

Articulation Points (or Cut Vertices) in a Graph - GeeksforGeeks

Source: <https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-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 Articulation Points (or Cut Vertices) in a Graph Last Updated : 23 Jul, 2025 Given an undirected graph with V vertices and E edges (edges[][]), Your task is to return all the articulation points in the graph. If no such point exists, return {-1}. Note: An articulation point is a vertex whose removal, along with all its connected edges, increases the number of connected components in the graph. The graph may contain more than one connected component. Examples: Input: V = 5, edges[][] = [[0, 1], [1, 4], [4, 3], [4, 2], [2, 3]] Output: [1, 4] Explanation: Removing the vertex 1 or 4 will disconnects the graph as: Input: V = 4, edges[][] = [[0, 1], [0, 2]] Output: [0] Explanation: Removing the vertex 0 will increase the number of disconnected components to 3. Try it on GfG Practice Table of Content [Naive Approach] Using DFS - $O(V * (V + E))$ Time and $O(V)$ Space [Expected Approach] - Using Tarjan's Algorithm - $O(V + E)$ Time and $O(V)$ Space [Naive Approach] Using DFS - $O(V * (V + E))$ Time and $O(V)$ Space Condition for particular node to be an articulation point: A node is an articulation point if, after removing it, you need more than one DFS traversal to visit all of its neighbors. This means that at least two of its neighbors (children) end up in different disconnected components, and cannot reach each other without this node. Step by Step implementations: Iterate over all nodes i (possible articulation candidates). For each node i, pretend it's removed from the graph by marking it as already visited, so DFS will skip it. For each unvisited neighbor of i, start DFS Count how many separate DFS calls are needed (stored in comp) to visit all neighbor. If comp > 1, it means i connects multiple components, thus i is an articulation point. After checking all nodes, return the list of articulation points, or {-1} if none found. C++ // C++ program to find articulation points using a naive DFS approach #include <bits/stdc++.h> using namespace std ; // Standard DFS to mark all reachable nodes void dfs (int node , vector < vector < int >> & adj , vector < bool > & visited) { visited [node] = true ; for (int neighbor : adj [node]) { if (! visited [neighbor]) { dfs (neighbor , adj , visited) ; } } } // Builds adjacency list from edge list 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 ; } // Finds articulation points using naive DFS approach vector < int > articulationPoints (int V , vector < vector < int >> & edges) { vector < vector < int >> adj = constructadj (V , edges) ; vector < int > res ; // Try removing each node one by one for (int i = 0 ; i < V ; ++ i) { vector < bool > visited (V , false) ; visited [i] = true ; // count DFS calls from i's neighbors int comp = 0 ; for (auto it : adj [i]) { // early stop if already more than 1 component if (comp > 1) break ; if (! visited [it]) { // explore connected part dfs (it , adj , visited) ; comp ++ ; } } // if more than one component forms, it's an articulation point if (comp > 1) res . push_back (i) ; } if (res . empty ()) return { -1 } ; return res ; } int main () { int V = 5 ; vector < vector < int >> edges = { { 0 , 1 } , { 1 , 4 } , { 2 , 3 } , { 2 , 4 } , { 3 , 4 } } ; vector < int > ans = articulationPoints (V , edges) ; for (auto it : ans) { cout << it << " " ; } return 0 ; } Java // Java program to find articulation points // using a naive DFS approach import java.util.* ; class GfG { // Standard DFS to mark all reachable nodes static void dfs (int node , ArrayList < ArrayList < Integer >> adj , boolean [] visited) { visited [node] = true ; for (int neighbor : adj . get (node)) { if (! visited [neighbor]) { dfs (neighbor , adj , visited) ; } } } // Builds adjacency list from edge list 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) { adj . get (edge [0]) .

```

add ( edge [ 1 ] ); adj . get ( edge [ 1 ] ) . add ( edge [ 0 ] ); } return adj ; } // Finds articulation points using
naive DFS approach static ArrayList < Integer > articulationPoints ( int V , int [][] edges ) { ArrayList <
ArrayList < Integer >> adj = constructadj ( V , edges ); ArrayList < Integer > res = new ArrayList <> (); //
Try removing each node one by one for ( int i = 0 ; i < V ; ++ i ) { boolean [] visited = new boolean [ V ] ;
visited [ i ] = true ; // count DFS calls from i's neighbors int comp = 0 ; for ( int it : adj . get ( i ) ) { // early
stop if already more than 1 component if ( comp > 1 ) break ; if ( ! visited [ it ] ) { // explore connected
part dfs ( it , adj , visited ); comp ++ ; } } // if more than one component forms, it's an articulation point if (
comp > 1 ) res . add ( i ); } if ( res . isEmpty () ) return new ArrayList <> ( Arrays . asList ( - 1 )); return res
; } public static void main ( String [] args ) { int V = 5 ; int [][] edges = { { 0 , 1 } , { 1 , 4 } , { 2 , 3 } , { 2 , 4 } , {
3 , 4 } }; ArrayList < Integer > ans = articulationPoints ( V , edges ); for ( int it : ans ) { System . out . print
( it + " " ); } } } Python # Python program to find articulation points using a naive DFS approach def dfs (
node , adj , visited ): # Standard DFS to mark all reachable nodes visited [ node ] = True for neighbor in
adj [ node ]: if not visited [ neighbor ]: dfs ( neighbor , adj , visited ) def constructadj ( V , edges ): #
Builds adjacency list from edge list adj = [] for _ in range ( V ) for u , v in edges : adj [ u ] . append ( v )
adj [ v ] . append ( u ) return adj def articulationPoints ( V , edges ): # Finds articulation points using
naive DFS approach adj = constructadj ( V , edges ) res = [] # Try removing each node one by one for i
in range ( V ): visited = [ False ] * V visited [ i ] = True # count DFS calls from i's neighbors comp = 0 for
it in adj [ i ]: if comp > 1 : break if not visited [ it ]: # explore connected part dfs ( it , adj , visited ) comp
+= 1 # if more than one component forms, it's an articulation point if comp > 1 : res . append ( i ) if not
res : return [ - 1 ] return res if __name__ == "__main__" : V = 5 edges = [ [ 0 , 1 ] , [ 1 , 4 ] , [ 2 , 3 ] , [ 2 , 4
] , [ 3 , 4 ] ] ans = articulationPoints ( V , edges ) for it in ans : print ( it , end = " " ) C# // C# program to
find articulation points // using a naive DFS approach using System ; using System.Collections.Generic
; class GfG { // Standard DFS to mark all reachable nodes static void DFS ( int node , List < List < int >>
adj , bool [] visited ) { visited [ node ] = true ; foreach ( int neighbor in adj [ node ] ) { if ( ! visited [
neighbor ] ) { DFS ( neighbor , adj , visited ); } } } // Builds adjacency list from edge list static List < List <
int >> constructAdj ( int V , int [,] edges ) { List < List < int >> adj = new List < List < int >> (); for ( int i =
0 ; i < V ; i ++ ) { adj . Add ( new List < int > () ); } int E = edges . GetLength ( 0 ); for ( int i = 0 ; i < E ; i ++
) { int u = edges [ i , 0 ] ; int v = edges [ i , 1 ] ; adj [ u ] . Add ( v ); adj [ v ] . Add ( u ); } return adj ; } // Finds
articulation points using naive DFS approach static List < int > articulationPoints ( int V , int [,] edges ) {
List < List < int >> adj = constructAdj ( V , edges ); List < int > res = new List < int > (); // Try removing
each node one by one for ( int i = 0 ; i < V ; ++ i ) { bool [] visited = new bool [ V ]; visited [ i ] = true ; //
count DFS calls from i's neighbors int comp = 0 ; foreach ( int it in adj [ i ] ) { // early stop if already more
than 1 component if ( comp > 1 ) break ; if ( ! visited [ it ] ) { // explore connected part DFS ( it , adj ,
visited ); comp ++ ; } } // if more than one component forms, it's an articulation point if ( comp > 1 ) res .
Add ( i ); } if ( res . Count == 0 ) return new List < int > { - 1 }; return res ; } public static void Main ( string
[] args ) { int V = 5 ; int [,] edges = new int [,] { { 0 , 1 } , { 1 , 4 } , { 2 , 3 } , { 2 , 4 } , { 3 , 4 } }; List < int > ans
= articulationPoints ( V , edges ); foreach ( int it in ans ) { Console . Write ( it + " " ); } } } JavaScript //
JavaScript program to find articulation points // using a naive DFS approach // Standard DFS to mark all
reachable nodes function dfs ( node , adj , visited ) { visited [ node ] = true ; for ( let neighbor of adj [
node ] ) { if ( ! visited [ neighbor ] ) { dfs ( neighbor , adj , visited ); } } } // Builds adjacency list from edge
list function constructadj ( V , edges ) { let adj = Array . from ( { length : V } , () => []); for ( let [ u , v ] of
edges ) { adj [ u ] . push ( v ); adj [ v ] . push ( u ); } return adj ; } // Finds articulation points using naive
DFS approach function articulationPoints ( V , edges ) { const adj = constructadj ( V , edges ); const res
= []; // Try removing each node one by one for ( let i = 0 ; i < V ; ++ i ) { let visited = Array ( V ) . fill ( false
); visited [ i ] = true ; // count DFS calls from i's neighbors let comp = 0 ; for ( let it of adj [ i ] ) { // early
stop if already more than 1 component if ( comp > 1 ) break ; if ( ! visited [ it ] ) { // explore connected
part dfs ( it , adj , visited ); comp ++ ; } } // if more than one component forms, it's an articulation point if (
comp > 1 ) res . push ( i ); } if ( res . length === 0 ) return [ - 1 ]; return res ; } // Driver Code const V = 5 ;
const edges = [ [ 0 , 1 ] , [ 1 , 4 ] , [ 2 , 3 ] , [ 2 , 4 ] , [ 3 , 4 ] ]; const ans = articulationPoints ( V , edges );
console . log ( ans . join ( " " )); Output 1 4 Time Complexity: O(V * ( V + E )) where V is number of
vertices, and E is number of edges. We are performing DFS operation which have O(V + E) time
complexity, for each of the vertex. Thus the overall time taken will be O(V * ( V + E )). Auxiliary Space:
O(V) , for storing the visited[] array. [Expected Approach] Using Tarjan's Algorithm - O(V + E) Time
and O(V) Space The idea is to use DFS ( Depth First Search ). In DFS, follow vertices in a tree form
called the DFS tree . In the DFS tree, a vertex u is the parent of another vertex v , if v is discovered by u
. In DFS tree, a vertex u is an articulation point if one of the following two conditions is true. u is the root
of the DFS tree and it has at least two children. u is not the root of the DFS tree and it has a child v

```

such that no vertex in the subtree rooted with v has a back edge to one of the ancestors in DFS tree of u . Let's understand with an example: For the vertex 3 (which is not the root), vertex 4 is the child of vertex 3. No vertex in the subtree rooted at vertex 4 has a back edge to one of ancestors of vertex 3. Thus on removal of vertex 3 and its associated edges the graph will get disconnected or the number of components in the graph will increase as the subtree rooted at vertex 4 will form a separate component. Hence vertex 3 is an articulation point. Now consider the following graph: Again the vertex 4 is the child of vertex 3. For the subtree rooted at vertex 4, vertex 7 in this subtree has a back edge to one of the ancestors of vertex 3 (which is vertex 1). Thus this subtree will not get disconnected on the removal of vertex 3 because of this back edge. Since there is no child v of vertex 3, such that subtree rooted at vertex v does not have a back edge to one of the ancestors of vertex 3. Hence vertex 3 is not an articulation point in this case.

Step by Step implementation: Maintain these arrays and integers and perform a DFS traversal. `disc[]`: Discovery time of each vertex during DFS. `low[]`: The lowest discovery time reachable from the subtree rooted at that vertex (via tree or back edges). `parent`: To keep track of each node's parent in the DFS tree. `visited[]`: To mark visited nodes.

Root Node Case: For the root node of DFS (i.e., `parent[u] == -1`), check how many child DFS calls it makes. If the root has two or more children, it is an articulation point. For any non-root node u , check all its adjacent nodes: If v is an unvisited child: Recur for v , and after returning update `low[u] = min(low[u], low[v])`. If `low[v] >= disc[u]`, then u is an articulation point because v and its subtree cannot reach any ancestor of u , so removing u would disconnect v .

Back Edge Case If v is already visited and is not the parent of u then It's a back edge. Update `low[u] = min(low[u], disc[v])`. This helps bubble up the lowest reachable ancestor through a back edge. After DFS traversal completes, all nodes marked as articulation points are stored in result array.

C++

```
#include <bits/stdc++.h> using namespace std ; vector < vector < int >> constructAdj ( int V , vector < vector < int >> & edges ) { vector < vector < int >> adj ( V ); for ( auto & edge : edges ) { adj [ edge [ 0 ] ]. push_back ( edge [ 1 ] ); adj [ edge [ 1 ] ]. push_back ( edge [ 0 ] ); } return adj ; } // Helper function to perform DFS and find articulation points // using Tarjan's algorithm. void findPoints ( vector < vector < int >> & adj , int u , vector < int > & visited , vector < int > & disc , vector < int > & low , int & time , int parent , vector < int > & isAP ) { // Mark vertex u as visited and assign discovery // time and low value visited [ u ] = 1 ; disc [ u ] = low [ u ] = ++ time ; int children = 0 ; // Process all adjacent vertices of u for ( int v : adj [ u ] ) { // If v is not visited, then recursively visit it if ( ! visited [ v ] ) { children ++ ; findPoints ( adj , v , visited , disc , low , time , u , isAP ); // Check if the subtree rooted at v has a // connection to one of the ancestors of u low [ u ] = min ( low [ u ] , low [ v ] ); // If u is not a root and low[v] is greater than or equal to disc[u], // then u is an articulation point if ( parent != -1 && low [ v ] >= disc [ u ] ) { isAP [ u ] = 1 ; } } // Update low value of u for back edge else if ( v != parent ) { low [ u ] = min ( low [ u ] , disc [ v ] ); } } // If u is root of DFS tree and has more than // one child, it is an articulation point if ( parent == -1 && children > 1 ) { isAP [ u ] = 1 ; } } // Main function to find articulation points in the graph vector < int > articulationPoints ( int V , vector < vector < int >> & edges ) { vector < vector < int >> adj = constructAdj ( V , edges ); vector < int > disc ( V , 0 ) , low ( V , 0 ) , visited ( V , 0 ) , isAP ( V , 0 ); int time = 0 ; // Run DFS from each vertex if not // already visited (to handle disconnected graphs) for ( int u = 0 ; u < V ; u ++ ) { if ( ! visited [ u ] ) { findPoints ( adj , u , visited , disc , low , time , -1 , isAP ); } } // Collect all vertices that are articulation points vector < int > result ; for ( int u = 0 ; u < V ; u ++ ) { if ( isAP [ u ] ) { result . push_back ( u ); } } // If no articulation points are found, return vector containing -1 return result . empty () ? vector < int > { -1 } : result ; } int main () { int V = 5 ; vector < vector < int >> edges = { { 0 , 1 } , { 1 , 4 } , { 2 , 3 } , { 2 , 4 } , { 3 , 4 } }; vector < int > ans = articulationPoints ( V , edges ); for ( int u : ans ) { cout << u << " " ; } cout << endl ; return 0 ; }
```

Java

```
import java.util.* ; class GfG { 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 ) { adj . get ( edge [ 0 ] ). add ( edge [ 1 ] ); adj . get ( edge [ 1 ] ). add ( edge [ 0 ] ); } return adj ; } // Helper function to perform DFS and find articulation points // using Tarjan's algorithm. static void findPoints ( ArrayList < ArrayList < Integer >> adj , int u , int [] visited , int [] disc , int [] low , int [] time , int parent , int [] isAP ) { // Mark vertex u as visited and assign discovery // time and low value visited [ u ] = 1 ; disc [ u ] = low [ u ] = ++ time [ 0 ] ; int children = 0 ; // Process all adjacent vertices of u for ( int v : adj . get ( u ) ) { // If v is not visited, then recursively visit it if ( visited [ v ] == 0 ) { children ++ ; findPoints ( adj , v , visited , disc , low , time , u , isAP ); // Check if the subtree rooted at v has a // connection to one of the ancestors of u low [ u ] = Math . min ( low [ u ] , low [ v ] ); // If u is not a root and low[v] is greater // than or equal to disc[u], // then u is an articulation point if ( parent != - 1 && low [ v ] >= disc [ u ] ) { isAP [ u ] = 1 ; } } // Update low value of u for back edge else if ( v != parent ) { low [ u ] = Math . min ( low [ u ] , disc [ v ] ); } } // If u is root of DFS tree and has more than // one child, it is an articulation point if ( parent
```

```

== - 1 && children > 1 ) { isAP [ u ] = 1 ; } } // Main function to find articulation points in the graph
static ArrayList < Integer > articulationPoints ( int V , int [][] edges ) { ArrayList < ArrayList < Integer >> adj =
constructAdj ( V , edges ); int [] disc = new int [ V ] , low = new int [ V ] , visited = new int [ V ] , isAP =
new int [ V ] ; int [] time = { 0 } ; // Run DFS from each vertex if not // already visited (to handle
disconnected graphs) for ( int u = 0 ; u < V ; u ++ ) { if ( visited [ u ] == 0 ) { findPoints ( adj , u , visited ,
disc , low , time , - 1 , isAP ); } } // Collect all vertices that are articulation points ArrayList < Integer >
result = new ArrayList <> (); for ( int u = 0 ; u < V ; u ++ ) { if ( isAP [ u ] == 1 ) { result . add ( u ); } } // If
no articulation points are found, return list containing -1 if ( result . isEmpty () ) result . add ( - 1 ); return
result ; } public static void main ( String [] args ) { int V = 5 ; int [][] edges = { { 0 , 1 } , { 1 , 4 } , { 2 , 3 } , { 2 ,
4 } , { 3 , 4 } }; ArrayList < Integer > ans = articulationPoints ( V , edges ); for ( int u : ans ) { System . out .
print ( u + " " ); } System . out . println (); } } Python def constructAdj ( V , edges ): adj = [] for _ in range
( V ) for edge in edges : adj [ edge [ 0 ] ] . append ( edge [ 1 ] ) adj [ edge [ 1 ] ] . append ( edge [ 0 ] )
return adj # Helper function to perform DFS and find articulation points # using Tarjan's algorithm. def
findPoints ( adj , u , visited , disc , low , time , parent , isAP ): # Mark vertex u as visited and assign
discovery # time and low value visited [ u ] = 1 time [ 0 ] += 1 disc [ u ] = low [ u ] = time [ 0 ] children = 0
# Process all adjacent vertices of u for v in adj [ u ]: # If v is not visited, then recursively visit it if not
visited [ v ]: children += 1 findPoints ( adj , v , visited , disc , low , time , u , isAP ) # Check if the subtree
rooted at v has a # connection to one of the ancestors of u low [ u ] = min ( low [ u ] , low [ v ] ) # If u is not
a root and low[v] is greater than or equal to disc[u], # then u is an articulation point if parent != - 1 and
low [ v ] >= disc [ u ]: isAP [ u ] = 1 # Update low value of u for back edge elif v != parent : low [ u ] = min
( low [ u ] , disc [ v ] ) # If u is root of DFS tree and has more than # one child, it is an articulation point if
parent == - 1 and children > 1 : isAP [ u ] = 1 # Main function to find articulation points in the graph def
articulationPoints ( V , edges ): adj = constructAdj ( V , edges ) disc = [ 0 ] * V low = [ 0 ] * V visited = [ 0
] * V isAP = [ 0 ] * V time = [ 0 ] # Run DFS from each vertex if not # already visited (to handle
disconnected graphs) for u in range ( V ): if not visited [ u ]: findPoints ( adj , u , visited , disc , low , time
, - 1 , isAP ) # Collect all vertices that are articulation points result = [ u for u in range ( V ) if isAP [ u ] ] #
If no articulation points are found, return list containing -1 return result if result else [ - 1 ] if __name__
== "__main__" : V = 5 edges = [ [ 0 , 1 ] , [ 1 , 4 ] , [ 2 , 3 ] , [ 2 , 4 ] , [ 3 , 4 ] ] ans = articulationPoints ( V ,
edges ) for u in ans : print ( u , end = ' ' ) print () C# using System ; using System.Collections.Generic ;
class GfG { static List < List < int >> constructAdj ( int V , int [,] edges ) { List < List < int >> adj = new
List < List < int >> (); for ( int i = 0 ; i < V ; i ++ ) { adj . Add ( new List < int > ()); } int M = edges .
GetLength ( 0 ); for ( int i = 0 ; i < M ; i ++ ) { int u = edges [ i , 0 ] ; int v = edges [ i , 1 ] ; adj [ u ] . Add ( v );
adj [ v ] . Add ( u ); } return adj ; } // Helper function to perform DFS and find articulation points // using
Tarjan's algorithm. static void findPoints ( List < List < int >> adj , int u , List < int > visited , List < int >
disc , List < int > low , ref int time , int parent , List < int > isAP ) { // Mark vertex u as visited and assign
discovery // time and low value visited [ u ] = 1 ; disc [ u ] = low [ u ] = ++ time ; int children = 0 ; //
Process all adjacent vertices of u foreach ( int v in adj [ u ] ) { // If v is not visited, then recursively visit it if
( visited [ v ] == 0 ) { children ++ ; findPoints ( adj , v , visited , disc , low , ref time , u , isAP ); } // Check if
the subtree rooted at v has a // connection to one of the ancestors of u low [ u ] = Math . Min ( low [ u ] ,
low [ v ] ); // If u is not a root and low[v] is greater // than or equal to disc[u], // then u is an articulation
point if ( parent != - 1 && low [ v ] >= disc [ u ] ) { isAP [ u ] = 1 ; } // Update low value of u for back edge
else if ( v != parent ) { low [ u ] = Math . Min ( low [ u ] , disc [ v ] ); } } // If u is root of DFS tree and has
more than // one child, it is an articulation point if ( parent == - 1 && children > 1 ) { isAP [ u ] = 1 ; } } //
Main function to find articulation points in the graph static List < int > articulationPoints ( int V , int [,]
edges ) { List < List < int >> adj = constructAdj ( V , edges ); List < int > disc = new List < int > ( new int [
V ] ); List < int > low = new List < int > ( new int [ V ] ); List < int > visited = new List < int > ( new int [ V ] );
List < int > isAP = new List < int > ( new int [ V ] ); int time = 0 ; // Run DFS from each vertex if not //
already visited (to handle disconnected graphs) for ( int u = 0 ; u < V ; u ++ ) { if ( visited [ u ] == 0 ) {
findPoints ( adj , u , visited , disc , low , ref time , - 1 , isAP ); } } // Collect all vertices that are articulation
points List < int > result = new List < int > (); for ( int u = 0 ; u < V ; u ++ ) { if ( isAP [ u ] == 1 ) { result .
Add ( u ); } } // If no articulation points are found, return list containing -1 return result . Count == 0 ?
new List < int > { - 1 } : result ; } static void Main () { int V = 5 ; int [,] edges = { { 0 , 1 } , { 1 , 4 } , { 2 , 3 } , {
2 , 4 } , { 3 , 4 } }; List < int > ans = articulationPoints ( V , edges ); foreach ( int u in ans ) { Console .
Write ( u + " " ); } Console . WriteLine (); } } JavaScript // Build adjacency list from edge list function
constructAdj ( V , edges ) { const adj = Array . from ( { length : V }, () => []); for ( let i = 0 ; i < edges .
length ; i ++ ) { const [ u , v ] = edges [ i ]; adj [ u ] . push ( v ); adj [ v ] . push ( u ); } return adj ; } // Helper
function to perform DFS and find articulation points // using Tarjan's algorithm. function findPoints ( adj ,

```

```

u , visited , disc , low , timeRef , parent , isAP ) { // Mark vertex u as visited and assign discovery // time
and low value visited [ u ] = 1 ; disc [ u ] = low [ u ] = ++ timeRef . value ; let children = 0 ; // Process all
adjacent vertices of u for ( let v of adj [ u ] ) { // If v is not visited, then recursively visit it if ( ! visited [ v ] ) {
children ++ ; findPoints ( adj , v , visited , disc , low , timeRef , u , isAP ); // Check if the subtree rooted
at v has a // connection to one of the ancestors of u low [ u ] = Math . min ( low [ u ], low [ v ] ); // If u is
not a root and low[v] is greater // than or equal to disc[u], // then u is an articulation point if ( parent !== -
1 && low [ v ] >= disc [ u ] ) { isAP [ u ] = 1 ; } } // Update low value of u for back edge else if ( v !==
parent ) { low [ u ] = Math . min ( low [ u ], disc [ v ] ); } } // If u is root of DFS tree and has more than //
one child, it is an articulation point if ( parent === - 1 && children > 1 ) { isAP [ u ] = 1 ; } } // Main
function to find articulation points in the graph function articulationPoints ( V , edges ) { const adj =
constructAdj ( V , edges ); const disc = Array ( V ). fill ( 0 ); const low = Array ( V ). fill ( 0 ); const visited
= Array ( V ). fill ( 0 ); const isAP = Array ( V ). fill ( 0 ); const timeRef = { value : 0 }; // Run DFS from
each vertex if not // already visited (to handle disconnected graphs) for ( let u = 0 ; u < V ; u ++ ) { if ( !
visited [ u ] ) { findPoints ( adj , u , visited , disc , low , timeRef , - 1 , isAP ); } } // Collect all vertices that
are articulation points const result = []; for ( let u = 0 ; u < V ; u ++ ) { if ( isAP [ u ] ) { result . push ( u ); } }
// If no articulation points are found, return array containing -1 return result . length === 0 ? [ - 1 ] : result
; } // Driver Code const V = 5 ; const edges = [ [ 0 , 1 ], [ 1 , 4 ], [ 2 , 3 ], [ 2 , 4 ], [ 3 , 4 ] ]; const ans =
articulationPoints ( V , edges ); console . log ( ans . join ( ' ' ) ); Output 1 4 Time Complexity: O(V + E),
we are performing a single DFS operation that works on O(V + E) time complexity. Auxiliary Space:
O(V), used by five arrays, each of size V Comment Article Tags: Article Tags: Graph DSA BFS DFS
graph-connectivity + 1 More

```