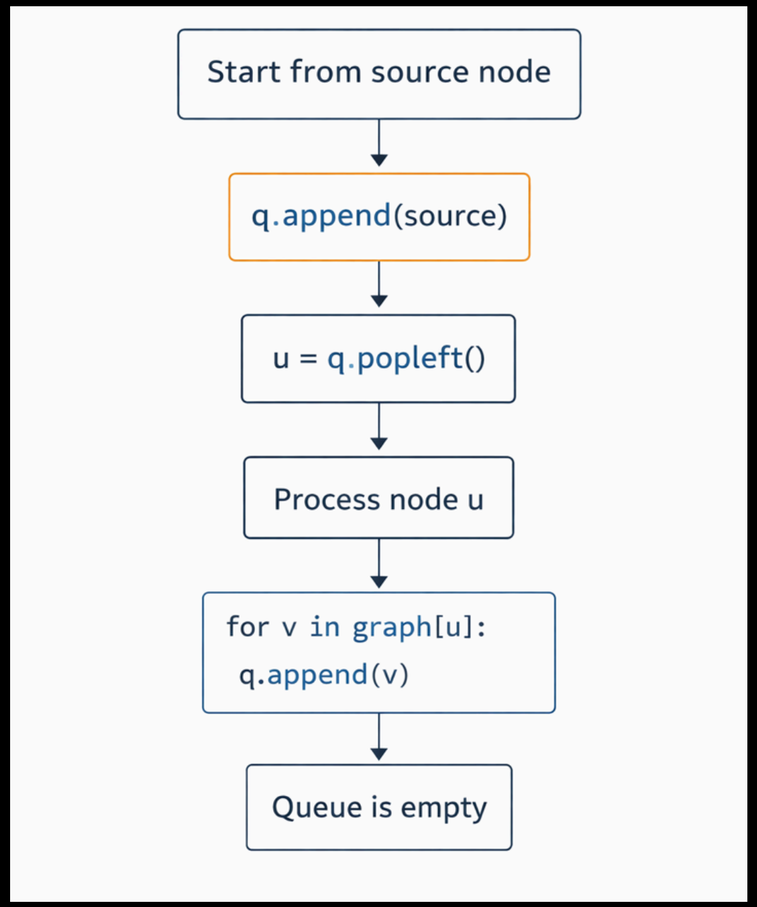
1.) BFS(Breadth First Search)

In Breadth first search algorithm we start the traversal from one source node and explores the graph level by level , when we are at a particular node we mark that node as visited and then explore all the unvisited child nodes of that node.



BFS CODE

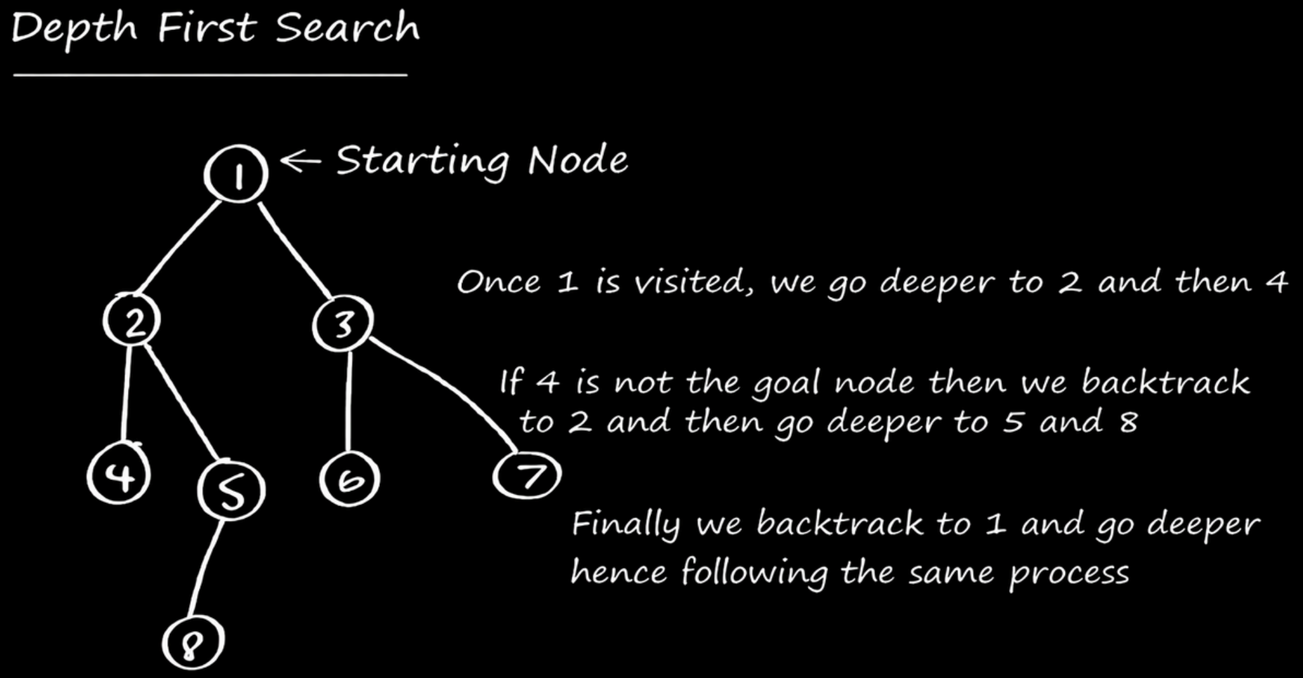
```python
from collections import deque

def bfs(graph, source):
    visited = set()
    parent = {}
    level = {}

    q = deque()

    visited.add(source)
    parent[source] = None
    level[source] = -1

    q.append(source)

    while q:
        u = q.popleft()

        for v in graph[u]:
            if v not in visited:
                visited.add(v)
                parent[v] = u
                level[v] = level[u] + 1
                q.append(v)

    return parent, level
```



Graph Traversal

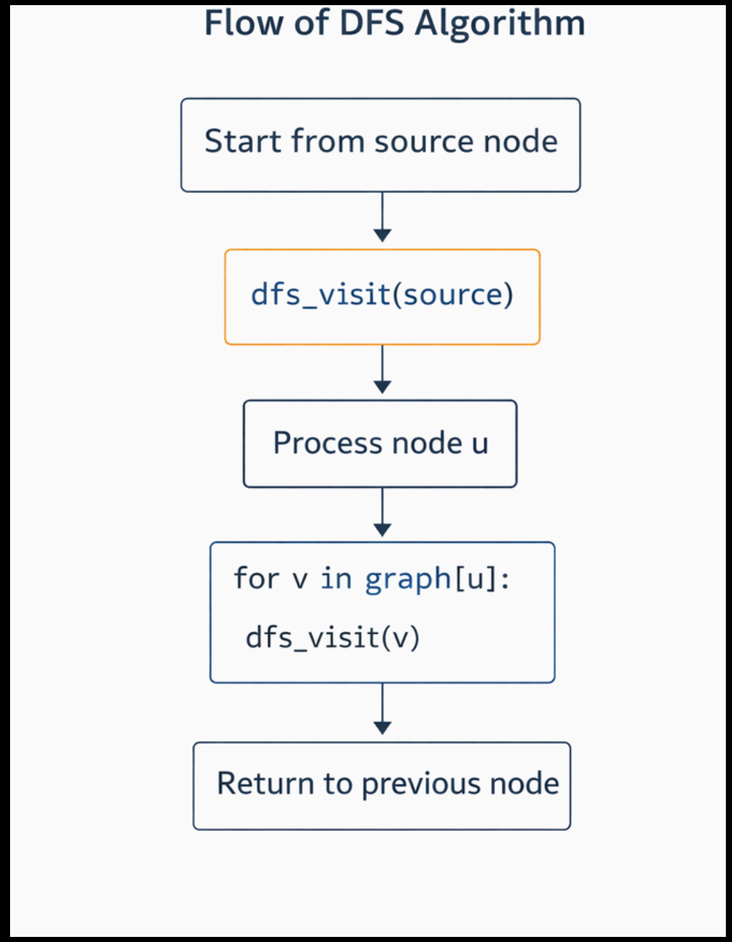
1.) DFS(Deapth First Search)

Depth First Search (DFS) is a graph traversal method that starts from a source vertex and explores each path completely before backtracking and exploring other paths This can be implemented using reccursion.



DFS CODE

```python
def dfs(graph, source):
    visited = set()

    def dfs_visit(u):
        visited.add(u)
        print(u, end=" ")

        for v in graph[u]:
            if v not in visited:
                dfs_visit(v)

    dfs_visit(source)
```

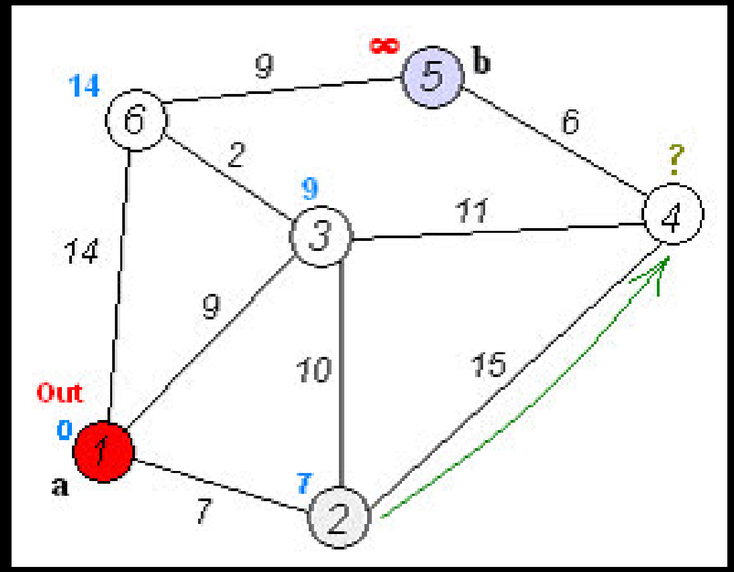


Dijkstra Algorithm

Dijkstra’s algorithm finds shortest paths from the starting node to all nodes of the graph, given there are no negative weight edges in the graph.

Dijkstra’s algorithm is efficient,
because it only processes each edge in the graph once, using the fact that there are no negative edges.

Dijkstra’s algorithm maintains distances to the nodes and reduces them during the search



Algorithm

For every vertex v:

- dist[v] ← ∞
- parent[v] ← None

Set source:

- dist[s] ← 0
- Create a min-priority queue Q
- Insert (0, s) into Q

While Q is not empty:

- Remove the node with smallest distance
- For each edge (u, v) with weight w
- Compute alternative smaller distance

When the queue is empty:

- dist[v] contains shortest distance from s to v
- parent[v] can be used to reconstruct paths

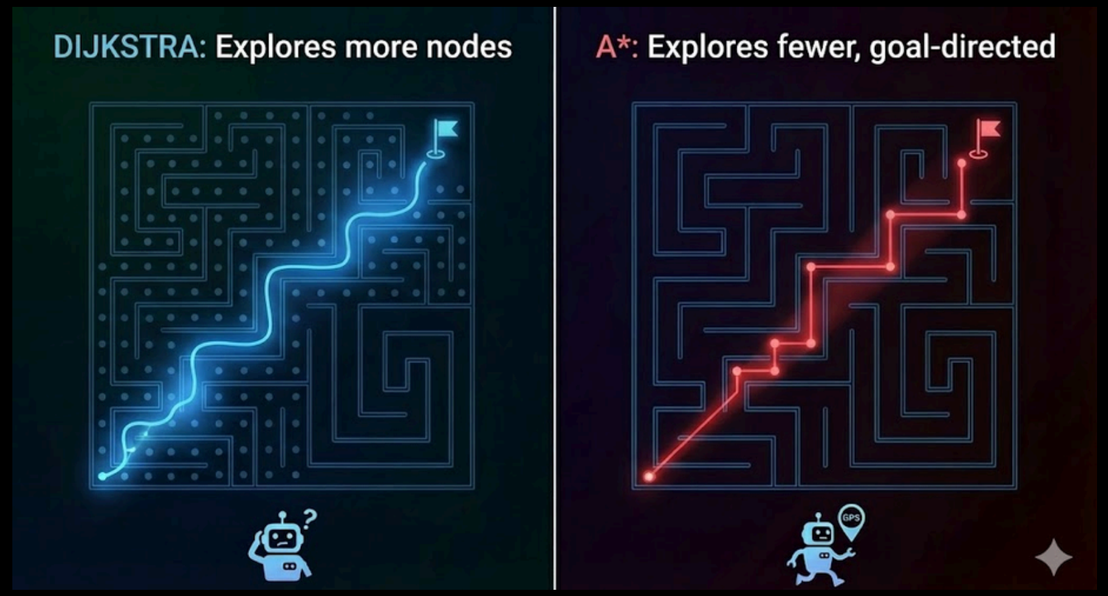
```
function Dijkstra(G, s):
    for v in V:
        dist[v] = ∞
        parent[v] = None

    dist[s] = 0
    Q = priority queue
    Q.push((0, s))

    while Q not empty:
        (d, u) = Q.pop_min()
        if d > dist[u]: continue

        for each (u, v, w) in E:
            alt = dist[u] + w
            if alt < dist[v]:
                dist[v] = alt
                parent[v] = u
                Q.push((alt, v))

    return dist, parent
```

Why A* Algorithm ?

Algorithms like Dijkstra find shortest paths, but they explore too many unnecessary nodes.

A* is needed because it is faster, smarter, and goal-directed

It uses a heuristic to estimate distance to the goal → saves time.



Algorithm :

A* Algorithm

Step-01 Define a list OPEN.
Initially, OPEN consists solely of a single node, the start node S.

Step-02 If the list is empty, return failure and exit.

Step-03 Remove node ***n*** with the smallest value of f(***n***) from OPEN and move it to list CLOSED.
If node ***n*** is a goal state, return success and exit.

Step-04 Expand node ***n***.

Step-05 If any successor to ***n*** is the goal node,
return success and the solution by tracing the path node to S.
Otherwise, go to Step-06.

Step-06 For each successor node,

- Apply the evaluation function f to the node.
- If the node has not been in either list, add it to OPEN.

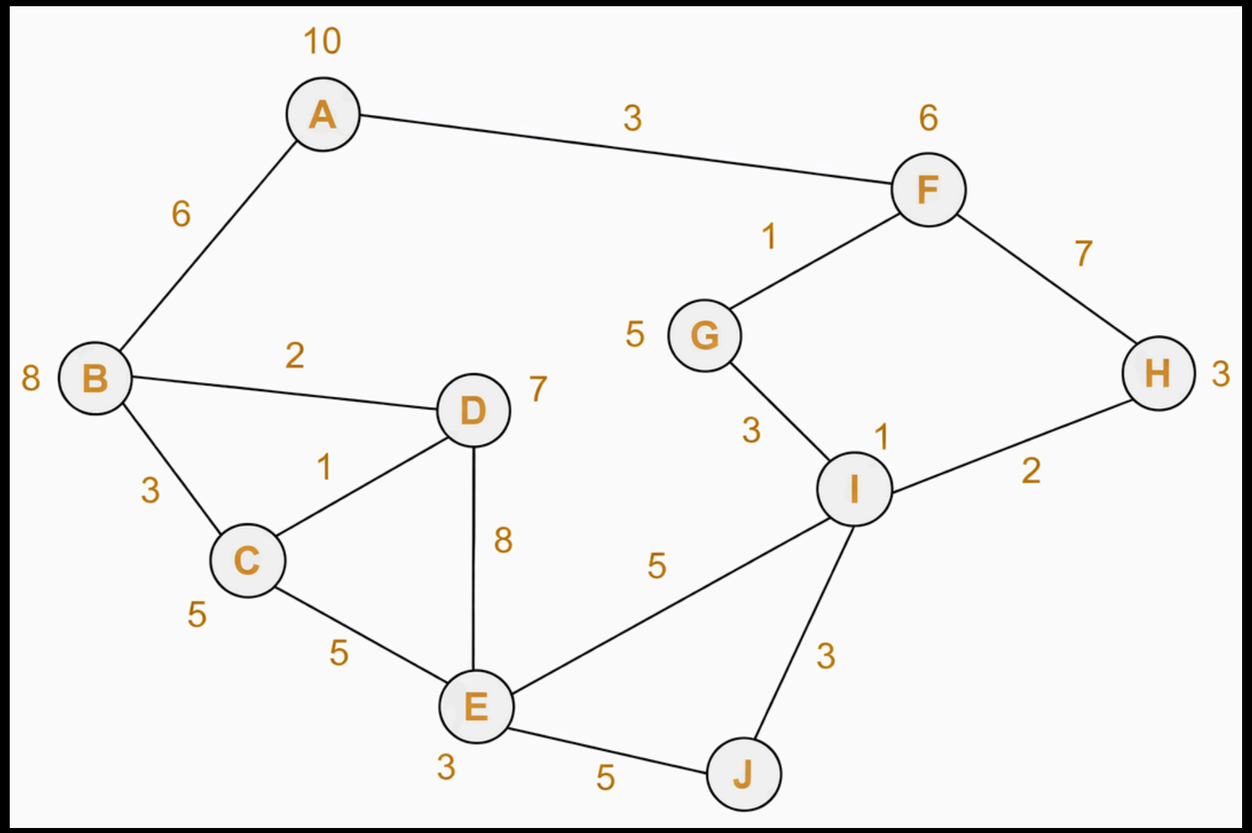
Step-07 Go back to Step-02.

Example :

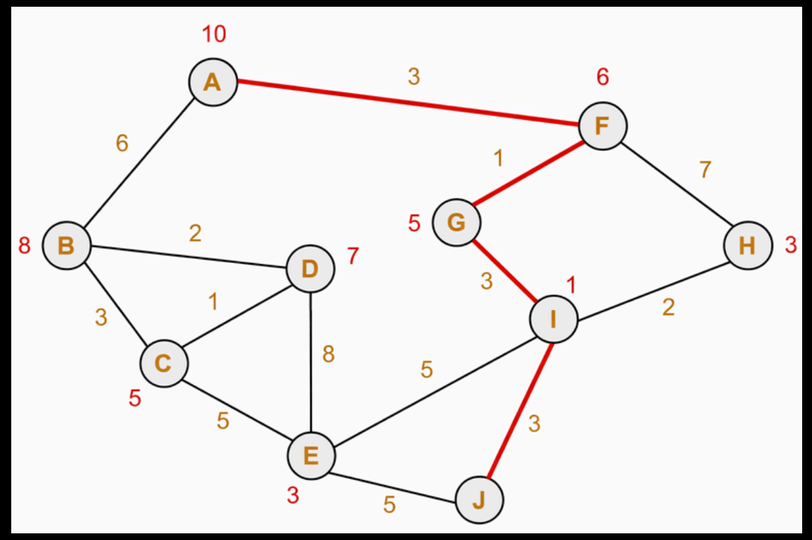
The numbers written on edges represent the distance between the nodes.

The numbers written on nodes represent the heuristic value.

Find the most cost-effective path to reach from start state A to final state J using A* Algorithm.



Solution :



A* Algorithm
(Shortest Path Example)

Step 01

We start with node **A**.

Node B and Node F can be reached from node **A**.

- f(B) = 6 + 8 = 14
- f(F) = 3 + 6 = 9

Since f(F) < f(B), so it decides to go to node **F**.

Path: A → F

Step 02

Node **G** and Node **H** can be reached from node **F**.

- f(G) = (3+1) + 5 = 9
- f(H) = (3+7) + 3 = 13

Since f(G) < f(H), so it decides to go to node **G**.

Path: A → F → G

Step 03

Node **I** can be reached from node **G**.

- f(I) = (3+1+3) + 1 = 8

It decides to go to node **I**.

Path: A → F → G → I → J

Step 04

Node **E**, Node **H** and Node **J** can be reached from node **I**.

- f(E) = (3+1+3+5) + 3 = 15
- f(H) = (3+1+3+2) + 3 = 12
- f(J) = (3+1+3+3) + 0 = 10

This is the required shortest path from node **A** to node **J**.

Pseudo Code :

```
def A_star(start, goal):

    open_list = [start]
    closed_list = []

    g[start] = 0
    f[start] = g[start] + h(start)

    while open_list is not empty:

        # pick node with smallest f-value
        current = node in open_list with lowest f

        if current == goal:
            return reconstructed_path(current)

        open_list.remove(current)
        closed_list.append(current)
```

```
    for neighbor in neighbors(current):

        tentative_g = g[current] + cost(current, neighbor)

        if neighbor in closed_list and tentative_g >= g[neighbor]
            continue

        if neighbor not in open_list or tentative_g < g[neighbor]

            parent[neighbor] = current
            g[neighbor] = tentative_g
            f[neighbor] = g[neighbor] + h(neighbor)

            if neighbor not in open_list:
                open_list.append(neighbor)

eturn "No path found"
```

Thank You