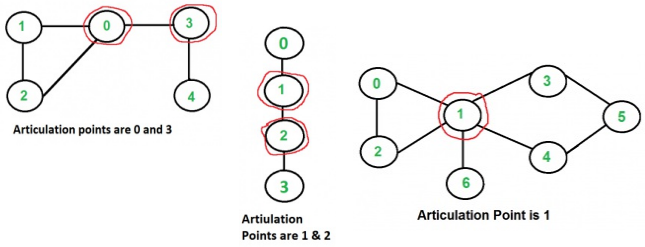
Articulation Points (or Cut Vertices) in a Graph

A vertex in an undirected connected graph is an articulation point (or cut vertex) iff 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 number of connected components.

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



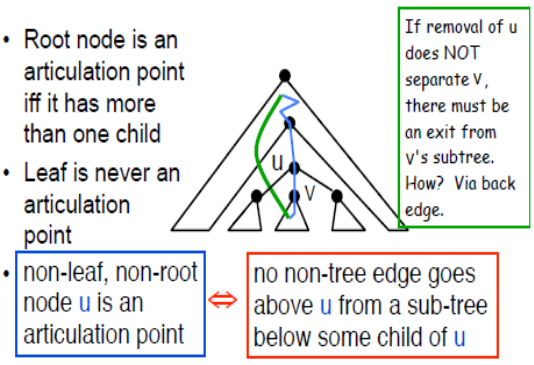
How to find all articulation points in a given graph?
A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graph. Following are steps of simple approach for connected graph.

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

Time complexity of 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 parent of another vertex v, if v is discovered by u (obviously v is an adjacent of u in graph). In DFS tree, a vertex u is articulation point if one of the following two conditions is true.
1) u is root of DFS tree and it has at least two children.
2) u is not root of DFS tree and it has a child v such that no vertex in subtree rooted with v has a back edge to one of the ancestors (in DFS tree) of u.

Following figure shows same points as above with one additional point that a leaf in DFS Tree can never be an articulation point. (Source Ref 2)



We do DFS traversal of 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 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 is C++ implementation of Tarjan's algorithm for finding articulation points.

```
// A C++ program to find articulation points in a given 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 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++)
    {
        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)
            cout << i << " ";
}

// 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);
    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 same as DFS which is O(V+E) for adjacency list representation of graph.

References:
https://www.cs.washington.edu/education/courses/421/04su/slides/artic.pdf
http://www.slideshare.net/TraianRebedea/algorithm-design-and-complexity-course-8
http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L25-Connectivity.htm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

► Graph Component
► Articulation
► Node

Writing code in comment? Please refer here for guidelines.

14 comments

Leave a message...

Newest Community

Avatar **Guest** • 3 days ago

As usual very good explanation. But i am confused about few points.

1. suppose we have an undirected graph like 1-2-3-4-1 (a circle) then according to removal of '1' doesn't divide graph into two components which is basic definition of point ?

2. In this line:

```
if (parent[u] != NIL && low[v] >= disc[u])
        ap[u] = true;
```

when will we have condition for equality in this condition?

```
low[v] >= disc[u]
```

• Reply • Share ›

Avatar **steve** • 3 days ago

How to print biconneted component?

• Reply • Share ›

Avatar **shan** • 2 months ago

"else if(v != parent[u])"
how this statement to equals to back-edge???

• Reply • Share ›

Tu Nguyen ➚ shan • a month ago

because v is visited (the else statement) and v is not a direct cycle. If the "if the if (v != parent[v]) is to make sure that u v is not a direct cycle (You come came to u from v!!)

• Reply • Share ›

sumit ➚ shan • a month ago

yeah , i too will like to understand this part , "else if(v != parent[u])" compute

• Reply • Share ›

Avatar **Yeleswarapu Pavan** • 3 months ago

Another approach is to find all the spanning trees paths and check which points in t paths.
Please verify the correctness of algorithm.

• Reply • Share ›

Avatar **akki** • 5 months ago

I think complexity should be mentioned clearly..If graph is connected and undirected rather than calling it in LOOP for every vertex.
hence it will be O(V+E)..however if graph is disconnected then
we need to call for loop for every vertex since we have disconnected components..
where k = no of disconnected components which in worst case will boil up to O(V*(

1 • Reply • Share ›

Avatar **Avaneesh Kumar** • 5 months ago

Pratibha Kumari tz easy yaar... :P

2 • Reply • Share ›

Avatar **Sarthak Mall 'shanky'** • 5 months ago

In the function void Graph::AP() and given the graph is connected and undirected th ap); for any vertex is sufficient.....

• Reply • Share ›

Avatar **subhajit** • 6 months ago

If we replace this snippet

```
else if (v != parent[u])
    low[u]  = min(low[u], disc[v]);
```

---

with this :

```
else if (v != parent[u])
    low[u]  = min(low[u], low[v]);
```

---

will it not be more appropriate? as the subtree of v is also a subtree of u...though th

• Reply • Share ›

Avatar **GeeksforGeeks** • 6 months ago

@spandan & @NNavneet

Thanks for pointing this out. We have updated the post. Keep it up!

• Reply • Share ›

Avatar **NNavneet** • 6 months ago

I think there will be "i" instead of "0" in the APUtil calling .

Thanks for so easily written and understandble theory for a this problem :)

```
// 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(0, visited, disc, low, parent, ap);
```

• Reply • Share ›

Avatar **spandan** • 6 months ago

In function AP()
the call APUtil(0, visited, disc, low, parent, ap) , the root of call is passed 0 (first par
unvisited node .

• Reply • Share ›

Avatar **Avaneesh Kumar** • 6 months ago

samre concept but easy. auto explanatory... step by step :)

```
/*@author AVANEESH KUMAR, BIET JHANSI, prmrs111@live.com*/.
#include <sstream>
#include <iostream>
#include <cstdio>
#include <cmath>
#include <cstdlib>
#include<vector>

using namespace std;.
// LAZY :) sorry.
typedef long long ll;.
typedef vector <int> vi;.
typedef vector <vi> vvi;.
typedef vector <string> vs;.
typedef pair< int , int > pii;.
typedef vector <ll> vll;.
```

see more

• Reply • Share ›