Lecture 16: Topological Sort / Connectivity (1997)

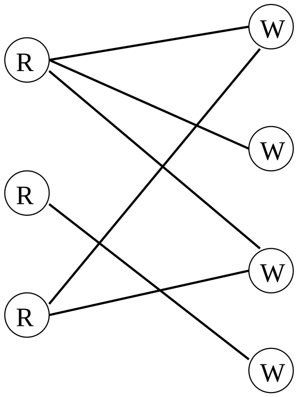
Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.sunysb.edu/∼skiena

Give an efficient algorithm to test if a graph is bipartite.

Bipartite means the vertices can be colored red or black such that no edge links vertices of the same color.



Suppose we color a vertex red - what color must its neighbors be? *black!*

We can augment either BFS or DFS when we first discover a new vertex, color it opposited its parents, and for each other edge, check it doesn’t link two vertices of the same color. The first vertex in any connected component can be red or black!

Bipartite graphs arise in many situations, and special algorithms are often available for them. What is the interpretation of a bipartite “had-sex-with” graph?

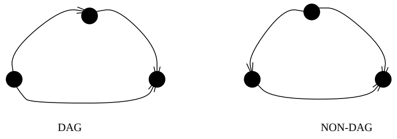
How would you break people into two groups such that no group contains a pair of people who hate each other?

Give an $O(n)$ algorithm to test whether an undirected graph contains a cycle.

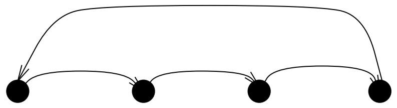
If you do a DFS, you have a cycle iff you have a back edge. This gives an $O(n+m)$ algorithm. But where does the m go? If the graph contains more than $n-1$ edges, it must contain a cycle! Thus we never need look at more than n edges if we are given an adjacency list representation!

Topological Sorting

A directed, acyclic graph is a directed graph with no directed cycles.



A topological sort of a graph is an ordering on the vertices so that all edges go from left to right.
Only a DAG can have a topological sort.



Any DAG has (at least one) topological sort.

Applications of Topological Sorting

Topological sorting is often useful in scheduling jobs in their proper sequence. In general, we can use it to order things given constraints, such as a set of left-right constraints on the positions of objects.

Example: Dressing schedule from CLR.

Example: Identifying errors in DNA fragment assembly.

Certain fragments are constrained to be to the left or right of other fragments, unless there are errors.

```
A B R A C
A C A D A
A D A B R
D A B R A
R A C A D
```

```
A B R A C A D A B R A
---------------------
A B R A C
    R A C A D
      A C A D A
          A D A B R
            D A B R A
```

Solution – build a DAG representing all the left-right constraints. Any topological sort of this DAG is a consistant ordering. If there are cycles, there must be errors.

A DFS can test if a graph is a DAG (it is iff there are no back edges - forward edges are allowed for DFS on directed graph).

Algorithm

Theorem: Arranging vertices in decreasing order of DFS finishing time gives a topological sort of a DAG.

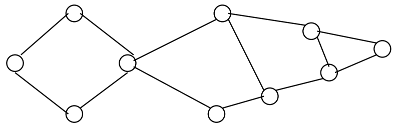
Proof: Consider any directed edge u, v, when we encounter it during the exploration of vertex u:

- If v is white - we then start a DFS of v before we continue with u.
- If v is grey - then u, v is a back edge, which cannot happen in a DAG.
- If v is black - we have already finished with v, so $f[v] < f[u]$.

Thus we can do topological sorting in $O(n + m)$ time.

Articulation Vertices

Suppose you are a terrorist, seeking to disrupt the telephone network. Which station do you blow up?



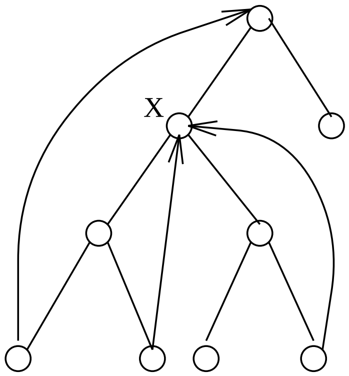
An *articulation vertex* is a vertex of a connected graph whose deletion disconnects the graph.

Clearly connectivity is an important concern in the design of any network.

Articulation vertices can be found in $O(n(m+n))$ – just delete each vertex to do a DFS on the remaining graph to see if it is connected.

A Faster $O(n+m)$ DFS Algorithm

Theorem: In a DFS tree, a vertex v (other than the root) is an articulation vertex iff v is not a leaf and some subtree of v has no back edge incident until a proper ancestor of v.



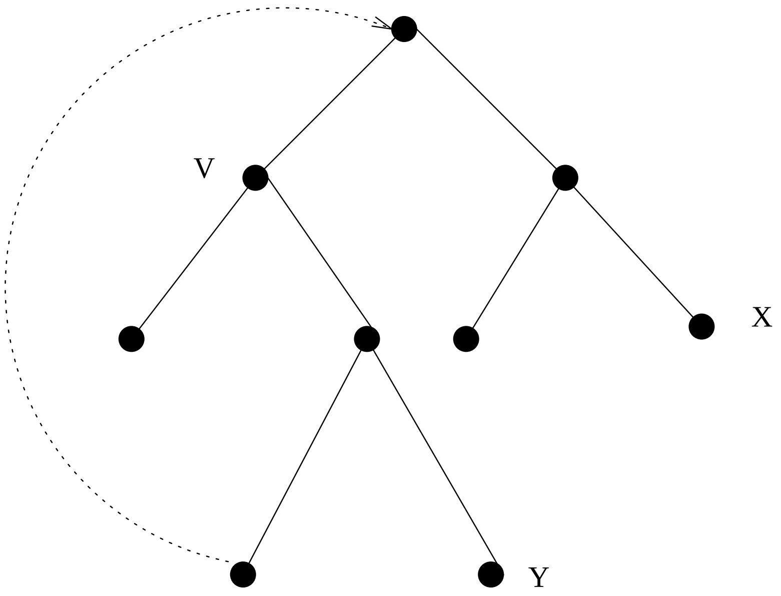
The root is a special case since it has no ancestors.

X is an articulation vertex since the right subtree does not have a back edge to a proper ancestor.

Leaves cannot be articulation vertices

Proof: (1) v is an articulation vertex $\longrightarrow$ v cannot be a leaf.

Why? Deleting v must seperate a pair of vertices x and y. Because of the other tree edges, this cannot happen unless y is a decendant of v.

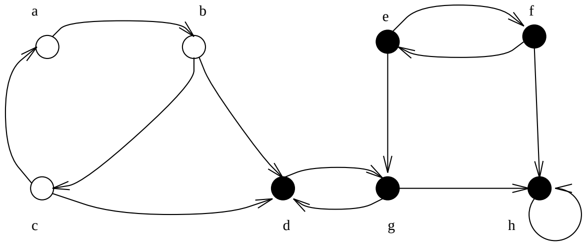


v separating x, y implies there is no back edge in the subtree

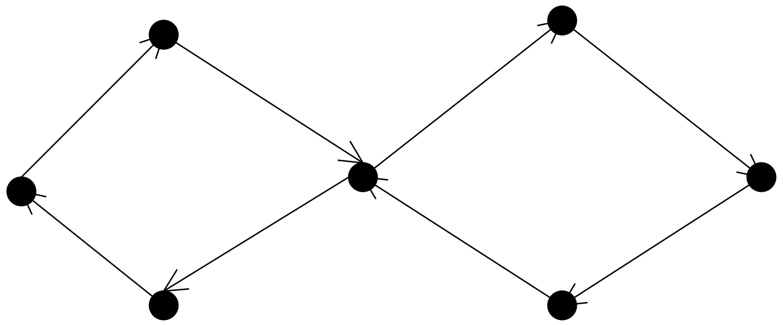
of y to a proper ancestor of v.
(2) Conditions $\rightarrow$ v is a non-root articulation vertex. v separates any ancestor of v from any decendant in the appropriate subtree.
Actually implementing this test in $O(n + m)$ is tricky – but believable once you accept this theorem.

Strongly Connected Components

A directed graph is strongly connected iff there is a directed path between any two vertices.
The strongly connected components of a graph is a partition of the vertices into subsets (maximal) such that each subset is strongly connected.



Observe that no vertex can be in two maximal components, so it is a partition.



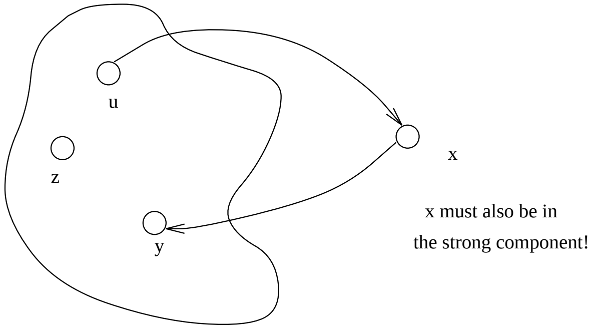
There is an amazingly elegant, linear time algorithm to find the strongly connected components of a directed graph, using DFS.

- Call DFS(σ) to compute finishing times for each vertex.
- Compute the transpose graph G^T (reverse all edges in G)
- Call DFS(G^T), but order the vertices in decreasing order of finish time.

- The vertices of each DFS tree in the forest of DFS(G^T) is a strongly connected component.

This algorithm takes $O(n + m)$, but why does it compute strongly connected components?

Lemma: If two vertices are in the same strong component, no path between them ever leaves the component.

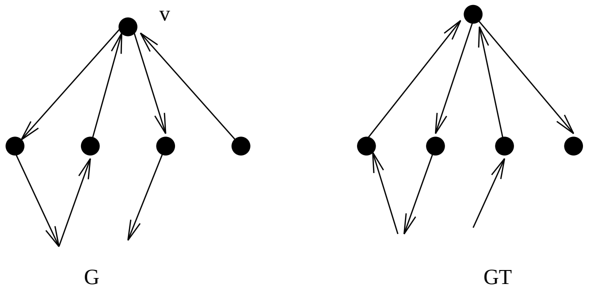


Lemma: In any DFS forest, all vertices in the same strongly connected component are in the same tree.

Proof: Consider the first vertex v in the component to be discovered. Everything in the component is reachable from it, so we will traverse it before finishing with v.

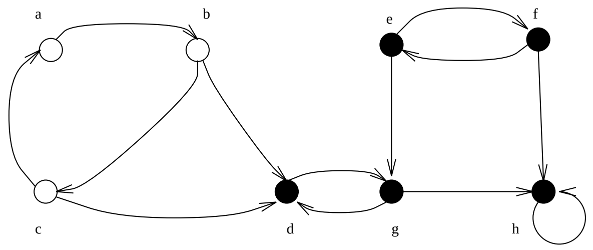
What does DFS(G^T, v) Do?

It tells you what vertices have directed paths to v, while DFS(σ,v) tells what vertices have directed paths from v. But why must any vertex in the search tree of DFS(G^T, v) also have a path from u?

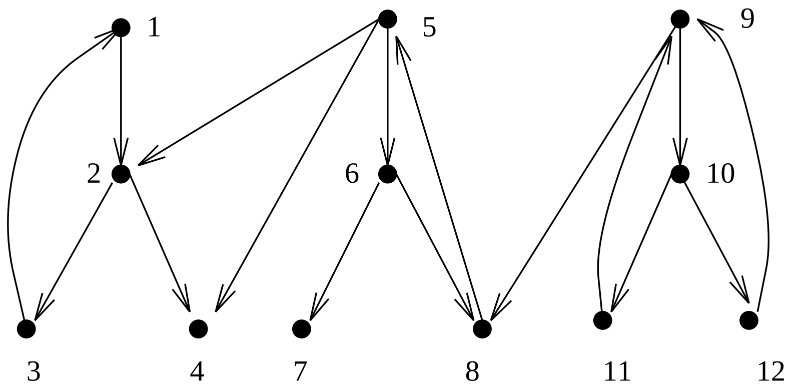


Because there is no edge from any previous DFS tree into the last tree!! Because we ordered the vertices by decreasing order of finish time, we can peel off the strongly connected components from right to left just be doing a DFS(G^T).

Example of Strong Components Algorithm



$9, 10, 11, 12$ can reach 9, oldest remaining finished is 5.
$5, 6, 8$ can reach 5, oldest remaining is 7.
7 can reach 7, oldest remaining is 1.
$1, 2, 3$ can reach 1, oldest remaining is 4.
4 can reach 4.



DFG(G)   9 is the last vertex to finish