**Depth First Search (DFS) Algorithm**

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

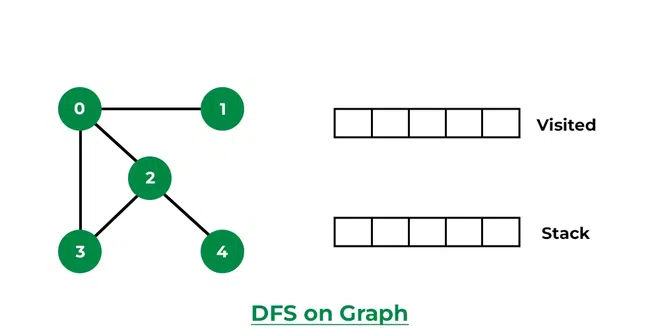
This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:  
Pick a starting node and push all its adjacent nodes into a stack.  
Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.  
Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

## **How does DFS work?**

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

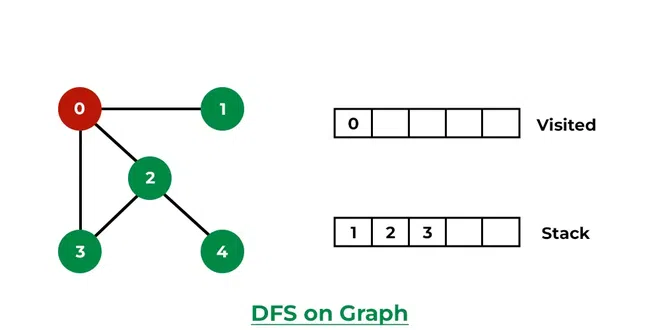
Let us understand the working of **Depth First Search** with the help of the following illustration:

**Step 1: Initially stack and visited arrays are empty.**

****

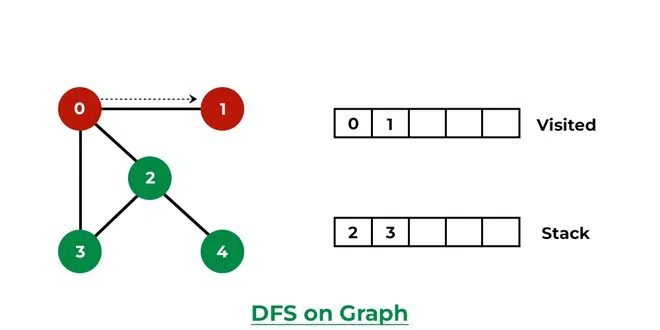
***Stack and visited arrays are empty initially.***

**Step 2: Visit 0 and put its adjacent nodes which are not visited yet into the stack.**

****

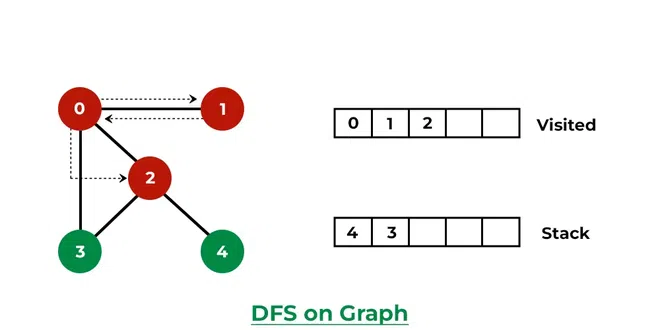
*Visit node 0 and put its adjacent nodes (1, 2, 3) into the stack*

**Step 3: Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.**

****

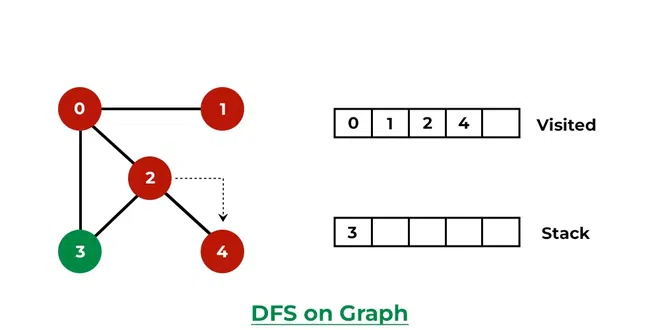
*Visit node 1*

***Step 4:****Now,**Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited (i.e, 3, 4) in the stack.*



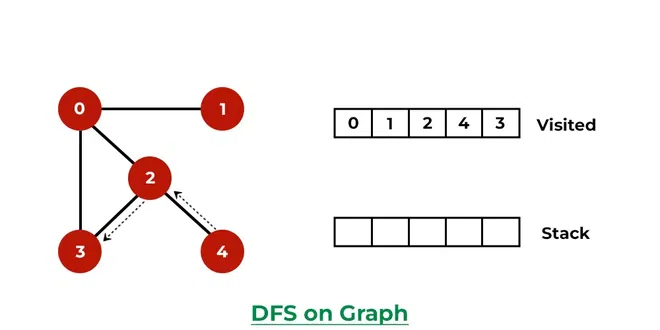
*Visit node 2 and put its unvisited adjacent nodes (3, 4) into the stack*

**Step 5: Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.**

****

*Visit node 4*

**Step 6: Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.**

****

*Visit node 3*

*Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.*

Below is the implementation of the above approach:

# Python3 program to print DFS traversal

# from a given  graph

from collections import defaultdict

# This class represents a directed graph using

# adjacency list representation

class Graph:

    # Constructor

    def \_\_init\_\_(self):

        # Default dictionary to store graph

        self.graph = defaultdict(list)

    # Function to add an edge to graph

    def addEdge(self, u, v):

        self.graph[u].append(v)

    # A function used by DFS

    def DFSUtil(self, v, visited):

        # Mark the current node as visited

        # and print it

        visited.add(v)

        print(v, end=' ')

        # Recur for all the vertices

        # adjacent to this vertex

        for neighbour in self.graph[v]:

            if neighbour not in visited:

                self.DFSUtil(neighbour, visited)

    # The function to do DFS traversal. It uses

    # recursive DFSUtil()

    def DFS(self, v):

        # Create a set to store visited vertices

        visited = set()

        # Call the recursive helper function

        # to print DFS traversal

        self.DFSUtil(v, visited)

# Driver's code

if \_\_name\_\_ == "\_\_main\_\_":

    g = Graph()

    g.addEdge(0, 1)

    g.addEdge(0, 2)

    g.addEdge(1, 2)

    g.addEdge(2, 0)

    g.addEdge(2, 3)

    g.addEdge(3, 3)

    print("Following is Depth First Traversal (starting from vertex 2)")

    # Function call

    g.DFS(2)

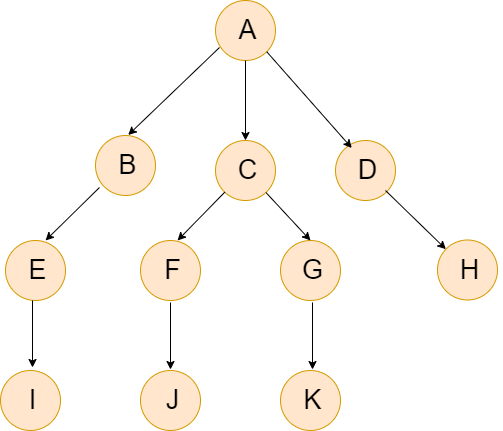
**Output**

Following is Depth First Traversal (starting from vertex 2)

2 0 1 3

### **Implementing Depth First Search (A non-recursive approach)**

Let's consider the following graph for the DFS implementation.



Let's define the graph as an adjacency list using the Python Dictionary.

|  |
| --- |
| 1. graph = {"A":["B","C","D"], 2. "B":["E"], 3. "C":["G","F"], 4. "D":["H"], 5. "E":["I"], 6. "F":["J"], 7. "G":["K"]} |

We can implement the DFS both recursion technique and non-recursion technique, iterative approach.

In this section, we will understand the iterative approach.

We will use a stack and a list to keep track of the visited nodes.

* First, we will visit to the root node and mark it as visited. Then, we will move towards all of its neighbors to the stack.
* At each step, we will pop out an item from the stack and check if it has been visited.
* If the node is not visited, it will be added to the path and all of its neighbors to the stack.

## **DFS Pseudocode**

Below is the Pseudocode of the DFS. In the **init()** function, we run the DFS function on every node because most of the times, a graph may contain two different disconnect part. So it makes sure that we cover every vertex, can also run DFS algorithm on every node.

|  |
| --- |
| 1. DFS(G, u) 2. u.visited = true 3. **for** each v ∈ G.Adj[u] 4. **if** v.visited == false 5. DFS(G,v) 7. init() { 8. For each u ∈ G 9. u.visited = false 10. For each u ∈ G 11. DFS(G, u) 12. } |

Let's implement the DFS using the Python code.

|  |
| --- |
| **Example -1**   1. **def** non\_recursive\_dfs(graph, source): 2. **if** source **is** None **or** source **not** **in** graph: 3. **return** "Please Enter Valid input" 4. path = [] 5. stack\_val = [source] 7. **while**(len(stack) != 0): 9. s =stack\_val.pop() 11. **if** s **not** **in** path: 12. path.append(s) 13. **if** s **not** **in** graph: 14. #leaf node 15. **continue** 17. **for** neighbor\_node **in** graph[s]: 18. stack\_val.append(neighbor\_node) 19. **return** " ".join(path) |

**Output:**

A D H C F J G K B E I

The order of the traversal of the graph is in the 'Depth First' manner.

## **DFS using a Recursive Method**

The recursive is a popular problem solving approach in which the same problem is divided into smaller instances. We will define the base case inside our program. Let's understand the below example.

|  |
| --- |
| **Example -2**   1. **def** dfs\_recursive(graph, source,path = []): 3. **if** source **not** **in** path: 4. path.append(source) 6. **if** source **not** **in** graph: 7. # leaf node, backtrack 8. **return** path 10. **for** neighbour **in** graph[source]: 12. path = dfs\_recursive(graph, neighbour, path)  15. **return** path 17. graph = {"A":["B","C","D"], 18. "B":["E"], 19. "C":["G","F"], 20. "D":["H"], 21. "E":["I"], 22. "F":["J"], 23. "G":["K"]} 24. dfs\_element = dfs\_recursive(graph, "A") 25. **print**(dfs\_element) |

**Output:**

['A', 'B', 'E', 'I', 'C', 'G', 'K', 'F', 'J', 'D', 'H']

The order of traversal is again in the Depth-first manner.

## **Complexity**

The time complexity of the DFS is represented as O(V+E), where V shows the number of nodes and E is the number of edges. The space complexity is 0(V).

## **Application of Algorithm**

The following are the real-life application of DFS.

* It is used to find the path.
* It is used to test if graph is bipartite.
* It is used to finding the strongly connected components of a graph.
* It is used for detecting cycle in a graph.
* It is used for scheduling the problem.
* It is used in topological sorting.

## **Conclusion**

This tutorial includes the concept of Depth-First Search in Python which is used to traverse the graph or tree. We have discussed both recursive and non-recursive methods to implement DFS in Python. We have also defined how to represent graph in Python.