VNU-HCM University of Science

Data Structure

CS163

ARTICULATION POINTS AND BRIDGES

23APCS02
Pham Gia Hung Khoa
Pham Nguyen Anh Tai

8th August 2024

Contents

1. Introduction **2**

2. Identifying Bridges and Articulation Points **2**
- 2.1 Naive algorithm $O(NM)$ 2
- 2.2 DFS Tree 2
- 2.3 How to find Bridges and Articulation Points 4
- 2.4 Tarjan's Algorithm 4
  - 2.4.1 Robert Tarjan 4
  - 2.4.2 The algorithm 4

3. Applications **5**
- 3.1 Network Reliability and Vulnerability Analysis 5
- 3.2 Transportation Networks 5

4. Additional Insights **5**
- 4.1 Biconnected Components 5
- 4.2 Block-cut Tree 5
- 4.3 Two-edge Connected Component 5
- 4.4 Online Bridges 5

5. Reference **6**

1 Introduction

In graph theory, understanding the structural integrity of networks is crucial. Bridges are edges whose removal increases the number of disconnected components in a graph, acting as critical links. Articulation points are vertices that, if removed, also increase the number of disconnected components, serving as essential hubs. Identifying these elements helps improve network design and resilience, whether in communication networks, transportation systems, social networks, or biological systems. By pinpointing these vulnerabilities, we can create more robust and efficient networks across various domains.

2 Identifying Bridges and Articulation Points

2.1 Naive algorithm $O(NM)$

For each edge $e \in E$, remove it from the graph and count the number of connected components C', if $C' > C$ (the initial number of connected components), then we can claim that e is a bridge, otherwise it is not a bridge.

The similar algorithm for finding the articulation points, which tests each removal of the vertex v, whether it increases the number of connected components of the graph. Which results in a $O(N^2)$ time complexity.

Can we do better?

2.2 DFS Tree

To understand how we can effectively find the bridges and articulation points in terms of time complexity, we need to dive into the concept of DFS Tree.

Without loss of generality, let's assume that the graph G is connected. In general, DFS Tree is a spanning tree of the graph when we do the DFS traversal of the graph.

Consider an undirected connected graph G. Let's run a depth-first traversal of the graph. It can be implemented by a recursive function like this:

```
 function visit(u)
     mark u as visited
     for each vertex v among the neighbours of u:
         if v is not visited:
             mark the edge u-v
             invoke visit(v)
```

[animation and demonstration]

This is the DFS tree of our graph:

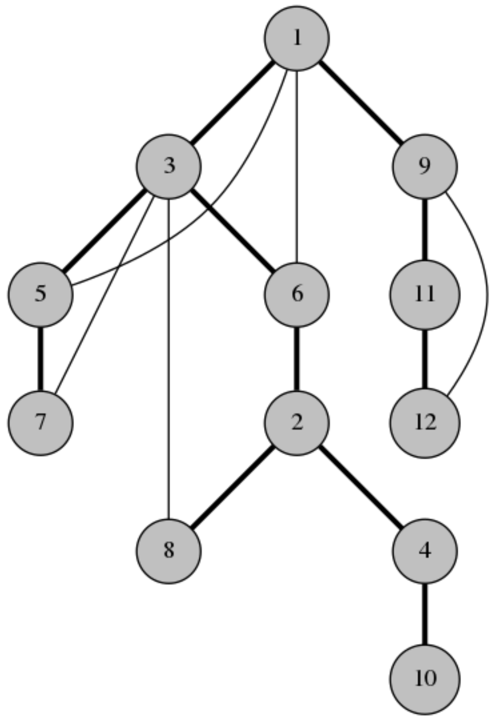

Figure 1: DFS Tree Illustration.

Let's look at all edges that were marked in line 5. They form a spanning tree of G, rooted at the vertex 1. We will call these edges *span-edges*, all other edges are *back-edges*.

Then we have the following observation:

Observation: The back-edges of the graph all connect a vertex with its descendant in the spanning tree. **This is why DFS tree is so useful.**

Why? – Suppose that there is an edge uv, and without loss of generality the depth-first traversal reaches u while v is still unexplored. Then:

- If the depth-first traversal goes to v from u using uv, then uv is a span-edge.
- If the depth-first traversal doesn't go to v from u using uv, then v was already visited when the traversal looked at it at step 4. Thus it was explored while exploring one of the other neighbours of u, which means that v is a descendent of u in the DFS tree.

For example in the graph above, vertices 4 and 8 couldn't possibly have a back-edge connecting them because neither of them is an ancestor of the other. If there was an edge between 4 and 8, the traversal would have gone to 8 from 4 instead of going back to 2.

This is the most important observation about the DFS tree. The DFS tree is so useful because it simplifies the structure of a graph. Instead of having to worry about all kinds of edges, we only need to care about a tree and some additional ancestor-descendant edges. This structure is so much easier to think and write algorithms about.

2.3 How to find Bridges and Articulation Points

From this observation, we can come up with the following intuitive dynamic programming algorithm:

Let $dp[u] = \#\textit{of back-edges that connects u or its descendants to its ancestors}$, then $dp[u]$ can be calculated using this formula:

$$\begin{aligned} dp[u] &= \#\ \textit{of back-edges going from u to its ancestors} \\ &\quad - \#\ \textit{of back-edges going from u to its descendants} + \sum_{v \in child(u)} dp[v] \end{aligned}$$

Using this definition of dynamic programming approach, we can check if an edge u->v in DFS tree is a bridge or not by checking whether $dp[v] = 0$. If it is true, then u->v is a bridge, otherwise it is not.

The same techniques can be used to check whether a vertex v is an articulation point or not by checking the condition $dp[v] = 0$. But there is a special case that v is the root of the DFS Tree, in which $dp[root]$ is always equal to 0, then we must check whether the root contains at least two children.

2.4 Tarjan's Algorithm

2.4.1 Robert Tarjan

Robert Endre Tarjan is an American computer scientist renowned for his contributions to the field of algorithms and data structures. He has made significant advances in graph theory, network analysis, and optimization, many of which are foundational in both theoretical and applied computer science. Tarjan is particularly famous for his development of efficient graph algorithms, some of which are used to solve problems like finding strongly connected components, shortest paths, and minimum spanning trees.

2.4.2 The algorithm

Tarjan suggested a more formal algorithm to find the articulation points and bridges. The algorithm based firmly on the previous observation:

- $num[u]$ = The first time it is visited by the DFS.
- $low[u]$ = The mininum of $num[u]$ that can be reached by going from u to any node in its sub-trees and using **at most one** back-edge.

To calculate these two 2 arrays, we will use Depth-first Search technique to calculate it as follows:

- $num[u]$ will be calculated using a timer, or a variable called cnt. This timer will increase by one whenever we invoke a step of $dfs()$. We set $num[u] = ++cnt$
- Initially: $low[u] = num[u]$. For each edge (u, v) (v is not parent of u):
  - (a) If (u, v) is a span-edge: $low[u] = min(low[u], low[v])$
  - (b) If (u, v) is a back-edge: $low[u] = min(low[u], num[v])$

After computing the num and low arrays, we could identify the bridges and articulation points in our graph:

- A span-edge (u, v) (u is parent of v in DFS Tree) is a bridge if $low[v] > num[u]$ (or $low[v] = num[v]$ or $low[v] >= num[v]$).
- A vertex u is a cut if $low[v] \geq num[u]$ for any span-edge (u, v).

3 Applications

3.1 Network Reliability and Vulnerability Analysis

Bridges: Identifying bridges helps in understanding critical connections whose failure would disrupt network connectivity.

Articulation Points: Identifying these points helps in determining critical nodes whose failure would fragment the network.

Therefore, by understanding how weak or strong one network is, we could reinforce it via redundant paths or nodes to improve its fault tolerance, thus ensuring connectivity within the network itself.

3.2 Transportation Networks

Infrastructure Planning: Identifying bridges and articulation points in transportation networks (e.g., roads, railways) helps in planning for alternative routes and ensuring smooth transportation even if some parts of the network fail.

Emergency Response: Critical nodes and edges can be prioritized for maintenance and quick repairs to minimize disruption during emergencies.

4 Additional Insights

4.1 Biconnected Components

A Biconnected Component (BCC) in a graph is a maximal subgraph in which any two vertices are connected by at least two disjoint paths. This means that there are no single vertices whose removal would disconnect the subgraph. In other words, a BCC is a portion of the graph where connectivity is maintained even if any one vertex is removed.

4.2 Block-cut Tree

Block-Cut Tree (BCT) is a tree structure that represents the decomposition of a graph into its biconnected components (blocks) and articulation points (cut vertices). In this tree, each block and each cut vertex is a node. Edges in the Block-Cut Tree connect cut vertices to the blocks they belong to, providing a clear hierarchical view of the graph's connectivity

4.3 Two-edge Connected Component

A two-edge connected component (also known as a 2-edge-connected component) in a graph is a maximal subgraph in which any two vertices are connected by at least two edge-disjoint paths. This means that there is no single edge whose removal would disconnect the subgraph.

A Bridge Tree is a tree structure derived from a graph, **where each node represents a two-edge connected component (2ECC)** of the original graph, and edges between nodes in the tree correspond to bridges (cut-edges) in the original graph that connects these components.

4.4 Online Bridges

Bridges can be found at the time of building the graph by the following idea:

- Initially we have the graph of n vertices and no edges.
- For each edge (u, v) to be added there can be 3 cases:
  (a) u and v are in the same 2-edges-connected component: The number of bridges does not change.
  (b) u and v are in completely different components: The number of bridges increases by one.

(c) u and v are in the same connected component but different 2-edge-connected component: Creates cycle with one or more previous bridges. The number of bridges decreases by one or more

5 Reference