Breadth First Search or BFS for a Graph - GeeksforGeeks

Source: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-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 Breadth First Search or BFS for a Graph Last Updated : 16 Jan, 2026 Breadth First Search (BFS) is a graph traversal algorithm that starts from a source node and explores the graph level by level. First, it visits all nodes directly adjacent to the source. Then, it moves on to visit the adjacent nodes of those nodes, and this process continues until all reachable nodes are visited. BFS is different from DFS in a way that closest vertices are visited before others. We mainly traverse vertices level by level. Popular graph algorithms like Dijkstra's shortest path , Kahn's Algorithm , and Prim's algorithm are based on BFS. BFS itself can be used to detect cycle in a directed and undirected graph, find shortest path in an unweighted graph and many more problems. Examples: Input: adj[][] = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]] Output: [0, 1, 2, 3, 4] Explanation: Starting from 0, the BFS traversal proceeds as follows: Visit 0 - Print: 0 Visit 1 (neighbor of 0) - Print: 1 Visit 2 (next neighbor of 0) - Print: 2 Visit 3 (first neighbor of 2 that hasn't been visited yet) - Print: 3 Visit 4 (next neighbor of 2) - Print: 4 Input: adj[][] = [[2, 3], [2], [0, 1], [0], [5], [4]] Output: [0, 2, 3, 1, 4, 5] Explanation: Starting from 0, the BFS traversal proceeds as follows: Visit 0 - Print: 0 Visit 2 (first neighbor of 0) - Print: 2 Visit 3 (next neighbor of 0) - Print: 3 Visit 1 (neighbor of 2 that hasn't been visited yet) - Print: 1 Start another BFS traversal with source as 4: Visit 4 - Print: 4 Visit 5 (neighbor of 4) - Print: 5 Try it on GfG Practice BFS from a Given Source in an Undirected Graph: The algorithm starts from a given source vertex and explores all vertices reachable from that source, visiting nodes in increasing order of their distance from the source, level by level using a queue. Since graphs may contain cycles, a vertex could be visited multiple times. To prevent revisiting a vertex, a visited array is used. Let us understand the working of Breadth First Search with the help of the following Illustration : C++ //Driver Code Starts #include <iostream> #include <vector> #include <queue> using namespace std ; //Driver Code Ends // BFS for single connected component vector < int > bfs (vector < vector < int >>& adj) { int V = adj . size (); vector < bool > visited (V , false); vector < int > res ; queue < int > q ; int src = 0 ; visited [src] = true ; q . push (src); while (! q . empty ()) { int curr = q . front (); q . pop (); res . push_back (curr); // visit all the unvisited // neighbours of current node for (int x : adj [curr]) { if (! visited [x]) { visited [x] = true ; q . push (x); } } } 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); vector < int > res = bfs (adj); for (int i : res) cout << i << " " ; } //Driver Code Ends C //Driver Code Starts #include <stdio.h> #define V 5 #define MAXQ 100 //Driver Code Ends // BFS for single connected component void bfs (int adj [V][V], int res [V], int * resSize) { int visited [V] = { 0 }; int q [MAXQ]; int front = 0 , rear = 0 ; int src = 0 ; visited [src] = 1 ; q [rear ++] = src ; while (front < rear) { int curr = q [front ++]; res [(* resSize) ++] = curr ; // visit all the unvisited // neighbours of current node for (int x = 0 ; x < V ; x ++) { if (adj [curr][x] && ! visited [x]) { visited [x] = 1 ; q [rear ++] = x ; } } } } //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]; int resSize = 0 ; bfs (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 ; import

```
java.util.Queue ; import java.util.LinkedList ; class GFG { //Driver Code Ends // BFS for single connected component static ArrayList < Integer > bfs ( ArrayList < ArrayList < Integer >> adj ) { int V = adj . size (); boolean [] visited = new boolean [ V ] ; ArrayList < Integer > res = new ArrayList <> (); int src = 0 ; Queue < Integer > q = new LinkedList <> (); visited [ src ] = true ; q . add ( src ); while ( ! q . isEmpty ()) { int curr = q . poll (); res . add ( curr ); // visit all the unvisited // neighbours of current node for ( int x : adj . get ( curr )) { if ( ! visited [ x ] ) { visited [ x ] = true ; q . add ( x ); } } } 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 ); ArrayList < Integer > res = bfs ( adj ); for ( int x : res ) System . out . print ( x + " " ); } } //Driver Code Ends Python #Driver Code Starts from collections import deque #Driver Code Ends # BFS for single connected component def bfs ( adj ): V = len ( adj ) visited = [ False ] * V res = [] src = 0 q = deque () visited [ src ] = True q . append ( src ) while q : curr = q . popleft () res . append ( curr ) # visit all the unvisited # neighbours of current node for x in adj [ curr ]: if not visited [ x ]: visited [ x ] = True q . append ( x ) 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 ) res = bfs ( 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 // BFS for single connected component static List < int > bfs ( List < List < int >> adj ) { int V = adj . Count ; bool [] visited = new bool [ V ]; List < int > res = new List < int > (); int src = 0 ; Queue < int > q = new Queue < int > (); visited [ src ] = true ; q . Enqueue ( src ); while ( q . Count > 0 ) { int curr = q . Dequeue (); res . Add ( curr ); // visit all the unvisited // neighbours of current node foreach ( int x in adj [ curr ]) { if ( ! visited [ x ]) { visited [ x ] = true ; q . Enqueue ( x ); } } } 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 ); List < int > res = bfs ( adj ); foreach ( int i in res ) Console . Write ( i + " " ); } } //Driver Code Ends JavaScript //Driver Code Starts const Denque = require ( "denque" ); //Driver Code Ends // BFS for single connected component function bfs ( adj ) { const V = adj . length ; const visited = new Array ( V ). fill ( false ); const res = []; const q = new Denque (); let src = 0 ; visited [ src ] = true ; q . push ( src ); while ( ! q . isEmpty ()) { const curr = q . shift (); res . push ( curr ); // visit all the unvisited // neighbours of current node for ( const x of adj [ curr ]) { if ( ! visited [ x ]) { visited [ x ] = true ; q . push ( x ); } } } 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 ); const res = bfs ( 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), BFS explores all the vertices and edges in the graph. It visits every vertex and edge only once. Auxiliary Space: O(V), Using a queue to keep track of the vertices that need to be visited. BFS of a Disconnected Undirected Graph: In a disconnected graph, some vertices may not be reachable from a single source. To ensure all vertices are visited in BFS traversal, we iterate through each vertex, and if any vertex is unvisited, we perform a BFS starting from that vertex being the source. This way, BFS explores every connected component of the graph. C++ //Driver Code Starts #include <iostream> #include <vector> #include <queue> using namespace std ; //Driver Code Ends // BFS for a single connected component void bfsConnected ( vector < vector < int >>& adj , int src , vector < bool >& visited , vector < int >& res ) { queue < int > q ; visited [ src ] = true ; q . push ( src ); while ( ! q . empty ()) { int curr = q . front (); q . pop (); res . push_back ( curr ); // visit all the unvisited // neighbours of current node for ( int x : adj [ curr ]) { if ( ! visited [ x ]) { visited [ x ] = true ; q . push ( x ); } } } } // BFS for all components (handles disconnected graphs) vector < int > bfs ( vector < vector < int >>& adj ) { int V = adj . size (); vector < bool > visited ( V , false ); vector < int > res ; for ( int i = 0 ; i < V ; i ++ ) { if ( ! visited [ i ]) bfsConnected ( adj , i , visited , 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 , 2 , 0 ); addEdge ( adj , 0 , 3 ); addEdge ( adj , 4 , 5 ); vector < int > res = bfs ( adj ); for ( int i : res ) cout << i << " " ; } //Driver Code Ends C
```

```
//Driver Code Starts #include <stdio.h> #define V 6 #define MAXQ 100 //Driver Code Ends // BFS for a single connected component void bfsConnected ( int adj [ V ][ V ], int src , int visited [ V ], int res [ V ], int * resSize ) { int q [ MAXQ ]; int front = 0 , rear = 0 ; visited [ src ] = 1 ; q [ rear ++ ] = src ; while ( front < rear ) { int curr = q [ front ++ ]; res [( * resSize ) ++ ] = curr ; // visit all the unvisited // neighbours of current node for ( int x = 0 ; x < V ; x ++ ) { if ( adj [ curr ][ x ] && ! visited [ x ]) { visited [ x ] = 1 ; q [ rear ++ ] = x ; } } } } // BFS for all components (handles disconnected graphs) void bfs ( int adj [ V ][ V ], int res [ V ], int * resSize ) { int visited [ V ] = { 0 }; for ( int i = 0 ; i < V ; i ++ ) { if ( ! visited [ i ]) bfsConnected ( adj , i , visited , res , resSize ); } } //Driver Code Starts void addEdge ( int adj [ V ][ V ], int u , int v ) { adj [ u ][ v ] = 1 ; adj [ v ][ u ] = 1 ; } int main () { int adj [ V ][ V ] = { 0 }; addEdge ( adj , 1 , 2 ); addEdge ( adj , 2 , 0 ); addEdge ( adj , 0 , 3 ); addEdge ( adj , 4 , 5 ); int res [ V ]; int resSize = 0 ; bfs ( 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 ; import java.util.Queue ; import java.util.LinkedList ; class GfG { //Driver Code Ends // BFS for a single connected component static void bfsConnected ( ArrayList < ArrayList < Integer >> adj , int src , boolean [] visited , ArrayList < Integer > res ) { Queue < Integer > q = new LinkedList <> (); visited [ src ] = true ; q . add ( src ); while ( ! q . isEmpty ()) { int curr = q . poll (); res . add ( curr ); // visit all the unvisited // neighbours of current node for ( int x : adj . get ( curr )) { if ( ! visited [ x ] ) { visited [ x ] = true ; q . add ( x ); } } } } // BFS for all components (handles disconnected graphs) static ArrayList < Integer > bfs ( ArrayList < ArrayList < Integer >> adj ) { int V = adj . size (); boolean [] visited = new boolean [ V ] ; ArrayList < Integer > res = new ArrayList <> (); for ( int i = 0 ; i < V ; i ++ ) { if ( ! visited [ i ] ) bfsConnected ( adj , i , visited , 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 , 2 , 0 ); addEdge ( adj , 0 , 3 ); addEdge ( adj , 4 , 5 ); ArrayList < Integer > res = bfs ( adj ); for ( int x : res ) System . out . print ( x + " " ); } } //Driver Code Ends Python #Driver Code Starts from collections import deque #Driver Code Ends # BFS for a single connected component def bfsConnected ( adj , src , visited , res ): q = deque () visited [ src ] = True q . append ( src ) while q : curr = q . popleft () res . append ( curr ) # visit all the unvisited # neighbours of current node for x in adj [ curr ]: if not visited [ x ]: visited [ x ] = True q . append ( x ) # BFS for all components (handles disconnected graphs) def bfs ( adj ): V = len ( adj ) visited = [ False ] * V res = [] for i in range ( V ): if not visited [ i ]: bfsConnected ( adj , i , visited , 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 , 4 , 5 ) res = bfs ( 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 // BFS for a single connected component static void bfsConnected ( List < List < int >> adj , int src , bool [] visited , List < int > res ) { Queue < int > q = new Queue < int > (); visited [ src ] = true ; q . Enqueue ( src ); while ( q . Count > 0 ) { int curr = q . Dequeue (); res . Add ( curr ); // visit all the unvisited // neighbours of current node foreach ( int x in adj [ curr ]) { if ( ! visited [ x ]) { visited [ x ] = true ; q . Enqueue ( x ); } } } } // BFS for all components (handles disconnected graphs) static List < int > bfs ( List < List < int >> adj ) { int V = adj . Count ; bool [] visited = new bool [ V ]; List < int > res = new List < int > (); for ( int i = 0 ; i < V ; i ++ ) { if ( ! visited [ i ]) bfsConnected ( adj , i , visited , 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 , 4 , 5 ); List < int > res = bfs ( adj ); foreach ( int i in res ) Console . Write ( i + " " ); } } //Driver Code Ends JavaScript //Driver Code Starts // BFS for a single connected component function bfsConnected ( adj , src , visited , res ) { //Driver Code Ends const q = []; visited [ src ] = true ; q . push ( src ); while ( q . length > 0 ) { // dequeue const curr = q . shift (); res . push ( curr ); // visit all the unvisited neighbours for ( const x of adj [ curr ]) { if ( ! visited [ x ]) { visited [ x ] = true ; // enqueue q . push ( x ); } } } } // BFS for all components (handles disconnected graphs) function bfs ( adj ) { const V = adj . length ; const visited = new Array ( V ). fill ( false ); const res = []; for ( let i = 0 ; i < V ; i ++ ) { if ( ! visited [ i ]) bfsConnected ( adj , i , visited , res ); } return res ; } function addEdge ( adj , u , v ) { adj [ u ]. push ( v ); //Driver Code Starts 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 , 4 , 5 ); const res = bfs ( adj ); for ( const num of res ) { process . stdout . write ( num + " " ); } //Driver Code Ends Output 0 2
```

3 1 4 5 Time Complexity: O(V + E), The for loop ensures BFS starts from every unvisited vertex to cover all components, but the visited array ensures each vertex and edge is processed only once, keeping the total time complexity to be linear. Auxiliary Space: O(V), using a queue to keep track of the vertices that need to be visited. Applications of BFS in Graphs BFS has various applications in graph theory and computer science, including: Shortest Path Finding Cycle Detection Connected Components Network Routing To read more about applications of BFS, read this article . Comment Article Tags: Article Tags: DSA BFS graph-basics