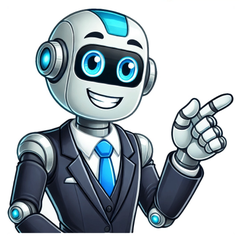

An edge in an undirected graph is said to be a bridge, if and only if by removing it, disconnects the graph, or make different components of the graph.In a practical approach, if some bridges are present in a network when the connection of bridges is broken, it can break the whole network.Input and OutputInput:The adjacency matrix of the graph.0 1 1 1 01 0 1 0 01 1 0 0 01 0 0 0 10 0 0 1 0Output:Bridges in given graph:Bridge 3--4Bridge 0--3 bridgeFind(start, visited, disc, low, parent)Input The start vertex, the visited array to mark when a node is visited, the disc will hold the discovery time of the vertex, and low will hold information about subtrees. The parent will hold the parent of the current vertex.Output print if any bridge is found.Begin time := 0 //the value of time will not be initialized for next function calls mark start as visited set disc[start] := time+1 and low[start] := time + 1 time := time + 1 for all vertex v in the graph G, do if there is an edge between (start, v), then if v is visited, then parent[v] := start bridgeFind(v, visited, disc, low, parent) low[start] := minimum of low[start] and low[v] if low[v] > disc[start], then display bridges from start to v else if v is not the parent of start, then low[start] := minimum of low[start] and disc[v] doneEndExample#include#define NODE 5using namespace std;int graph[NODE][NODE] = { {0, 1, 1, 1, 0}, {1, 0, 1, 0, 0}, {1, 1, 0, 0, 0}, {1, 0, 0, 0, 1}, {0, 0, 0, 1, 0}}; int min(int a, int b) { return (a disc[u]: bridges.append((u, v)) elif v != parent[u]: low[u] = min(low[u], disc[v]) def find_bridges(self): visited = [False] * self.V disc = [float("Inf")] * self.V low = [float("Inf")] * self.V parent = [-1] * self.V bridges = [] for i in range(self.V): if not visited[i]: self.bridge_util(i, visited, parent, low, disc, bridges) return bridges# Example usageg = Graph(5)g.add_edge(1, 0)g.add_edge(0, 2)g.add_edge(2, 1)g.add_edge(0, 3)g.add_edge(3, 4)print("Bridges in the graph:")print(g.find_bridges())This implementation creates a Graph class with methods to add edges and find bridges. The find_bridges() method initializes the necessary data structures and calls bridge_util() for each unvisited vertex. The bridge_util() method performs the DFS traversal and identifies the bridges.2. Union-Find AlgorithmAnother approach to finding critical connections is using the Union-Find data structure. This method involves removing each edge from the graph and checking if the graph remains connected.Heres a Python implementation using the Union-Find algorithm:class UnionFind: def __init__(self, n): self.parent = list(range(n)) self.rank = [0] * n def find(self, x): if self.parent[x] != x: self.parent[x] = self.find(self.parent[x]) return self.parent[x] def union(self, x, y): px, py = self.find(x), self.find(y) if px == py: return False if self.rank[px] < self.rank[py]: self.parent[px] = py elif self.rank[px] > self.rank[py]: self.parent[py] = px else: self.parent[py] = px self.rank[px] += 1 return Truedef critical_connections(n, connections): uf = UnionFind(n) edge_set = set(map(tuple, (map(sorted, connections)))) result = [] for u, v in edge_set: uf_copy = UnionFind(n) for x, y in edge_set: if (x, y) != (u, v): uf_copy.union(x, y) if uf_copy.find(u) != uf_copy.find(v): result.append([u, v]) return result# Example usagen = 4connections = [[0,1],[1,2],[2,0],[1,3]]print(critical_connections(n, connections))This implementation uses a Union-Find data structure to efficiently check the connectivity of the graph. For each edge, we create a copy of the graph without that edge and check if it remains connected. If not, the edge is a critical connection.Time Complexity AnalysisLets compare the time complexities of the two algorithms weve discussed:Tarjans Algorithm: O(V + E), where V is the number of vertices and E is the number of edges. This algorithm is very efficient as it only requires a single pass through the graph.Union-Find Algorithm: O(E * (V + E)), where E is the number of edges and V is the number of vertices. For each edge, we potentially iterate through all other edges, making this approach less efficient for large graphs.For most practical purposes, Tarjans algorithm is preferred due to its linear time complexity.Real-World ApplicationsUnderstanding critical connections has numerous practical applications across various fields:1. Network InfrastructureIn computer networks, identifying critical connections helps in:Designing robust network topologiesPlanning redundancy to prevent single points of failurePrioritizing maintenance and upgrades for crucial links2. Social Network AnalysisIn social networks, bridges can represent:Key individuals connecting different communitiesWeak ties that are crucial for information flow between groupsPotential points of intervention for information dissemination or containment3. Transportation SystemsIn transportation networks, critical connections help in:Identifying crucial roads or bridges for emergency planningOptimizing traffic flow and reducing congestionPrioritizing infrastructure investments4. Biological NetworksIn biological systems, understanding critical connections aids in:Identifying key protein interactions in cellular networksUnderstanding the spread of diseases in populationsAnalyzing ecological networks and food websChallenges and ConsiderationsWhile the concept of critical connections is powerful, there are some challenges and considerations to keep in mind:1. Dynamic NetworksMany real-world networks are dynamic, with connections forming and breaking over time. Algorithms for finding critical connections in dynamic graphs are an active area of research.2. Weighted GraphsIn some scenarios, edges may have different weights or importance. Adapting bridge-finding algorithms to consider edge weights can provide more nuanced insights.3. ScalabilityAs networks grow larger, the computational cost of finding critical connections increases. Developing efficient algorithms for massive graphs remains a challenge.4. Approximation AlgorithmsFor very large graphs, approximation algorithms that identify most, but not all, critical connections might be more practical than exact algorithms.Future DirectionsThe study of critical connections continues to evolve. Some exciting areas for future research include:Quantum Algorithms: Exploring how quantum computing might accelerate the discovery of critical connections in large graphs.Machine Learning Approaches: Developing ML models to predict critical connections based on graph structure and dynamics.Temporal Graphs: Extending the concept of critical connections to graphs that change over time.Hypergraphs: Investigating critical connections in hypergraphs, where edges can connect more than two vertices.ConclusionCritical connections, or bridges, are fundamental concepts in graph theory with wide-ranging applications. From ensuring the robustness of computer networks to understanding social dynamics, the ability to identify these crucial links provides valuable insights across various domains.As weve explored in this article, there are efficient algorithms like Tarjans method to find these connections, each with its own strengths and use cases. Understanding these algorithms and their implementations is crucial for anyone working with network analysis, whether in computer science, social sciences, or biological systems.As networks continue to grow in size and complexity, the importance of identifying and analyzing critical connections will only increase. Whether youre a software engineer designing resilient systems, a data scientist analyzing social networks, or a researcher studying complex biological interactions, the concept of critical connections provides a powerful tool for understanding and optimizing interconnected systems.By mastering these concepts and algorithms, youll be well-equipped to tackle a wide range of problems in network analysis and graph theory. As you continue your journey in coding education and programming skills development, remember that understanding these fundamental algorithms is key to solving complex real-world problems and excelling in technical interviews at top tech companies. Try it on GfG Practice Given an undirected Graph, The task is to find the Bridges in this Graph. An edge in an undirected connected graph is a bridge if removing it disconnects the graph. For a disconnected undirected graph, the definition is similar, a bridge is an edge removal that increases the number of disconnected components. Like Articulation Points, bridges represent vulnerabilities in a connected network and are useful for designing reliable networks. Examples: Input: Output: (0, 3) and (3, 4) Input: Output: (1, 6) Input: Output: (0, 1), (1, 2), and (2, 3) Naive Approach: Below is the idea to solve the problem: One by one remove all edges and see if the removal of an edge causes a disconnected graph. Follow the below steps to Implement the idea: For every edge (u, v), do the following:Remove (u, v) from the graphSee if the graph remains connected (either uses BFS or DFS)Add (u, v) back to the graph.Time Complexity: O(E*(V+E)) for a graph represented by an adjacency list.Auxiliary Space: O(V+E) Find Bridges in a graph using Tarjan's Algorithm.Before heading towards the approach understand which edge is termed as bridge. Suppose there exists a edge from u -> v, now after removal of this edge if v can't be reached by any other edges then u -> v edge is bridge. Our approach is based on this intuition, so take time and grasp it. ALGORITHM: - To implement this algorithm, we need the following data structures - visited[ ] = to keep track of the visited vertices to implement DFSdisc[ ] = to keep track when for the first time that particular vertex is reachedlow[ ] = to keep track of the lowest possible time by which we can reach that vertex 'other than parent' so that if edge from parent is removed can the particular node can be reached other than parent.We will traverse the graph using DFS traversal but with slight modifications i.e. while traversing we will keep track of the parent node by which the particular node is reached because we will update the low[node] = min(low[all it's adjacent node except parent]) hence we need to keep track of the parent. While traversing adjacent nodes let 'v' of a particular node let 'u', then 3 cases arise - 1. v is parent of u then, 2. v is visited then, update the low of u i.e. low[u] = min(low[u] , disc[v]) this arises when a node can be visited by more than one node, but low is to keep track of the lowest possible time so we will update it.3. v is not visited then, call the DFS to traverse aheadnow update the low[u] = min(low[u], low[v]) as we know v can't be parent cause we have handled that case first.now check if (low[v] > disc[u]) i.e. the lowest possible to time to reach 'v' is greater than 'u' this means we can't reach 'v' without 'u' so the edge u -> v is a bridge.Below is the implementation of the above approach: C++ // A C++ program to find bridges in a given undirected graph#includeusing namespace std; // A class that represents an undirected graphclass Graph{ int V; // No. of vertices list *adj; // A dynamic array of adjacency lists void bridgeUtil(int u, vector& visited, vector& disc, vector& low, int parent);public: Graph(int V); // Constructor void addEdge(int v, int w); // to add an edge to graph void bridge(); // prints all bridges}; Graph::Graph(int V){ this->V = V; adj = new list[V];} void Graph::addEdge(int v, int w){ adj[v].push_back(w); adj[w].push_back(v); // Note: the graph is undirected} // A recursive function that finds and prints bridges using// DFS traversal// u --> The vertex to be visited next// visited[] --> keeps track of visited vertices// disc[] --> Stores discovery times of visited vertices// parent[] --> Stores parent vertices in DFS treevoid Graph::bridgeUtil(int u, vector& visited, vector& disc, vector& low, int parent){ // A static variable is used for simplicity, we can // avoid use of static variable by passing a pointer. static int time = 0; // Mark the current node as visited visited[u] = true; // Initialize discovery time and low value disc[u] = low[u] = ++time; // Go through all vertices adjacent to this list::iterator i; for (i = adj[u].begin(); i != adj[u].end(); ++i) { int v = *i; // v is current adjacent of u // 1. If v is parent if(v==parent) continue; //2. If v is visited if(visited[v]){ low[u] = min(low[u], disc[v]); } //3. If v is not visited else{ parent = u; bridgeUtil(v, visited, disc, low, parent); // update the low of u as it's quite possible // a connection exists from v's descendants to u low[u] = min(low[u], low[v]); // if the lowest possible time to reach vertex v // is greater than discovery time of u it means // that v can be only be reached by vertex above v // so if that edge is removed v can't be reached so it's a bridge if (low[v] > disc[u]) cout disc[u]: print ("%d %d" %(u,v)) elif v != parent[u]: # Update low value of u for parent function calls. low[u] = min(low[u], disc[v]) # DFS based function to find all bridges. It uses recursive # function bridgeUtil() def bridge(self): # Mark all the vertices as not visited and Initialize parent and visited, # and ap(articulation point) arrays visited = [False] * (self.V) disc = [float("Inf")] * (self.V) low = [float("Inf")] * (self.V) parent = [-1] * (self.V) # Call the recursive helper function to find bridges # in DFS tree rooted with vertex 'i' for i in range(self.V): if visited[i] == False: self.bridgeUtil(i, visited, parent, low, disc) # Create a graph given in the above diagramg1 = Graph(5)g1.addEdge(1, 0)g1.addEdge(0, 2)g1.addEdge(2, 1)g1.addEdge(0, 3)g1.addEdge(3, 4) print ("Bridges in first graph ")g1.bridge() g2 = Graph(4)g2.addEdge(0, 1)g2.addEdge(1, 2)g2.addEdge(2, 3)print ("Bridges in second graph ")g2.bridge() g3 = Graph (7)g3.addEdge(0, 1)g3.addEdge(1, 2)g3.addEdge(2, 0)g3.addEdge(1, 3)g3.addEdge(1, 4)g3.addEdge(1, 6)g3.addEdge(3, 5)g3.addEdge(4, 5)print ("Bridges in third graph ")g3.bridge() #This code is contributed by Neelam Yadav C# // A C# program to find bridges // in a given undirected graph using System;using System.Collections.Generic; // This class represents a undirected graph // using adjacency list representation public class Graph { private int V; // No. of vertices // Array of lists for Adjacency List Representation private List []adj; int time = 0; static readonly int NIL = -1; // Constructor Graph(int v) { V = v; adj = new List[v]; for (int i = 0; i < v; ++i) adj[i] = new List(); } // Function to add an edge into the graph void addEdge(int v, int w) { adj[v].Add(w); // Add w to v's list. adj[w].Add(v); //Add v to w's list } // A recursive function that finds and prints bridges // using DFS traversal // u --> The vertex to be visited next // visited[] --> keeps track of visited vertices // disc[] --> Stores discovery times of visited vertices // parent[] --> Stores parent vertices in DFS tree void bridgeUtil(int u, bool []visited, int []disc, int []low, int []parent) { // Mark the current node as visited visited[u] = true; // Initialize discovery time and low value disc[u] = low[u] = ++time; // Go through all vertices adjacent to this foreach(int i in adj[u]) { int v = i; // v is current adjacent of u // If v is not visited yet, then make it a child // of u in DFS tree and recur for it. // If v is not visited yet, then recur for it if (!visited[v]) { parent[v] = u; bridgeUtil(v, visited, disc, low, parent); // Check if the subtree rooted with v has a // connection to one of the ancestors of u low[u] = Math.Min(low[u], low[v]); // If the lowest vertex reachable from subtree // under v is below u in DFS tree, then u-v is // a bridge if (low[v] > disc[u]) Console.WriteLine(u + " " + v); } // Update low value of u for parent function calls. else if (v != parent[u]) low[u] = Math.Min(low[u], disc[v]); } } // DFS based function to find all bridges. It uses recursive // function bridgeUtil() void bridge() { // Mark all the vertices as not visited bool []visited = new bool[V]; int []disc = new int[V]; int []low = new int[V]; int []parent = new int[V]; // Initialize parent and visited, // and ap(articulation point) arrays for (int i = 0; i < V; i++) { parent[i] = NIL; visited[i] = false; } // Call the recursive helper function to find Bridges // in DFS tree rooted with vertex 'i' for (int i = 0; i < V; i++) if (visited[i] == false) bridgeUtil(i, visited, disc, low, parent); } // Driver code public static void Main(String []args) { // Create graphs given in above diagrams Console.WriteLine("Bridges in first graph "); Graph g1 = new Graph(5); g1.addEdge(1, 0); g1.addEdge(0, 2); g1.addEdge(2, 1); g1.addEdge(0, 3); g1.addEdge(3, 4); g1.bridge(); Console.WriteLine(); Console.WriteLine("Bridges in Second graph"); Graph g2 = new Graph(4); g2.addEdge(0, 1); g2.addEdge(1, 2); g2.addEdge(2, 3); g2.bridge(); Console.WriteLine(); Console.WriteLine("Bridges in Third graph "); Graph g3 = new Graph(7); g3.addEdge(0, 1); g3.addEdge(1, 2); g3.addEdge(2, 0); g3.addEdge(1, 3); g3.addEdge(1, 4); g3.addEdge(1, 6); g3.addEdge(3, 5); g3.addEdge(4, 5); g3.bridge(); } } // This code is contributed by Rajput-Ji JavaScript // A Javascript program to find bridges in a given undirected graph // This class represents a directed graph using adjacency// list representationclass Graph{ // Constructor constructor(v) { this.V = v; this.adj = new Array(v); this.NIL = -1; this.time = 0; for (let i=0; i The vertex to be visited next // visited[] --> keeps track of visited vertices // disc[] --> Stores discovery times of visited vertices // parent[] --> Stores parent vertices in DFS tree bridgeUtil(u,visited,disc,low,parent) { // Mark the current node as visited visited[u] = true; // Initialize discovery time and low value disc[u] = low[u] = ++this.time; // Go through all vertices adjacent to this for(let i of this.adj[u]) { let v = i; // v is current adjacent of u // If v is not visited yet, then make it a child // of u in DFS tree and recur for it. // If v is not visited yet, then recur for it if (!visited[v]) { parent[v] = u; this.bridgeUtil(v, visited, disc, low, parent); // Check if the subtree rooted with v has a // connection to one of the ancestors of u low[u] = Math.min(low[u], low[v]); // If the lowest vertex reachable from subtree // under v is below u in DFS tree, then u-v is // a bridge if (low[v] > disc[u]) document.write(u+" "+v+""); } // Update low value of u for parent function calls. else if (v != parent[u]) low[u] = Math.min(low[u], disc[v]); } } // DFS based function to find all bridges. It uses recursive // function bridgeUtil() bridge() { // Mark all the vertices as not visited let visited = new Array(this.V); let disc = new Array(this.V); let low = new Array(this.V); let parent = new Array(this.V); // Initialize parent and visited, and ap(articulation point) // arrays for (let i = 0; i < this.V; i++) { parent[i] = this.NIL; visited[i] = false; } // Call the recursive helper function to find Bridges // in DFS tree rooted with vertex 'i' for (let i = 0; i < this.V; i++) if (visited[i] == false) this.bridgeUtil(i, visited, disc, low, parent); }} // Create graphs given in above diagramsdocument.write("Bridges in first graph ");let g1 = new Graph(5);g1.addEdge(1, 0);g1.addEdge(0, 2);g1.addEdge(2, 1);g1.addEdge(0, 3);g1.addEdge(3, 4);g1.bridge();document.write("");document.write("Bridges in Second graph");let g2 = new Graph(4);g2.addEdge(0, 1);g2.addEdge(1, 2);g2.addEdge(2, 3);g2.bridge();document.write("");document.write("Bridges in Third graph ");let g3 = new Graph(7);g3.addEdge(0, 1);g3.addEdge(1, 2);g3.addEdge(2, 0);g3.addEdge(1, 3);g3.addEdge(1, 4);g3.addEdge(1, 6);g3.addEdge(3, 5);g3.addEdge(4, 5);g3.bridge(); // This code is contributed by avanitrachhadiya2155OutputBridges in first graph 3 40 3Bridges in second graph 2 31 20 1Bridges in third graph 1 6Time Complexity: O(V+E), The above approach uses simple DFS along with Tarjan's Algorithm.So time complexity is the same as DFS which is O(V+E) for adjacency list representation of the graph.Auxiliary Space: O(V) is used for visited, disc and low arrays. Graph Algorithms Introduction to Graph Data Structure Graph and its representations Types of Graphs with Examples Basic Properties of a Graph Applications, Advantages and Disadvantages of Graph Transpose graph Difference Between Graph and Tree Breadth First Search or BFS for a Graph Depth First Search or DFS for a Graph Applications, Advantages and Disadvantages of Depth First Search (DFS) Applications, Advantages and Disadvantages of Breadth First Search (BFS) Iterative Depth First Traversal of Graph BFS for Disconnected Graph Transitive Closure of a Graph using DFS Difference between BFS and DFS Detect Cycle in a Directed Graph Detect cycle in an undirected graph Detect Cycle in a directed graph using colors Detect a negative cycle in a Graph | (Bellman Ford) Cycles of length n in an undirected and connected graph Detecting negative cycle using Floyd Warshall Clone a Directed Acyclic Graph A comprehensive guide to bridges, their properties, and applications in graph theory and network analysis Sarah Lee AI generated Llama-4-Maverick-17B-128E-Instruct-FP8 5 min read May 27, 2025 In the realm of graph theory, a bridge is an edge whose removal increases the number of connected components in a graph. In other words, a bridge is a critical connection between two parts of a graph, and its removal can significantly impact the graph's connectivity and reliability.Definition of a Bridge in a GraphFormally, given a graph $G = (V, E)$, an edge $e \in E$ is a bridge if the graph $G' = (V, E \setminus {e})$ has more connected components than G. This definition implies that a bridge is an edge that connects two otherwise disconnected subgraphs.Examples of Bridges in Different Types of GraphsTo illustrate the concept of bridges, consider the following examples:In a simple path graph, every edge is a bridge because removing any edge disconnects the graph.In a cycle graph, no edge is a bridge because removing any edge still leaves the graph connected.In a more complex graph, such as a network of roads or a social network, bridges can be critical connections between different regions or communities.The following Mermaid diagram illustrates a graph with a bridge:graph LR; A --- B; B --- C; C --- D; D --- E; B --- E; A --- D; style B fill:#f9f,stroke:#333,stroke-width:4px; style E fill:#f9f,stroke:#333,stroke-width:4px; linkStyle 1 stroke:#ff3,stroke-width:4px;In this graph, the edge between B and E is not a bridge, but if we remove the edge between C and D, it becomes a bridge.Importance of Bridges in Network AnalysisBridges play a crucial role in network analysis because they can significantly impact the connectivity and reliability of a network. In a communication network, for example, a bridge might represent a critical connection between two otherwise disconnected sub-networks. If this connection is lost, the network becomes fragmented, and communication between the sub-networks is disrupted.Properties of BridgesRelationship Between Bridges and Articulation PointsAn articulation point is a node in a graph that, when removed, increases the number of connected components in the graph. There is a close relationship between bridges and articulation points:If an edge is a bridge, then its endpoints are articulation points, unless the edge is connected to a leaf node (a node with degree 1).However, not all edges connected to articulation points are bridges.Effect of Removing a Bridge on Graph ConnectivityWhen a bridge is removed from a graph, the graph becomes disconnected, and the number of connected components increases. This can have significant consequences for network reliability and communication.Characteristics of Graphs with BridgesGraphs with bridges have certain characteristics that can be useful in identifying and analyzing them:A graph with a bridge is not necessarily a disconnected graph; it can be a connected graph with a critical connection between two subgraphs.A graph with multiple bridges can be more vulnerable to fragmentation and disconnection.Detecting Bridges in GraphsDetecting bridges in graphs is an essential task in network analysis. Several algorithms can be used to identify bridges in graphs, including:Algorithms for Finding Bridges in GraphsDepth-First Search (DFS): DFS is a popular algorithm for traversing graphs and detecting bridges. By keeping track of the discovery time and low value for each node, DFS can identify bridges in a graph.Tarjan's Algorithm: Tarjan's algorithm is a more efficient algorithm for detecting bridges in graphs. It uses a DFS traversal and keeps track of the low value and discovery time for each node to identify bridges.The following is a step-by-step outline of Tarjan's algorithm:Perform a DFS traversal of the graph, starting from an arbitrary node.For each node, keep track of its discovery time and low value.For each edge, check if it is a bridge by comparing the low value of the destination node with the discovery time of the source node.If the low value of the destination node is greater than the discovery time of the source node, then the edge is a bridge.Time and Space Complexity of Bridge Detection AlgorithmsThe time and space complexity of bridge detection algorithms vary depending on the algorithm used:AlgorithmTime ComplexitySpace ComplexityDFS$O(V + E)$$O(V)$Tarjan's Algorithm$O(V + E)$$O(V)$Optimizations for Detecting Bridges in Large GraphsFor large graphs, optimizing bridge detection algorithms is crucial for performance. Some optimizations include:Using a more efficient data structure, such as an adjacency list, to represent the graph.Using parallel processing or distributed computing to speed up the bridge detection process.Using approximation algorithms or heuristics to quickly identify potential bridges.ConclusionIn conclusion, bridges play a critical role in graph theory and network analysis. Understanding the properties and characteristics of bridges is essential for analyzing and optimizing network connectivity and reliability. By using algorithms such as DFS and Tarjan's algorithm, bridges can be detected and identified in graphs. Optimizing these algorithms for large graphs is crucial for performance.References FAQQ: What is a bridge in a graph?A: A bridge is an edge in a graph that, when removed, increases the number of connected components in the graph.Q: How do I detect bridges in a graph?A: You can detect bridges in a graph using algorithms such as Depth-First Search (DFS) or Tarjan's algorithm.Q: What is the time complexity of detecting bridges in a graph?A: The time complexity of detecting bridges in a graph is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph.Q: Why are bridges important in network analysis?A: Bridges are important in network analysis because they can significantly impact the connectivity and reliability of a network. Removing a bridge can fragment the network and disrupt communication between sub-networks. Sarah Lee 2025-05-27 21:57:15 You need to be logged in to add comments. Click here to login. When we delve into graph theory, we often encounter critical notions like bridges and articulation points. These concepts significantly impact how we analyze networks, be it in telecommunications, computer networks, or transportation.What are Bridges in Graphs?A bridge in a graph is an edge that, when removed, increases the number of connected components in the graph. In simpler terms, removing a bridge would separate the graph into two or more distinct parts, indicating that it plays a vital role in maintaining the connectivity of the graph.Example of a Bridge:Consider a simple graph:A - B| \| C| /DIn this scenario, the edge B - C acts as a bridge. If we remove this edge, we find that vertices A, B, and D can no longer connect to vertex C. The graph disconnects, confirming that B - C is a bridge.Identifying Bridges Using Depth-First Search (DFS)A typical way to find bridges in a graph is by utilizing Depth-First Search. Lets break down the approach:Initialization: Maintain a discovery time for each vertex, as well as the lowest discovery time reachable from that vertex.DFS Traversal: As we traverse the graph using DFS, we update the discovery and low values.Bridge Condition: For any edge (u, v):If the lowest vertex reachable from v is greater than the discovery time of u, then (u, v) is a bridge.Heres a concise Java implementation:import java.util.*;public class BridgeFinder { private int time = 0; public void bridgeUtil(int u, boolean visited[], int disc[], int low[], int parent[], List adj) { visited[u] = true; disc[u] = low[u] = ++time; for (int v : adj.get(u)) { if (!visited[v]) { parent[v] = u; bridgeUtil(v, visited, disc, low, parent, adj); low[u] = Math.min(low[u], low[v]); if (low[v] > disc[u]) { System.out.println("Bridge: " + u + " - " + v); } } else if (v != parent[u]) { low[u] = Math.min(low[u], disc[v]); } } } public void findBridges(List adj) { int V = adj.size(); boolean[] visited = new boolean[V]; int[] disc = new int[V]; int[] low = new int[V]; int[] parent = new int[V]; for (int i = 0; i < V; i++) { if (!visited[i]) { bridgeUtil(i, visited, disc, low, parent, adj); } } }}What are Articulation Points?An articulation point (or cut vertex) is a vertex that, when removed, increases the number of connected components in a graph. This means that the removal of an articulation point can disconnect parts of the graph, showcasing its pivotal role.Example of an Articulation Point:Using the same example above, in our graph,A - B| \| C| /DIf we remove vertex B, the graph disconnects. Hence, vertex B is an articulation point.Identifying Articulation Points with DFSThe approach to find articulation points is similar to that used for bridges:Maintain disc[], low[], and a parent tracker.Traverse with DFS and update values.Articulation Point Conditions:If the root has two or more children in the DFS tree, it is an articulation point.For any other vertex v, if low[v] >= disc[u], then u is an articulation point.Here's a Java implementation that demonstrates this:public class ArticulationPointFinder { private int time = 0; public void articulationPointUtil(int u, boolean visited[], int disc[], int low[], int parent[], boolean ap[], List adj) { int children = 0; visited[u] = true; disc[u] = low[u] = ++time; for (int v : adj.get(u)) { if (!visited[v]) { parent[v] = u; children++; articulationPointUtil(v, visited, disc, low, parent, ap, adj); low[u] = Math.min(low[u], low[v]); if (parent[u] == -1 && children > 1) ap[u] = true; if (parent[u] != -1 && low[v] >= disc[u]) ap[u] = true; } else if (v != parent[u]) { low[u] = Math.min(low[u], disc[v]); } } } public void findArticulationPoints(List adj) { int V = adj.size(); boolean[] visited = new boolean[V]; int[] disc = new int[V]; int[] low = new int[V]; int[] parent = new int[V]; boolean[] ap = new boolean[V]; for (int i = 0; i < V; i++) { if (!visited[i]) { articulationPointUtil(i, visited, disc, low, parent, ap, adj); } } for (int i = 0; i < V; i++) { if (ap[i]) System.out.println("Articulation Point: " + i); } }}By utilizing these methods, you can effectively identify bridges and articulation points in any given graph, making your solutions robust and reliable for advanced graph-related problems.

What is a bridge in a graph.