DFS (Depth First Search) algorithm

We will discuss the DFS algorithm in the data structure. It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

The step by step process to implement the DFS traversal is given as follows -

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Applications of DFS algorithm

The applications of using the DFS algorithm are given as follows -

- DFS algorithm can be used to implement the topological sorting.
- It can be used to find the paths between two vertices.
- It can also be used to detect cycles in the graph.
- DFS algorithm is also used for one solution puzzles.
- DFS is used to determine if a graph is bipartite or not.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

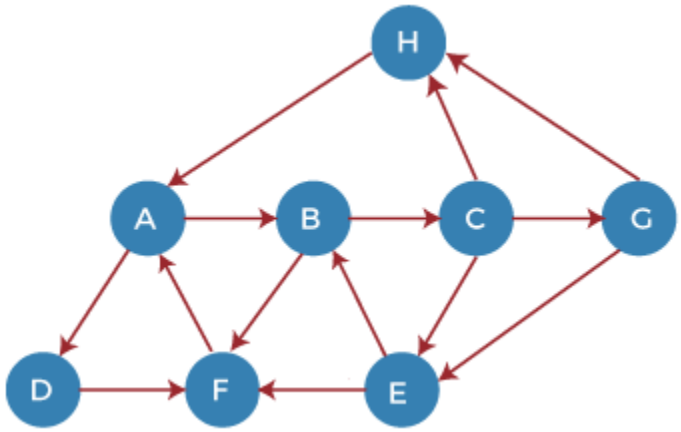
Step 6: EXIT

Pseudocode

```
1.  DFS(G,v)   ( v is the vertex where the search starts )
2.        Stack S := {};   ( start with an empty stack )
3.        for each vertex u, set visited[u] := false;
4.        push S, v;
5.        while (S is not empty) do
6.          u := pop S;
7.          if (not visited[u]) then
8.            visited[u] := true;
9.            for each unvisited neighbour w of uu
10.             push S, w;
11.         end if
12.       end while
13.     END DFS()
```

Example of DFS algorithm

Now, let's understand the working of the DFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Now, let's start examining the graph starting from Node H.

Step 1 - First, push H onto the stack.

1. STACK: H

Step 2 - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.

1. Print: H]STACK: A

Step 3 - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.

1. Print: A

2. STACK: B, D

Step 4 - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.

1. Print: D
2. STACK: B, F

Step 5 - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.

1. Print: F
2. STACK: B

Step 6 - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.

1. Print: B
2. STACK: C

Step 7 - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.

1. Print: C
2. STACK: E, G

Step 8 - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.

1. Print: G
2. STACK: E

Step 9 - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.

1. Print: E
2. STACK:

Now, all the graph nodes have been traversed, and the stack is empty.

Complexity of Depth-first search algorithm

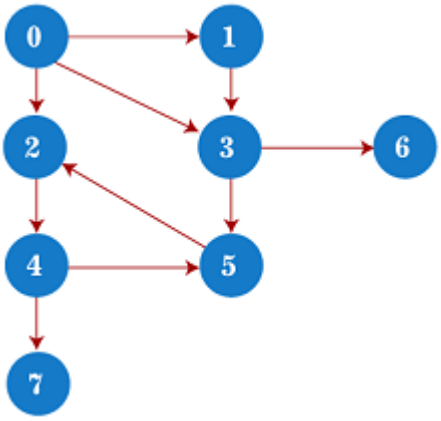
The time complexity of the DFS algorithm is **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.

The space complexity of the DFS algorithm is O(V).

Implementation of DFS algorithm

Now, let's see the implementation of DFS algorithm in Java.

In this example, the graph that we are using to demonstrate the code is given as follows -




```
1.  /*A sample java program to implement the DFS algorithm*/
2.
3.  import java.util.*;
4.
5.  class DFSTraversal {
6.    private LinkedList<Integer> adj[]; /*adjacency list representation*/
7.    private boolean visited[];
8.
9.    /* Creation of the graph */
10.  DFSTraversal(int V) /*'V' is the number of vertices in the graph*/
11.  {
12.    adj = new LinkedList[V];
13.    visited = new boolean[V];
14.
15.    for (int i = 0; i < V; i++)
16.      adj[i] = new LinkedList<Integer>();
17.  }
18.
19.  /* Adding an edge to the graph */
20.  void insertEdge(int src, int dest) {
21.    adj[src].add(dest);
22.  }
23.
24.  void DFS(int vertex) {
25.    visited[vertex] = true; /*Mark the current node as visited*/
26.    System.out.print(vertex + " ");
27.
28.    Iterator<Integer> it = adj[vertex].listIterator();
29.    while (it.hasNext()) {
30.      int n = it.next();
31.      if (!visited[n])
32.        DFS(n);
33.    }
34.  }
35.
```

```
36.   public static void main(String args[]) {
37.     DFSTraversal graph = new DFSTraversal(8);
38.
39.         graph.insertEdge(0, 1);
40.         graph.insertEdge(0, 2);
41.         graph.insertEdge(0, 3);
42.         graph.insertEdge(1, 3);
43.         graph.insertEdge(2, 4);
44.         graph.insertEdge(3, 5);
45.         graph.insertEdge(3, 6);
46.         graph.insertEdge(4, 7);
47.         graph.insertEdge(4, 5);
48.         graph.insertEdge(5, 2);
49.
50.         System.out.println("Depth First Traversal for the graph is:");
51.         graph.DFS(0);
52.   }
53. }
```

Output