Graphs

Depth First & Breadth First Search

Graph Traversal: BFS and DFS

- Graph traversal refers to the process of visiting all the nodes (or vertices) in a graph. Two common methods for graph traversal are
  - Breadth-First Search (BFS) and
  - Depth-First Search (DFS).
- Each method explores the graph in a different manner and has unique characteristics and use cases.

Breadth-First Search (BFS)

- BFS explores the graph level by level. It starts at a given node, explores all its neighboring nodes before moving on to the neighbors of those neighbors. BFS is particularly useful for finding the shortest path in unweighted graphs.

BFS - Algorithm

Algorithm:

- Start from the given source node and mark it as visited.
- Create a queue and enqueue the source node.
- While the queue is not empty:
  - Dequeue a node from the queue.
  - Visit all unvisited neighbors of the dequeued node, mark them as visited, and enqueue them.

Applications

1. Finding the shortest path in unweighted graphs.
2. Solving puzzles and games that involve finding the shortest sequence of moves.
3. Crawling the web (breadth-first web crawlers).

Depth-First Search (DFS)

- DFS explores as far as possible along each branch before backtracking. It starts at a given node and goes as deep as possible in one direction before retracing steps to explore other branches. DFS is implemented using recursion or a stack.

Algorithm

1. Start from the given source node and mark it as visited.
2. For each unvisited neighbor of the current node:

   Recursively apply DFS on the neighbor.
3. Continue this process until all nodes are visited.

Applications

1. Solving problems that require exploring all possibilities, such as maze solving.
2. Topological sorting in directed acyclic graphs.
3. Detecting cycles in a graph.

DFS – Non Recursive Method

```cpp
void DFS(int start, vector<int> adj[], int V)
{
    vector<bool> visited(V, false);
    stack<int> s;

    s.push(start);

    while (!s.empty())
    {
        int u = s.top();
        s.pop();

        if (!visited[u])
        {
            cout << u << " ";
            visited[u] = true;
        }

        // Push all unvisited neighbors into the stack
        for (int i = adj[u].size() - 1; i >= 0; i--)
        {
            if (!visited[adj[u][i]])
            {
                s.push(adj[u][i]);
            }
        }
    }
}
```

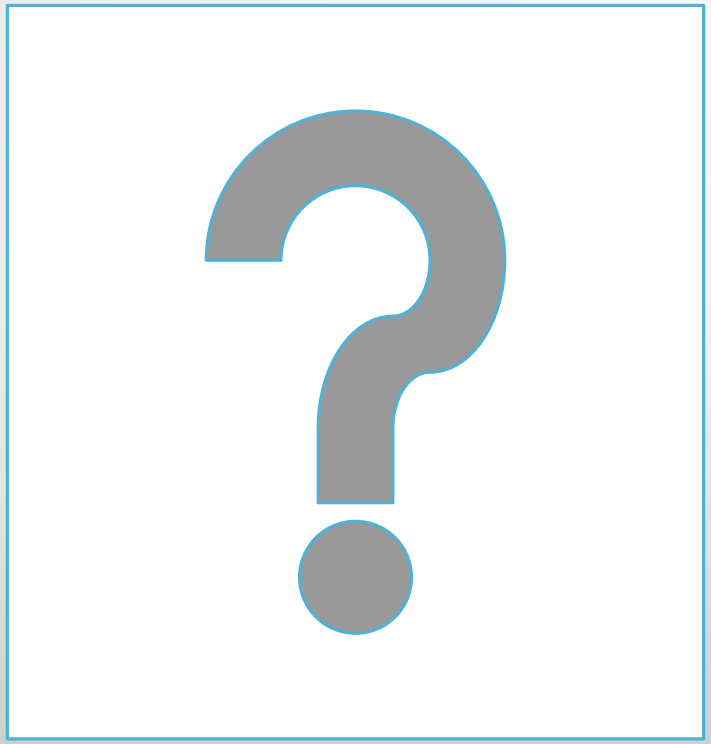
BFS – Non Recursive Method

```
void BFS(int start, vector<int> adj[], int V)
{
    vector<bool> visited(V, false);
    queue<int> q;

    visited[start] = true;
    q.push(start);

    while (!q.empty())
    {
        int u = q.front();
        cout << u << " ";
        q.pop();

        for (int i = 0; i < adj[u].size(); i++)
        {
            if (!visited[adj[u][i]])
            {
                visited[adj[u][i]] = true;
                q.push(adj[u][i]);
            }
        }
    }
}
```

DFS – Recursive Method

```
void DFSUtil(int u, vector<int> adj[],
                    vector<bool> &visited)
{
    visited[u] = true;
    cout << u << " ";
    for (int i=0; i<adj[u].size(); i++)
        if (visited[adj[u][i]] == false)
            DFSUtil(adj[u][i], adj, visited);
}

// This function does DFSUtil() for all
// unvisited vertices.
void DFS(vector<int> adj[], int V)
{
    vector<bool> visited(V, false);
    for (int u=0; u<V; u++)
        if (visited[u] == false)
            DFSUtil(u, adj, visited);
}
```

BFS – Recursive Method



Thank You

Live To Code, Code To Live