Check whether a given graph is Bipartite or not - GeeksforGeeks

Source: https://www.geeksforgeeks.org/bipartite-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 Check whether a given graph is Bipartite or not Last Updated : 23 Jul, 2025 Given a graph with V vertices numbered from 0 to V-1 and a list of edges , determine whether the graph is bipartite or not. Note: A bipartite graph is a type of graph where the set of vertices can be divided into two disjoint sets , say U and V , such that every edge connects a vertex in U to a vertex in V, there are no edges between vertices within the same set . Example: Input: V = 4, edges[][]= [[0, 1], [0, 2], [1, 2], [2, 3]] Output: false Explanation: node 1 and node 2 have same color while coloring The graph is not bipartite because no matter how we try to color the nodes using two colors, there exists a cycle of odd length (like 1–2–0–1), which leads to a situation where two adjacent nodes end up with the same color. This violates the bipartite condition, which requires that no two connected nodes share the same color. Input: V = 4, edges[][] = [[0, 1], [1, 2], [2, 3]] Output: true Explanation: The given graph can be colored in two colors so, it is a bipartite graph. Try it on GfG Practice Table of Content Using Breadth-First Search (BFS) Using Depth-First Search (DFS) Using Breadth-First Search (BFS) Checking if a graph is bipartite is like trying to color the graph using only two colors, so that no two adjacent vertices have the same color. One approach is to check whether the graph is 2-colorable or not using backtracking algorithm m coloring problem . A common and efficient way to solve this is by using Breadth-First Search (BFS) . The idea is to traverse the graph level by level and assign colors alternately to the vertices as we proceed. Step-by-step approach: Start BFS from any uncolored vertex and assign it color 0 . For each vertex, color its uncolored neighbors with the opposite color (1 if current is 0 , and vice versa) Check if a neighbor already has the same color as the current vertex, return false (graph is not bipartite). If BFS completes without any conflicts, return true (graph is bipartite). Below is the implementation of the above approach: C++ //Driver Code Starts #include <bits/stdc++.h> using namespace std ; vector < vector < int >> constructadj (int V , vector < vector < int >> & edges){ vector < vector < int >> adj (V); for (auto it : edges){ adj [it [0]]. push_back (it [1]); adj [it [1]]. push_back (it [0]); } return adj ; } // Function to check if the graph is Bipartite using BFS //Driver Code Ends bool isBipartite (int V , vector < vector < int >> & edges) { // Vector to store colors of vertices. // Initialize all as -1 (uncolored) vector < int > color (V , -1); //create adjacency list vector < vector < int >> adj = constructadj (V , edges); // Queue for BFS queue < int > q ; // Iterate through all vertices to handle disconnected graphs for (int i = 0 ; i < V ; i ++) { // If the vertex is uncolored, start BFS from it if (color [i] == -1) { // Assign first color (0) to the starting vertex color [i] = 0 ; q . push (i); // Perform BFS while (! q . empty ()) { int u = q . front (); q . pop (); // Traverse all adjacent vertices for (auto & v : adj [u]) { // If the adjacent vertex is uncolored, // assign alternate color if (color [v] == -1) { color [v] = 1 - color [u]; q . push (v); } // If the adjacent vertex has the same color, // graph is not bipartite else if (color [v] == color [u]) { return false ; } } } } } // If no conflicts in coloring, graph is bipartite //Driver Code Starts return true ; } int main () { int V = 4 ; vector < vector < int >> edges = {{ 0 , 1 }, { 0 , 2 }, { 1 , 2 }, { 2 , 3 }}; if (isBipartite (V , edges)) cout << "true" ; else cout << "false" ; return 0 ; } //Driver Code Ends Java //Driver Code Starts import java.util.* ; class GfG { // Function to construct the adjacency list from edges static ArrayList < ArrayList < Integer >> constructadj (int V , int [][] edges) { ArrayList < ArrayList < Integer >> adj = new ArrayList <> (); for (int i = 0 ; i < V ; i ++) { adj . add (new ArrayList <> ()); } for (int [] edge : edges) { int u = edge [0] ; int v = edge [1] ; adj . get (u). add (v); adj . get (v). add (u); } return adj ; } // Function to

check if the graph is Bipartite using BFS //Driver Code Ends static boolean isBipartite (int V , int [][] edges) { int [] color = new int [V] ; Arrays . fill (color , - 1); // create adjacency list ArrayList < ArrayList < Integer >> adj = constructadj (V , edges); for (int i = 0 ; i < V ; i ++) { if (color [i] == - 1) { Queue < Integer > q = new LinkedList <> (); color [i] = 0 ; q . offer (i); while (! q . isEmpty ()) { int u = q . poll (); for (int v : adj . get (u)) { if (color [v] == - 1) { color [v] = 1 - color [u] ; q . offer (v); } else if (color [v] == color [u]) { return false ; // Conflict found } } } } //Driver Code Starts } return true ; } public static void main (String [] args) { int V = 4 ; // Edges of the graph int [][] edges = {{ 0 , 1 }, { 0 , 2 }, { 1 , 2 }, { 2 , 3 }}; // Check if the graph is bipartite System . out . println (isBipartite (V , edges)); } } //Driver Code Ends Python #Driver Code Starts from collections import deque # Function to construct the adjacency list from edges def constructadj (V , edges): adj = [[] for _ in range (V)] for edge in edges : u , v = edge adj [u] . append (v) adj [v] . append (u) return adj # Function to check if the graph is Bipartite using BFS #Driver Code Ends def isBipartite (V , adj): # Initialize all as uncolored color = [- 1] * V # create adjacency list adj = constructadj (V , edges) for i in range (V): if color [i] == - 1 : color [i] = 0 q = deque ([i]) while q : u = q . popleft () for v in adj [u]: if color [v] == - 1 : color [v] = 1 - color [u] q . append (v) elif color [v] == color [u]: return False # Conflict found #Driver Code Starts # No conflict, graph is bipartite return True if __name__ == "__main__" : V = 4 edges = [[0 , 1], [0 , 2], [1 , 2], [2 , 3]] print ("true" if isBipartite (V , edges) else "false") #Driver Code Ends C# //Driver Code Starts using System ; using System.Collections.Generic ; class GfG { public static List < List < int >> constructadj (int V , List < List < int >> edges) { List < List < int >> adj = new List < List < int >> (); for (int i = 0 ; i < V ; i ++) adj . Add (new List < int > ()); foreach (var edge in edges) { int u = edge [0]; int v = edge [1]; adj [u]. Add (v); adj [v]. Add (u); } return adj ; } //Driver Code Ends // Function to check if the graph is Bipartite using BFS public static bool IsBipartite (int V , List < List < int > > edges){ int [] color = new int [V]; // create adjacency list List < List < int >> adj = constructadj (V , edges); // Initialize all as -1 (uncolored) Array . Fill (color , - 1); // Iterate through all vertices to handle // disconnected graphs for (int i = 0 ; i < V ; i ++) { // If the vertex is uncolored, start BFS from it if (color [i] == - 1) { // Assign first color (0) color [i] = 0 ; Queue < int > q = new Queue < int > (); q . Enqueue (i); // Perform BFS while (q . Count > 0) { int u = q . Dequeue (); // Traverse all adjacent vertices foreach (int v in adj [u]){ // If the adjacent vertex is // uncolored, assign alternate color if (color [v] == - 1) { color [v] = 1 - color [u]; q . Enqueue (v); } // If the adjacent vertex has the // same color, graph is not // bipartite else if (color [v] == color [u]) { return false ; } } } } } // If no conflicts in coloring, graph is bipartite //Driver Code Starts return true ; } static void Main (){ int V = 4 ; List < List < int >> edges = new List < List < int >> { new List < int > { 0 , 1 }, new List < int > { 0 , 2 }, new List < int > { 1 , 2 }, new List < int > { 2 , 3 } }; if (IsBipartite (V , edges)) Console . WriteLine ("true"); else Console . WriteLine ("false"); } } //Driver Code Ends JavaScript //Driver Code Starts // Function to construct adjacency list from edges function constructadj (V , edges) { const adj = Array . from ({ length : V }, () => []); for (const [u , v] of edges) { adj [u]. push (v); adj [v]. push (u); // undirected graph } return adj ; } //Driver Code Ends function isBipartite (V , edges){ // Initialize all as -1 (uncolored) const color = Array (V). fill (- 1); // create adjacency list let adj = constructadj (V , edges); // Iterate through all vertices to handle disconnected // graphs for (let i = 0 ; i < V ; i ++) { // If the vertex is uncolored, start BFS from it if (color [i] === - 1) { // Assign first color (0) color [i] = 0 ; const queue = [i]; // Perform BFS while (queue . length > 0) { const u = queue . shift (); // Traverse all adjacent vertices for (let v of adj [u]) { // If the adjacent vertex is uncolored, // assign alternate color if (color [v] === - 1) { color [v] = 1 - color [u]; queue . push (v); // Push to queue } // If the adjacent vertex has the same // color, graph is not bipartite else if (color [v] === color [u]) { return false ; } } } } } // If no conflicts in coloring, graph is bipartite //Driver Code Starts return true ; } // Driver Code const V = 4 ; const adj = Array . from ({ length : V }, () => []); let edges = [[0 , 1], [0 , 2], [1 , 2], [2 , 3]]; console . log (isBipartite (V , edges)); //Driver Code Ends Output false Time Complexity: O(V + E), where V is the number of vertices and E is the number of edges. This is because BFS explores each vertex and edge exactly once. Auxiliary Space: O(V), The queue used in BFS, which can hold up to V vertices and The color array (or map), which stores the color for each vertex, We do not count the adjacency list in auxiliary space as it is necessary for representing the input graph. Using Depth-First Search (DFS) We can also check if a graph is bipartite using Depth-First Search (DFS) . We need to color the graph with two colors such that no two adjacent vertices share the same color. We start from any uncolored vertex, assigning it a color (e.g., color 0). As we explore each vertex, we recursively color its uncolored neighbors with the another color. If we ever find a neighbor that shares the same color as the current vertex, we can simply conclude that the graph is not bipartite. If there is no conflict found after the traversal then the given graph is bipartite. For implementation of DFS approach please refer to this article " Check if a given

graph is Bipartite using DFS ". Related Article: What is Bipartite Graph M-Coloring Problem Comment Article Tags: Article Tags: Graph DSA Samsung BFS DFS Graph Coloring + 2 More