Dash

All

Articles

Videos

Problems

Quiz

Contest

« Prev

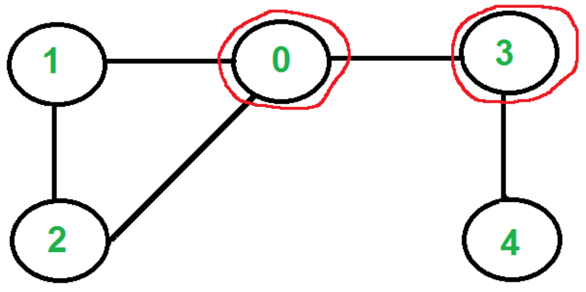
Next »

Articulation Points (or Cut Vertices) in a Graph

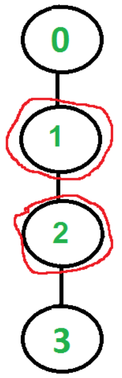
A vertex in an undirected connected graph is an articulation point (or cut vertex) if and only if removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more disconnected components. They are useful for designing reliable networks.

For a disconnected undirected graph, an articulation point is a vertex removing which increases the number of connected components.

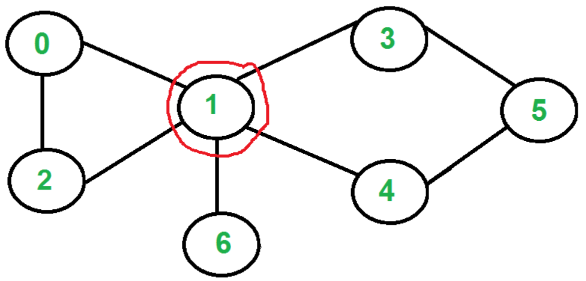
Following are some example graphs with articulation points encircled with red color.



Articulation points are 0 and 3



Artiulation
Points are 1 & 2



Articulation Point is 1

How to find all articulation points in a given graph?

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

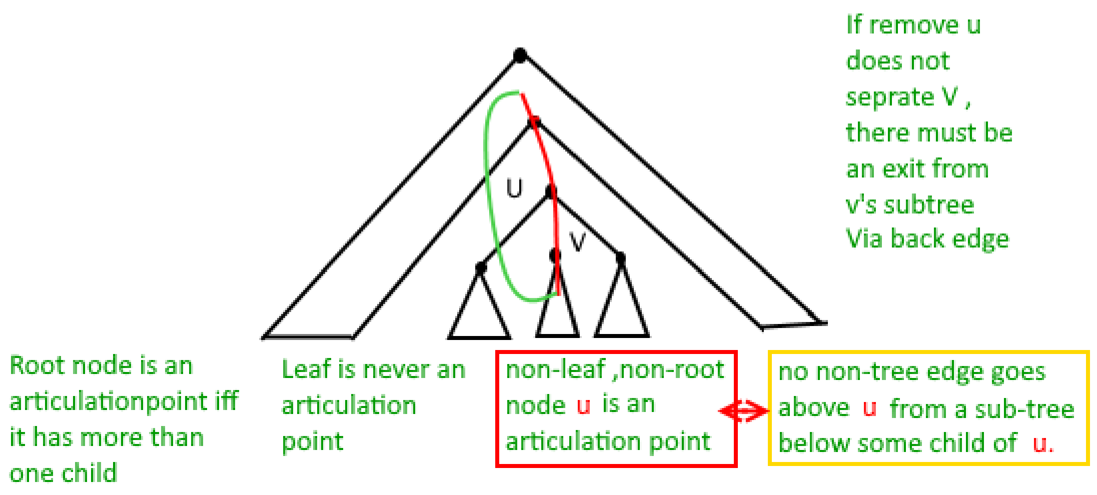
1. For every vertex v, do following:
    - Remove v from graph

- See if the graph remains connected (We can either use BFS or DFS)
- Add v back to the graph

The **time complexity** of the above method is O(V*(V+E)) for a graph represented using adjacency list. Can we do better? **A O(V+E) algorithm to find all Articulation Points (APs)**. The idea is to use DFS (Depth First Search). In DFS, we follow vertices in tree form called DFS tree. In DFS tree, a vertex u is the parent of another vertex v, if v is discovered by u (obviously v is adjacent of u in the graph). In DFS tree, a vertex u is articulation point if one of the following two conditions is true.

1. u is the root of DFS tree and it has at least two children.
2. u is not the root of 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.

The following figure shows the same points as above with one additional point that a leaf in DFS Tree can never be an articulation point.


If remove u does not seprate V , there must be an exit from v's subtree Via back edge
U
V
Root node is an articulationpoint iff it has more than one child
Leaf is never an articulation point
non-leaf ,non-root node u is an articulation point
no non-tree edge goes above u from a sub-tree below some child of u.

We do DFS traversal of the given graph with additional code to find out Articulation Points (APs). In DFS traversal, we maintain a parent[] array where parent[u] stores parent of vertex u. Among the above mentioned two cases, the first case is simple to detect. For every vertex, count children. If currently visited vertex u is root (parent[u] is NIL) and has more than two children, print it. How to handle the second case? The second case is trickier. We maintain an array disc[] to store discovery time of vertices. For every node u, we need to find out the earliest visited vertex (the vertex with minimum discovery time) that can be reached from subtree rooted with u. So we maintain an additional array low[] which is defined as follows.

```
low[u] = min(disc[u], disc[w])
where w is an ancestor of u and there is a back edge from
some descendant of u to w.
```

Following are C++ and Java implementation of Tarjan's algorithm for finding articulation points:

C++ Java

```cpp
// A C++ program to find articulation points
// in an undirected graph
#include<iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
    void APUtil(int v, bool visited[], int disc[], int low[],
                int parent[], bool ap[]);
public:
    Graph(int V);   // Constructor
    void addEdge(int v, int w);   // function to add an edge to graph
    void AP();    // prints articulation points
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[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 find articulation points 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
// ap[] --> Store articulation points
void Graph::APUtil(int u, bool visited[], int disc[],
                                      int low[], int parent[], bool ap[])
{
    // A static variable is used for simplicity,
    // we can avoid the use of static
    // variable by passing a pointer
    static int time = 0;

    // Count of children in DFS Tree
    int children = 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 aadjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        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 (!visited[v])
        {
            children++;
            parent[v] = u;
            APUtil(v, visited, disc, low, parent, ap);

            // Check if the subtree rooted with v has a connection to
            // one of the ancestors of u
            low[u]  = min(low[u], low[v]);

            // u is an articulation point in following cases

            // (1) u is root of DFS tree and has two or more chilren.
            if (parent[u] == NIL && children > 1)
               ap[u] = true;

            // (2) If u is not root and low value of one of its child is more
            // than discovery value of u.
            if (parent[u] != NIL && low[v] >= disc[u])
               ap[u] = true;
        }

        // Update low value of u for parent function calls.
        else if (v != parent[u])
            low[u]  = min(low[u], disc[v]);
    }
}

// The function to do DFS traversal.
// It uses recursive function APUtil()
void Graph::AP()
{
    // 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];
    bool *ap = new bool[V]; // To store articulation points

    // Initialize parent and visited, and ap(articulation point) arrays
    for (int i = 0; i < V; i++)
```

Dash

All

Articles

```
    {
        parent[i] = NIL;
        visited[i] = false;
        ap[i] = false;
    }

    // Call the recursive helper function to find articulation points
    // in DFS tree rooted with vertex 'i'
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            APUtil(i, visited, disc, low, parent, ap);

    // Now ap[] contains articulation points, print them
    for (int i = 0; i < V; i++)
        if (ap[i] == true)
```

On Offers

Courses Tutorials Jobs Practice Contests

Videos

Problems

Quiz

Contest

```
// Driver program to test above function
int main()
{
    // Create graphs given in above diagrams
    cout << "\nArticulation points in first graph \n";
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.AP();

    cout << "\nArticulation points in second graph \n";
    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.AP();

    cout << "\nArticulation points in third graph \n";
    Graph g3(7);
    g3.addEdge(0, 1);
```

Dash

All

Articles

Videos

Problems

Quiz

Contest

```
        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.AP();

        return 0;
    }
```

Output:

```
Articulation points in first graph
0 3
Articulation points in second graph
1 2
Articulation points in third graph
1
```

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

Mark as Read

Report An Issue

If you are facing any issue on this page. Please let us know.