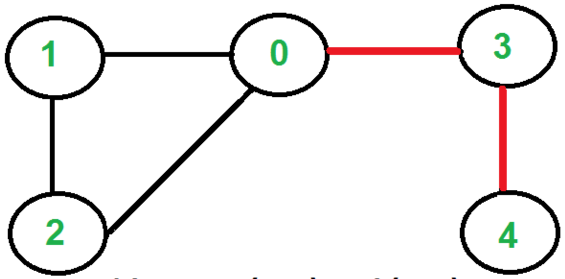
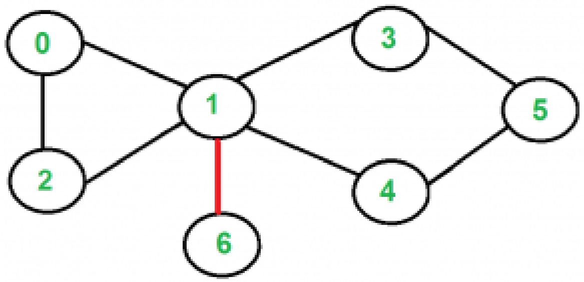
Bridges in a Graph

An edge in an undirected connected graph is a bridge *if and only if* removing it disconnects the graph. For a disconnected undirected graph, the definition is similar, a bridge is an edge removing which increases the number of disconnected components. Like ***Articulation Points*** bridges represent vulnerabilities in a connected network and are useful for designing reliable networks. For example, in a wired computer network, an articulation point indicates the critical computers and a bridge indicates the critical wires or connections.

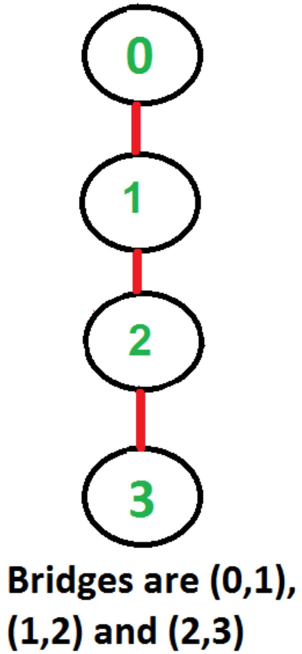
Following are some example graphs with bridges highlighted with red colour:



Bridges are (0, 3) and (3, 4)



Bridge is (1, 6)



How to find all bridges in a given graph?

A simple approach is to one by one remove all edges and see if removal of an edge causes disconnected graph. Following are steps of a simple approach for a connected graph.

1. For every edge (u, v), do following
   - Remove (u, v) from graph.
   - See if the graph remains connected (We can either use BFS or DFS)
   - Add (u, v) back to the graph.

The **time complexity** of the above method is O(E*(V+E)) for a graph represented using adjacency list. Can we do better?

A O(V+E) algorithm to find all Bridges is similar to that of O(V+E) algorithm for Articulation Points. We do DFS traversal of the given graph. In DFS tree an edge (u, v) (u is parent of v in DFS tree) is bridge if there does not exist any other alternative to reach u or an ancestor of u from subtree rooted with v. As discussed in the previous post, the value low[v] indicates earliest visited vertex reachable from subtree rooted with v. *The condition for an edge (u, v) to be a bridge is, "low[v] > disc[u]"*

.Following are C++ and Java implementations of the above approach:

C++ Java

```java
// A Java program to find bridges in a given undirected graph
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using
// adjacency list representation
class Graph
{
    private int V;   // No. of vertices

    // Array  of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];
    int time = 0;
    static final int NIL = -1;

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // 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
```

Dash

All

Articles

Videos

Problems

Quiz

Contest

```
}

// A recursive function that finds and prints bridges
// using DFS traversal
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
void bridgeUtil(int u, boolean 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 aadjacent to this
    Iterator<Integer> i = adj[u].iterator();
    while (i.hasNext())
    {
        int v = i.next();  // 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])
                System.out.println(u+" "+v);
```

Dash

All

Articles

Videos

Problems

Quiz

Contest

```
            }

            // 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
        boolean visited[] = new boolean[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);
    }

    public static void main(String args[])
    {
        // Create graphs given in above diagrams
        System.out.println("Bridges in first graph ");
        Graph g1 = new Graph(5);
        g1.addEdge(1, 0);
```

Dash

All

Articles

Videos

Problems

Quiz

Contest

```
        g1.addEdge(0, 2);
        g1.addEdge(2, 1);
        g1.addEdge(0, 3);
        g1.addEdge(3, 4);
        g1.bridge();
        System.out.println();

        System.out.println("Bridges in Second graph");
        Graph g2 = new Graph(4);
        g2.addEdge(0, 1);
        g2.addEdge(1, 2);
        g2.addEdge(2, 3);
        g2.bridge();
        System.out.println();

        System.out.println("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();
    }
}
```

Output:

```
Bridges in first graph
3 4
0 3

Bridges in second graph
2 3
```

```
1 2
0 1

Bridges in third graph
1 6
```

Time Complexity:
The above function is simple DFS with additional arrays. So time complexity is same as DFS which is O(V+E) for adjacency list representation of graph.

Auxiliary Space: O(V)