Graph traversal algorithms

Traversing a graph means examining the nodes and edges of the graph. There are two standard methods of graph traversal which we will discuss in this section. These two methods are -

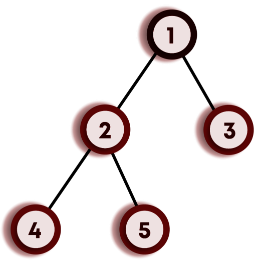
a) Depth-first search
b) Breadth-first search.

While breadth-first search uses a **queue** as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a **stack**. But both these algorithms make use of a bool variable **VISITED**. During the execution of the algorithm, every node in the graph will have the variable VISITED set to **false** or **true**, depending on its **current state,** whether the node has been processed/visited or not.

Depth-first search (DFS)

The **depth-first search(DFS)** algorithm, as the name suggests, first goes into the depth and then recursively does the same in other directions, it progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
In other words, the depth-first search begins at a starting node A which becomes the current node. Then, it examines each node along with a path P which begins at A. That is, we process a neighbor of A, then a neighbor of the processed node, and so on. During the execution of the algorithm, if we reach a path that has a node that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.

For example: DFS for the below graph is:



DFS Traversal : 1->2->4->5->3

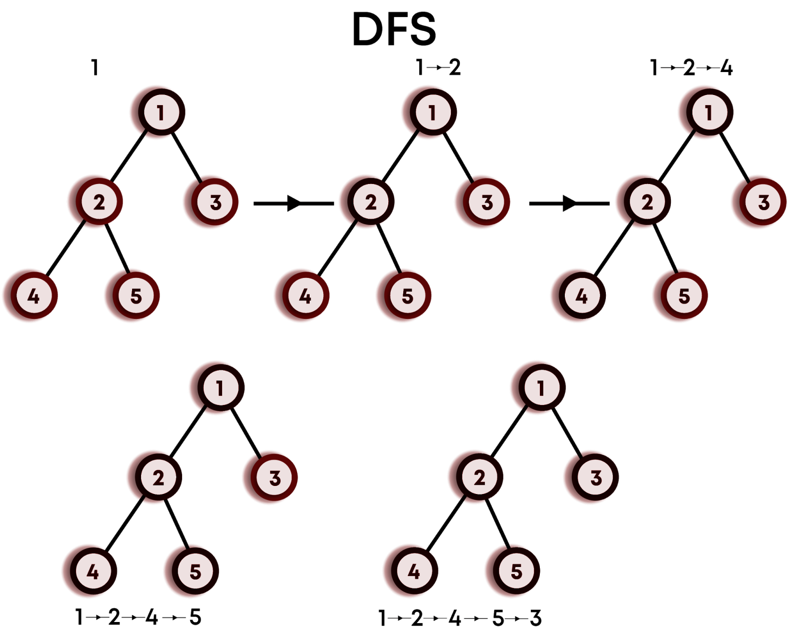
Other possible DFS traversals for the above graph can be:

1. 1->3->2->4->5

2. 1->2->5->4->3
3. 1->3->2->5->4

i.

We can clearly observe that there can be more than one DFS traversals for the same graph.



Implementation of DFS (Iterative) :

```
function DFS_iterative(graph,source)

/*
Let St be a stack, pushing source vertex in the stack.
St represents the vertices that have been processed/visited
so far.
*/

        St.push(source)

        //   Mark source vertex as visited.
        visited[source] = true

        //   Iterate through the vertices present in the stack

        while St is not empty
                //   Pop a vertex from the stack to visit its neighbors
                cur = St.top()
                St.pop()
```

```
            /*
Push all the neighbors of the cur vertex that have not been visited yet, push them into
the stack and mark them as visited.
            */

            for all neighbors v of cur in graph:
                  if visited[v] is false
                        St.push(v)
                        visited[v] = true

      return
```

Implementation of DFS (Recursive)

```
function DFS_recursive(graph,cur)

      //   Mark the cur vertex as visited.
      visited[cur] = true

      /*
Recur for all the neighbors of the cur vertex that have not been visited yet.
      */

      for all neighbors v of cur in graph:
            if visited[v] is false
                  DFS_recursive(graph,v)
      return
```

Features of Depth-First Search Algorithm

- **Time Complexity:** The time complexity of a depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as **$(O(|V| + |E|))$**, where **$|V|$** is the number of vertices and **$|E|$** is the number of edges in the graph considering the graph is represented by adjacency list.
- **Completeness:** Depth-first search is said to be a **complete** algorithm in the case of a finite graph. If there is a solution, a depth-first search will find it regardless of the kind of graph. But in the case of an infinite graph, where there is no possible solution, it will diverge.

Applications of Depth-First Search Algorithm

- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph