


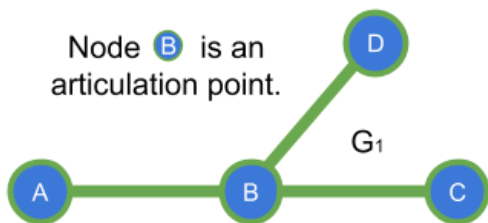
searleser97's blog

Articulation points and bridges (Tarjan's Algorithm)

By [searleser97](#), 4 years ago, 

Articulation Points

Let's define what an *articulation point* is. We say that a vertex V in a graph G with C connected components is an *articulation point* if its removal increases the number of connected components of G . In other words, let C' be the number of connected components after removing vertex V , if $C' > C$ then V is an *articulation point*.



How to find articulation points?

Naive approach $O(V * (V + E))$

For every vertex V in the graph G do
 Remove V from G
 if the number of connected components increases then V is an articulation point
 Add V back to G

Tarjan's approach $O(V + E)$

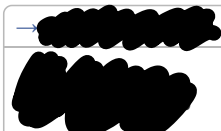
First, we need to know that an ancestor of some node V is a node A that was discovered before V in a DFS traversal. *(thus dependent on DFS implementation, similar to lowlink values)*

In the graph G_1 shown above, if we start our DFS from **A** and follow the path to **C** through **B** ($A \rightarrow B \rightarrow C$), then **A** is an ancestor of **B** and **C** in this spanning tree generated from the DFS traversal.

Example of DFS spanning trees of a graph

→ Pay attention

Before contest
[Codeforces Round 903 \(Div. 3\)](#)
 08:13:09
[Register now »](#)



- [Settings](#)
- [Blog](#)
- [Teams](#)
- [Submissions](#)
- [Talks](#)
- [Contests](#)

→ Top rated

#	User	Rating
1	tourist	3775
2	Radewoosh	3752
3	Benq	3724
4	orzdevinwang	3697
5	jiangly	3627
6	cnnfls_csy	3602
7	-0.5	3545
8	inaFStream	3478
9	fantasy	3468
10	maroonrk	3427

[Countries](#) | [Cities](#) | [Organizations](#)
[View all →](#)

→ Top contributors

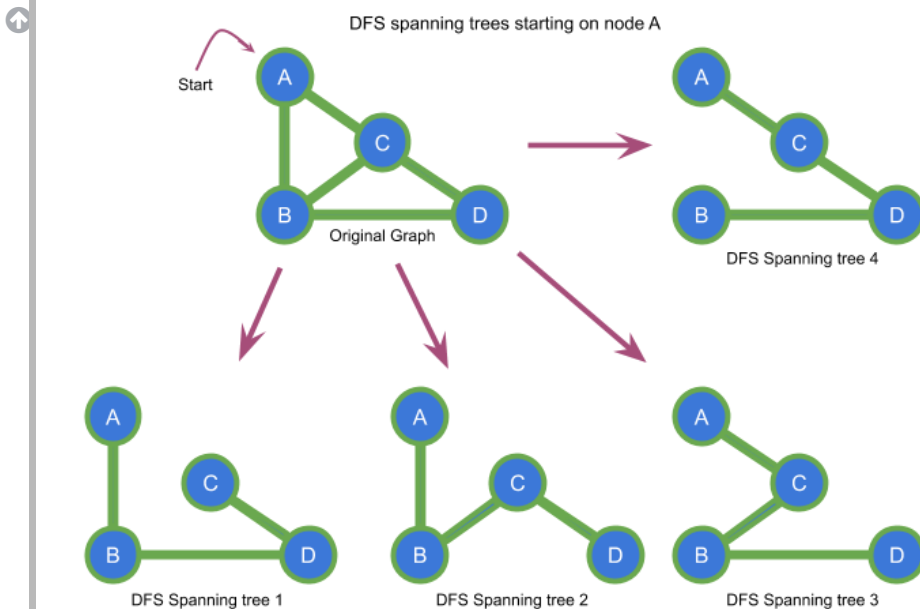
#	User	Contrib.
1	adamant	178
2	awoo	168
3	BledDest	165
4	Um_nik	163
5	maroonrk	162
6	SecondThread	159
7	nor	158
8	-is-this-fft-	152
9	kostka	145
10	Geothermal	144

[View all →](#)

→ Find user

Handle:

Find



Now that we know the definition of **ancestor** let's dive into the main idea.

Idea

Let's say there is a node V in some graph G that can be reached by a node U through some intermediate nodes (maybe non intermediate nodes) following some DFS traversal, if V can also be reached by $A = \text{"ancestor of } U\text{"}$ without passing through U then, U is NOT an articulation point because it means that if we remove U from G we can still reach V from A , hence, the number of connected components will remain the same.

So, we can conclude that the only 2 conditions for U to be an *articulation point* are:

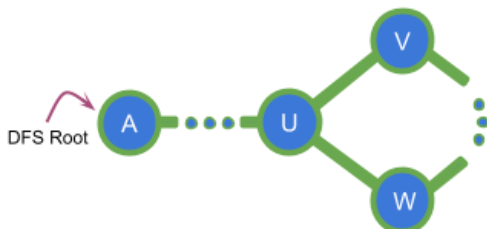
1. If all paths from A to V require U to be in the graph.
2. If U is the root of the DFS traversal with at least 2 children subgraphs disconnected from each other.

Then we can break condition #1 into 2 subconditions:

- U is an *articulation point* if it does not have an adjacent node V that can reach A without requiring U to be in G .



- U is an *articulation point* if it is the root of some cycle in the DFS traversal.



Examples:



Here **B** is an articulation point because all paths from ancestors of **B** to **C** require **B** to be in the graph.

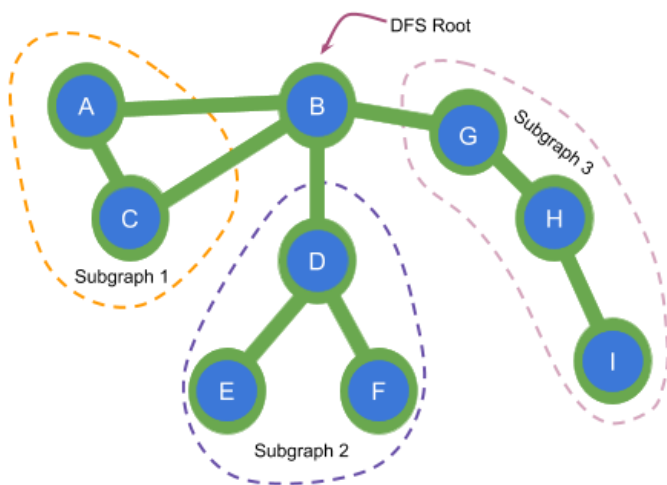
Recent actions

- [mfv](#) → [Experimental Educational Round: VolBIT Formulas Blitz](#)
- [wuhudsm](#) → [Invitation to TheForces Round #24 \(DIV3-Forces, TheForces-Rated, Prizes!\)](#)
- [Vladosiya](#) → [Codeforces Round #903 \(Div. 3\)](#)
- [DioHERO](#) → [Why so less "RED" coders in India?](#)
- [Aphrim](#) → [5 steps to reach expert](#)
- [TheScrasse](#) → [Italian Olympiad in Informatics \(OII\) 2023 — Online Contest](#)
- [Hizenberg](#) → [Help needed in a question. \(Right Triangles\)](#)
- [Nil_paracetamol](#) → [All Div-4 Contest link](#)
- [Pyqe](#) → [Codeforces Round #902 \(Div. 1, Div. 2, based on COMPFEST 15 — Final Round\) Editorial](#)
- [awoo](#) → [Educational Codeforces Round 156 Editorial](#)
- [maroonrk](#) → [AtCoder Regular Contest 166 Announcement](#)
- [MR.MICROSOFT](#) → [STL \(Standard Template Library\) in C++ is](#)
- [Satoru](#) → [Codeforces blocking my account for a fake cheating accuse](#)
- [SecondThread](#) → [Meta Hacker Cup 2023 Round 1](#)
- [greenhand.n](#) → [Why I can't find c++11 in the submit Language?](#)
- [prabowo](#) → [Travel Concerns for The Upcoming ICPC World Finals](#)
- [glebustim](#) → [CodeTON Round 6 \(Div. 1 + Div. 2, Rated, Prizes!\)](#)
- [_skb_](#) → [Replay of Battle of Brains 2022, University of Dhaka](#)
- [aman.acid.118ml](#) → [Help needed in a AtCoder problem](#)
- [DhruvBohara](#) → [Can anyone help in this Q? I thought of BS on search space but couldn't write isPossible function](#)
- [tmwilliamlin168](#) → [Problem from Taiwan Regional Contest](#)
- [vioalbert](#) → [Invitation to TLX Regular Open Contest #34 \(based on BNPCHS 2023 Final Round\)](#)
- [SlavicG](#) → [EGOI 2024 Call for Problems](#)
- [ElectroMaster3](#) → [I've had enough](#)
- [YogeshZT](#) → [Binary Lifting](#)

[Detailed →](#)



Here **B** is NOT an articulation point because there is at least one path from an ancestor of **B** to **C** which does not require **B**.



Here **B** is an articulation point since it has at least 2 children subgraphs disconnected from each other.

Implementation

Well, first thing we need is a way to know if some node A is ancestor of some other node V , for this we can assign a *discovery time* to each vertex V in the graph G based on the DFS traversal, so that we can know which node was discovered before or after another. e.g. in G_1 with the traversal $A \rightarrow B \rightarrow C$ the discovery times for each node will be respectively 1, 2, 3; with this we know that **A** was discovered before **C** since `discovery_time[A] < discovery_time[C]`.

Now we need to know if some vertex U is an articulation point. So, for that we will check the following conditions:

1. If there is NO way to get to a node V with **strictly** smaller discovery time than the discovery time of U following the DFS traversal, then U is an articulation point. (it has to be **strictly** because if it is equal it means that U is the root of a cycle in the DFS traversal which means that U is still an *articulation point*).
2. If U is the root of the DFS tree and it has at least 2 children subgraphs disconnected from each other, then U is an articulation point.

So, for implementation details, we will think of it as if for every node U we have to find the node V with the least discovery time that can be reached from U following some DFS traversal path which does not require to pass **through** any already visited nodes, and let's call this node *low*.

Then, we can know that U is an articulation point if the following condition is satisfied:

`discovery_time[U] <= low[V]` (V in this case represents an adjacent node of U).

To implement this, in each node V we will store some identifier of its corresponding node *low* found, this identifier will be the corresponding *low*'s discovery time because it helps us to know when the node *low* was discovered, hence it helps us to know by which node we can discover U first.

C++ Code

```
// adj[u] = adjacent nodes of u
// ap = AP = articulation points
// p = parent
// disc[u] = discovery time of u
// low[u] = 'low' node of u
```

```

int dfsAP(int u, int p) {
    int children = 0;
    low[u] = disc[u] = ++Time;
    for (int& v : adj[u]) {
        if (v == p) continue; // we don't want to go back through the same
        path.
                                // if we go back is because we found another
way back
        if (!disc[v]) { // if V has not been discovered before
            children++;
            dfsAP(v, u); // recursive DFS call
            if (disc[u] <= low[v]) // condition #1
                ap[u] = 1;
            low[u] = min(low[u], low[v]); // low[v] might be an ancestor of u
        } else // if v was already discovered means that we found an ancestor
            low[u] = min(low[u], disc[v]); // finds the ancestor with the least
discovery time
        }
    }
    return children;
}

void AP() {
    ap = low = disc = vector<int>(adj.size());
    Time = 0;
    for (int u = 0; u < adj.size(); u++)
        if (!disc[u])
            ap[u] = dfsAP(u, u) > 1; // condition #2
}

```



Bridges

Let's define what a *bridge* is. We say that an edge UV in a graph G with C connected components is a *bridge* if its removal increases the number of connected components of G . In other words, let C' be number of connected components after removing edge UV , if $C' > C$ then the edge UV is a *bridge*.

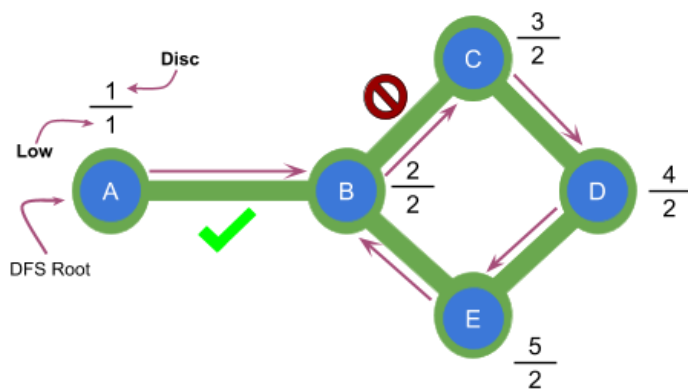
The idea for the implementation is exactly the same as for *articulation points* except for one thing, to say that the edge UV is a bridge, the condition to satisfy is:

`discovery_time[U] < low[V]` instead of `discovery_time[U] <= low[V]`.

Notice that the only change was comparing strictly lesser instead of lesser of equal.

But why is this ?

If `discovery_time[U]` is equal to `low[V]` it means that there is a path from V that goes back to U (V in this case represents an adjacent node of U), or in other words we can just say that we found a cycle rooted in U . For *articulation points* if we remove U from the graph it will increase the number of connected components, but in the case of *bridges* if we remove the edge UV the number of connected components will remain the same. For *bridges* we need to be sure that the edge UV is not involved in any cycle. A way to be sure of this is just to check that `low[V]` is strictly greater than `discovery_time[U]`.



In the graph shown above the edge AB is a *bridge* because $low[B]$ is strictly greater than $disc[A]$. The edge BC is not a *bridge* because $low[C]$ is equal to $disc[B]$.

C++ Code

// br = bridges, p = parent

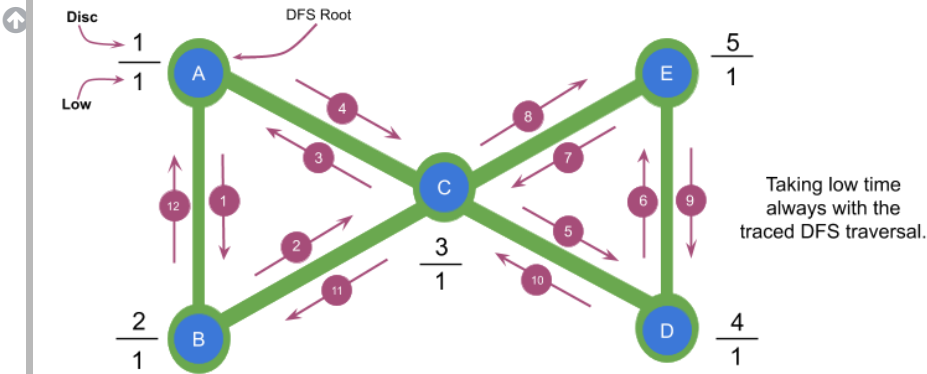
`vector<pair<int, int>> br;`

```
int dfsBR(int u, int p) {
    low[u] = disc[u] = ++Time;
    for (int& v : adj[u]) {
        if (v == p) continue; // we don't want to go back through the same
        path.
        // if we go back is because we found another
        way back
        if (!disc[v]) { // if V has not been discovered before
            dfsBR(v, u); // recursive DFS call
            if (disc[u] < low[v]) // condition to find a bridge
                br.push_back({u, v});
            low[u] = min(low[u], low[v]); // low[v] might be an ancestor of u
        } else // if v was already discovered means that we found an ancestor
            low[u] = min(low[u], disc[v]); // finds the ancestor with the least
            discovery time
    }
}
```

```
void BR() {
    low = disc = vector<int>(adj.size());
    Time = 0;
    for (int u = 0; u < adj.size(); u++)
        if (!disc[u])
            dfsBR(u, u)
}
```

FAQ

- Why $low[u] = \min(low[u], disc[v])$ instead of $low[u] = \min(low[u], low[v])$?



Let's consider node **C** in the graph above, in the DFS traversal the nodes after **C** are: **D** and **E**, when the DFS traversal reaches **E** we find **C** again, if we take its *low* time, `low[E]` will be equal to `disc[A]` but at this point, when we return back to **C** in the DFS we will be omitting the fact that **C** is the **root of a cycle** (which makes it an *articulation point*) and we will be saying that there is a path from **E** to some ancestor of **C** (in this case **A**) which does not require **C** and such path does not exist in the graph, therefore the algorithm will say that **C** is NOT an *articulation point* which is totally false since the only way to reach **D** and **E** is passing through **C**.

Problems

- 315 Network (Points)
- 610 Street Directions (Bridges)
- 796 Critical Links (Bridges)
- 10199 Tourist Guide (Points)
- 10765 Doves and Bombs (Points)

graphs, articulation points, strongly connected, connected components, #tarjan, tutorial, bridges

+76

searleser97

4 years ago

57



Comments (30)

☐ Show archived | [Write comment?](#)



alaneos777

4 years ago, # | ☆
+TREE(G(64)) prro :v □
→ [Reply](#)

0



Jorgelbanez

4 years ago, # | ☆
The best explanation that I found, i've been looking for someone who could explain this subject, and I think I found that guy. +10 and to favorite.
→ [Reply](#)

+3



MZuenni

4 years ago, # | ☆
can you also explain how to find the biconnected components?
→ [Reply](#)

0



SPyofgame

4 years ago, # | ☆
Good tutorial with nice explanations and examples
→ [Reply](#)

0