Depth First Search or DFS for a Graph - GeeksforGeeks

Source: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Depth First Search or DFS for a Graph Last Updated : 25 Oct, 2025 Given a graph, traverse the graph using Depth First Search and find the order in which nodes are visited. 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. To avoid revisiting nodes in graphs with cycles, a visited array is used to track visited vertices. Note: There can be multiple DFS traversals of a graph according to the order in which we pick adjacent vertices. Here we pick vertices as per the insertion order. Example: Input: adj[][] = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]] Output: [0, 1, 2, 3, 4] Explanation: The source vertex is 0. We visit it first, then we visit its adjacent. Start at 0: Mark as visited. Print 0. Move to 1: Mark as visited. Print 1. Move to 2: Mark as visited. Print 2. Move to 3: Mark as visited. Print 3.(backtrack to 2) Move to 4: Mark as visited. Print 4(backtrack to 2, then backtrack to 1, then to 0). Note that there can be more than one DFS Traversals of a Graph. For example, after 1, we may pick adjacent 2 instead of 0 and get a different DFS. Input: adj[][] = [[2, 3], [2], [0, 1], [0], [5], [4]] Output: [0, 2, 1, 3, 4, 5] Explanation: DFS Steps: Start at 0: Mark as visited. Print 0. Move to 2: Mark as visited. Print 2. Move to 1: Mark as visited. Print 1 (backtrack to 2, then backtrack to 0). Move to 3: Mark as visited. Print 3 (backtrack to 0). Start with 4. Start at 4: Mark as visited. Print 4. Move to 5: Mark as visited. Print 5. (backtrack to 4) Try it on GfG Practice DFS from a Given Source of Graph: Depth First Search (DFS) starts from a given source vertex and explores one path as deeply as possible. When it reaches a vertex with no unvisited neighbors, it backtracks to the previous vertex to explore other unvisited paths. This continues until all vertices reachable from the source are visited. In a graph, there might be loops. So we use an extra visited array to make sure that we do not process a vertex again. Let us understand the working of Depth First Search with the help of the following Illustration : C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends void dfsRec (vector < vector < int >> & adj , vector < bool > & visited , int s , vector < int > & res) { visited [s] = true ; res . push_back (s); // Recursively visit all adjacent vertices // that are not visited yet for (int i : adj [s]) if (visited [i] == false) dfsRec (adj , visited , i , res); } vector < int > dfs (vector < vector < int >> & adj) { vector < bool > visited (adj . size (), false); vector < int > res ; dfsRec (adj , visited , 0 , res); return res ; } //Driver Code Starts void addEdge (vector < vector < int >>& adj , int u , int v) { adj [u]. push_back (v); adj [v]. push_back (u); } int main () { int V = 5 ; vector < vector < int >> adj (V); // creating adjacency list addEdge (adj , 1 , 2); addEdge (adj , 1 , 0); addEdge (adj , 2 , 0); addEdge (adj , 2 , 3); addEdge (adj , 2 , 4); // Perform DFS starting from the source vertex 0 vector < int > res = dfs (adj); for (int i = 0 ; i < V ; i ++) cout << res [i] << " " ; } //Driver Code Ends C //Driver Code Starts #include <stdio.h> #define V 5 //Driver Code Ends void dfsRec (int adj [V][V], int visited [V], int s , int res [V], int * idx) { visited [s] = 1 ; res [(* idx) ++] = s ; // Recursively visit all adjacent vertices // that are not visited yet for (int i = 0 ; i < V ; i ++) { if (adj [s][i] && visited [i] == 0) dfsRec (adj , visited , i , res , idx); } } void dfs (int adj [V][V], int res [V]) { int visited [V] = { 0 }; int idx = 0 ; dfsRec (adj , visited , 0 , res , & idx); } //Driver Code Starts void addEdge (int adj [V][V], int u , int v) { adj [u][v] = 1 ; adj [v][u] = 1 ; // undirected } int main () { int adj [V][V] = { 0 }; // creating adjacency list addEdge (adj , 1 , 2); addEdge (adj , 1 , 0); addEdge (adj , 2 , 0); addEdge (adj , 2 , 3); addEdge (adj , 2 , 4); int res [V]; // Perform DFS starting from the source vertex 0 dfs (adj , res); for (int i = 0 ; i < V ; i ++) printf (

"%d " , res [i]); return 0 ; } //Driver Code Ends Java //Driver Code Starts import java.util.ArrayList ; class GFG { //Driver Code Ends static void dfsRec (ArrayList < ArrayList < Integer >> adj , boolean [] visited , int s , ArrayList < Integer > res) { visited [s] = true ; res . add (s); // Recursively visit all adjacent vertices // that are not visited yet for (int i : adj . get (s)) { if (! visited [i]) { dfsRec (adj , visited , i , res); } } } static ArrayList < Integer > dfs (ArrayList < ArrayList < Integer >> adj) { boolean [] visited = new boolean [adj . size ()] ; ArrayList < Integer > res = new ArrayList <> (); dfsRec (adj , visited , 0 , res); return res ; } //Driver Code Starts static void addEdge (ArrayList < ArrayList < Integer >> adj , int u , int v) { adj . get (u). add (v); adj . get (v). add (u); } public static void main (String [] args) { int V = 5 ; ArrayList < ArrayList < Integer >> adj = new ArrayList <> (); // creating adjacency list for (int i = 0 ; i < V ; i ++) adj . add (new ArrayList <> ()); addEdge (adj , 1 , 2); addEdge (adj , 1 , 0); addEdge (adj , 2 , 0); addEdge (adj , 2 , 3); addEdge (adj , 2 , 4); // Perform DFS starting from vertex 0 ArrayList < Integer > res = dfs (adj); for (int i = 0 ; i < res . size (); i ++) { System . out . print (res . get (i) + " "); } } } //Driver Code Ends Python def dfsRec (adj , visited , s , res): visited [s] = True res . append (s) # Recursively visit all adjacent vertices # that are not visited yet for i in adj [s]: if not visited [i]: dfsRec (adj , visited , i , res) def dfs (adj): visited = [False] * len (adj) res = [] dfsRec (adj , visited , 0 , res) return res #Driver Code Starts def addEdge (adj , u , v): adj [u] . append (v) adj [v] . append (u) if __name__ == "__main__" : V = 5 adj = [] # creating adjacency list for i in range (V): adj . append ([]) addEdge (adj , 1 , 2) addEdge (adj , 1 , 0) addEdge (adj , 2 , 0) addEdge (adj , 2 , 3) addEdge (adj , 2 , 4) # Perform DFS starting from vertex 0 res = dfs (adj) for node in res : print (node , end = " ") #Driver Code Ends C# //Driver Code Starts using System ; using System.Collections.Generic ; class GFG { //Driver Code Ends static void dfsRec (List < List < int >> adj , bool [] visited , int s , List < int > res) { visited [s] = true ; res . Add (s); // Recursively visit all adjacent vertices // that are not visited yet foreach (int i in adj [s]) { if (! visited [i]) dfsRec (adj , visited , i , res); } } static List < int > dfs (List < List < int >> adj) { bool [] visited = new bool [adj . Count]; List < int > res = new List < int > (); dfsRec (adj , visited , 0 , res); return res ; } //Driver Code Starts static void addEdge (List < List < int >> adj , int u , int v) { adj [u]. Add (v); adj [v]. Add (u); } static void Main () { int V = 5 ; List < List < int >> adj = new List < List < int >> (); // creating adjacency list for (int i = 0 ; i < V ; i ++) adj . Add (new List < int > ()); addEdge (adj , 1 , 2); addEdge (adj , 1 , 0); addEdge (adj , 2 , 0); addEdge (adj , 2 , 3); addEdge (adj , 2 , 4); // Perform DFS starting from vertex 0 List < int > res = dfs (adj); foreach (int i in res) Console . Write (i + " "); } } //Driver Code Ends JavaScript function dfsRec (adj , visited , s , res) { visited [s] = true ; res . push (s); // Recursively visit all adjacent vertices // that are not visited yet for (let i of adj [s]) { if (! visited [i]) { dfsRec (adj , visited , i , res); } } } function dfs (adj) { const visited = new Array (adj . length). fill (false); const res = []; dfsRec (adj , visited , 0 , res); return res ; } //Driver Code Starts function addEdge (adj , u , v) { adj [u]. push (v); adj [v]. push (u); } // Driver code let V = 5 ; let adj = []; // creating adjacency list for (let i = 0 ; i < V ; i ++) adj . push ([]); addEdge (adj , 1 , 2); addEdge (adj , 1 , 0); addEdge (adj , 2 , 0); addEdge (adj , 2 , 3); addEdge (adj , 2 , 4); // Perform DFS starting from vertex 0 const res = dfs (adj); for (let i = 0 ; i < res . length ; i ++) { process . stdout . write (res [i] + " "); } //Driver Code Ends Output 0 1 2 3 4 Time complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph. Auxiliary Space: O(V + E), since an extra visited array of size V is required, And stack size for recursive calls to dfsRec function. DFS of a Disconnected Graph: In a disconnected graph, some vertices may not be reachable from a single source. To ensure all vertices are visited in DFS traversal, we iterate through each vertex, and if a vertex is unvisited, we perform a DFS starting from that vertex being the source. This way, DFS explores every connected component of the graph. Below if the implementation of DFS for a disconnected graph: C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends void dfsRec (vector < vector < int >> & adj , vector < bool > & visited , int s , vector < int > & res) { visited [s] = true ; res . push_back (s); // Recursively visit all adjacent // vertices that are not visited yet for (int i : adj [s]) if (visited [i] == false) dfsRec (adj , visited , i , res); } vector < int > dfs (vector < vector < int >> & adj) { vector < bool > visited (adj . size (), false); vector < int > res ; // Loop through all vertices // to handle disconnected graph for (int i = 0 ; i < adj . size (); i ++) { if (visited [i] == false) { dfsRec (adj , visited , i , res); } } return res ; } //Driver Code Starts void addEdge (vector < vector < int >>& adj , int u , int v) { adj [u]. push_back (v); adj [v]. push_back (u); } int main () { int V = 6 ; vector < vector < int >> adj (V); // creating adjacency list addEdge (adj , 1 , 2); addEdge (adj , 0 , 3); addEdge (adj , 2 , 0); addEdge (adj , 5 , 4); vector < int > res = dfs (adj); for (auto it : res) cout << it << " " ; return 0 ; } //Driver Code Ends C //Driver Code Starts #include <stdio.h> #define V 6 //Driver Code Ends void dfsRec (int adj [V][V], int visited [V], int s , int res [V], int * idx) { visited [s] = 1 ; res [(* idx) ++] = s ; // Recursively

visit all adjacent // vertices that are not visited yet for (int i = 0 ; i < V ; i ++) if (adj [s][i] && visited [i] == 0) dfsRec (adj , visited , i , res , idx); } void dfs (int adj [V][V], int res [V], int * resSize) { int visited [V] = { 0 }; int idx = 0 ; // Loop through all vertices // to handle disconnected graph for (int i = 0 ; i < V ; i ++) { if (visited [i] == 0) { dfsRec (adj , visited , i , res , & idx); } } * resSize = idx ; } //Driver Code Starts void addEdge (int adj [V][V], int u , int v) { adj [u][v] = 1 ; adj [v][u] = 1 ; // undirected } int main () { int adj [V][V] = { 0 }; addEdge (adj , 1 , 2); addEdge (adj , 2 , 0); addEdge (adj , 0 , 3); addEdge (adj , 4 , 5); // Perform DFS int res [V]; int resSize = 0 ; dfs (adj , res , & resSize); for (int i = 0 ; i < resSize ; i ++) printf ("%d " , res [i]); return 0 ; } //Driver Code Ends Java //Driver Code Starts import java.util.ArrayList ; public class GFG { //Driver Code Ends private static void dfsRec (ArrayList < ArrayList < Integer > > adj , boolean [] visited , int s , ArrayList < Integer > res) { visited [s] = true ; res . add (s); // Recursively visit all adjacent vertices that are // not visited yet for (int i : adj . get (s)) { if (! visited [i]) { dfsRec (adj , visited , i , res); } } } public static ArrayList < Integer > dfs (ArrayList < ArrayList < Integer > > adj) { boolean [] visited = new boolean [adj . size ()] ; ArrayList < Integer > res = new ArrayList <> (); // Loop through all vertices // to handle disconnected graphs for (int i = 0 ; i < adj . size (); i ++) { if (! visited [i]) { dfsRec (adj , visited , i , res); } } return res ; } //Driver Code Starts static void addEdge (ArrayList < ArrayList < Integer >> adj , int u , int v) { adj . get (u). add (v); adj . get (v). add (u); } public static void main (String [] args) { int V = 6 ; ArrayList < ArrayList < Integer >> adj = new ArrayList <> (); // creating adjacency list for (int i = 0 ; i < V ; i ++) adj . add (new ArrayList <> ()); addEdge (adj , 1 , 2); addEdge (adj , 0 , 3); addEdge (adj , 2 , 0); addEdge (adj , 5 , 4); // Perform DFS ArrayList < Integer > res = dfs (adj); for (int num : res) { System . out . print (num + " "); } } } //Driver Code Ends Python #Driver Code Starts from collections import defaultdict #Driver Code Ends def dfsRec (adj , visited , s , res): visited [s] = True res . append (s) # Recursively visit all adjacent # vertices that are not visited yet for i in adj [s]: if not visited [i]: dfsRec (adj , visited , i , res) def dfs (adj): visited = [False] * len (adj) res = [] # Loop through all vertices to # handle disconnected graph for i in range (len (adj)): if not visited [i]: dfsRec (adj , visited , i , res) return res #Driver Code Starts def addEdge (adj , u , v): adj [u] . append (v) adj [v] . append (u) if __name__ == "__main__" : V = 6 adj = [] # creating adjacency list for i in range (V): adj . append ([]) addEdge (adj , 1 , 2) addEdge (adj , 2 , 0) addEdge (adj , 0 , 3) addEdge (adj , 5 , 4) # Perform DFS res = dfs (adj) print (* res) #Driver Code Ends C# //Driver Code Starts using System ; using System.Collections.Generic ; class GFG { //Driver Code Ends static void dfsRec (List < List < int >> adj , bool [] visited , int s , List < int > res) { visited [s] = true ; res . Add (s); // Recursively visit all adjacent // vertices that are not visited yet foreach (int i in adj [s]) { if (! visited [i]) { dfsRec (adj , visited , i , res); } } } static List < int > dfs (List < List < int >> adj) { bool [] visited = new bool [adj . Count]; List < int > res = new List < int > (); // Loop through all vertices // to handle disconnected graphs for (int i = 0 ; i < adj . Count ; i ++) { if (! visited [i]) { dfsRec (adj , visited , i , res); } } return res ; } //Driver Code Starts static void addEdge (List < List < int >> adj , int u , int v) { adj [u]. Add (v); adj [v]. Add (u); } static void Main () { int V = 6 ; List < List < int >> adj = new List < List < int >> (); // creating adjacency list for (int i = 0 ; i < V ; i ++) adj . Add (new List < int > ()); addEdge (adj , 1 , 2); addEdge (adj , 2 , 0); addEdge (adj , 0 , 3); addEdge (adj , 5 , 4); // Perform DFS List < int > res = dfs (adj); foreach (int num in res) { Console . Write (num + " "); } } } //Driver Code Ends JavaScript function dfsRec (adj , visited , s , res) { visited [s] = true ; res . push (s); // Recursively visit all adjacent // vertices that are not visited yet for (let i of adj [s]) { if (! visited [i]) { dfsRec (adj , visited , i , res); } } } function dfs (adj) { const visited = new Array (adj . length). fill (false); const res = []; // Loop through all vertices // to handle disconnected graphs for (let i = 0 ; i < adj . length ; i ++) { if (! visited [i]) { dfsRec (adj , visited , i , res); } } return res ; } //Driver Code Starts function addEdge (adj , u , v) { adj [u]. push (v); adj [v]. push (u); } // Driver code let V = 6 ; let adj = []; // creating adjacency list for (let i = 0 ; i < V ; i ++) adj . push ([]); addEdge (adj , 1 , 2); addEdge (adj , 2 , 0); addEdge (adj , 0 , 3); addEdge (adj , 5 , 4); // Perform DFS const res = dfs (adj); for (const num of res) { process . stdout . write (num + " "); } //Driver Code Ends Output 0 3 2 1 4 5 Time complexity: O(V + E). We visit every vertex at most once and every edge is traversed at most once (in directed) and twice in undirected. Auxiliary Space: O(V + E), since an extra visited array of size V is required, and stack size for recursive calls of dfs function. Related Articles: Depth First Search or DFS on Directed Graph Breadth First Search or BFS for a Graph Comment Article Tags: Article Tags: Graph Algorithms DSA DFS graph-basics + 1 More