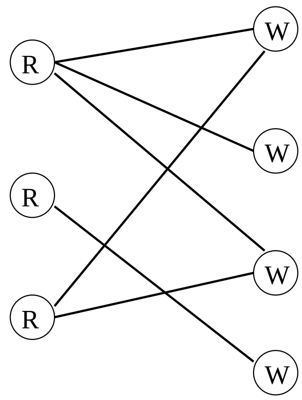
23.2-6 Give an efficient algorithm to test if a graph is bipartite.

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?

23.4-3 Given 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!

23.4-5 Show that you can topologically sort in $O(n+m)$ *by repeatedly deleting vertices of degree* 0*.*

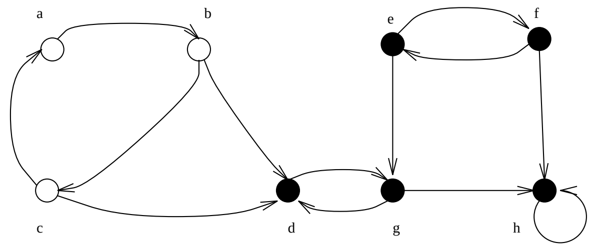
The correctness of this algorithm follows since in a DAG there must always be a vertex of indegree 0, and such a vertex can be first in topological sort. Suppose each vertex is initialized with its indegree (do DFS on G to get this). Deleting a vertex takes O(degree v). Reduce the indegree of each efficient vertex - and keep a list of degree-0 vertices to delete next.

Time: $\sum_{i=1}^{n} O(deg(v_i)) = O(n+m)$

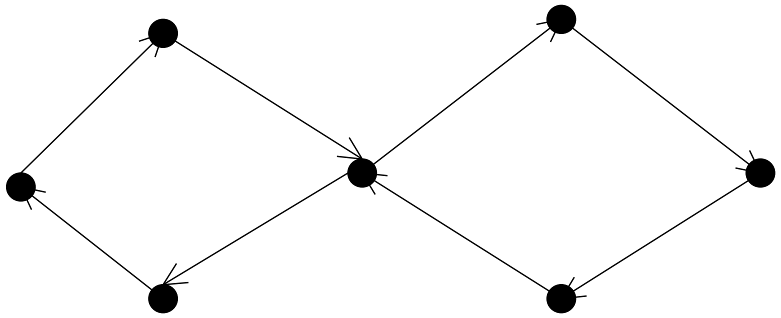
Strongly Connected Components

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.

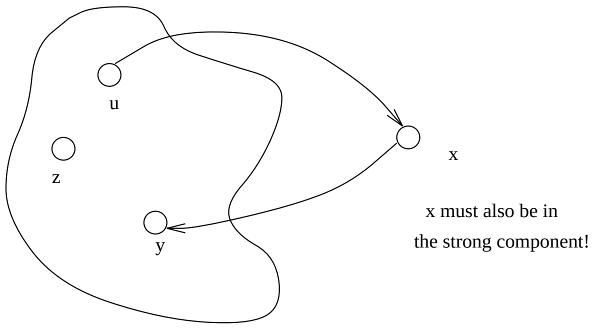
- Compute the transpose graph G^T (reverse all edges in G)

- 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.

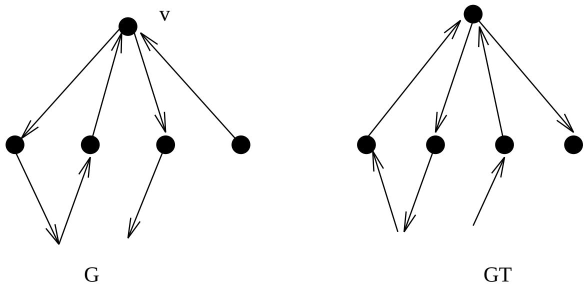


Lemma: In any DFS forest, all vertices in the same strongly connected component are in the same tree.

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.

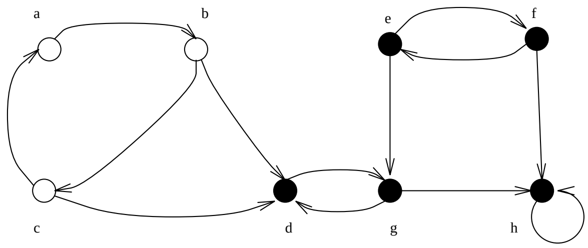
What does DFS(G^T, v) Do?

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



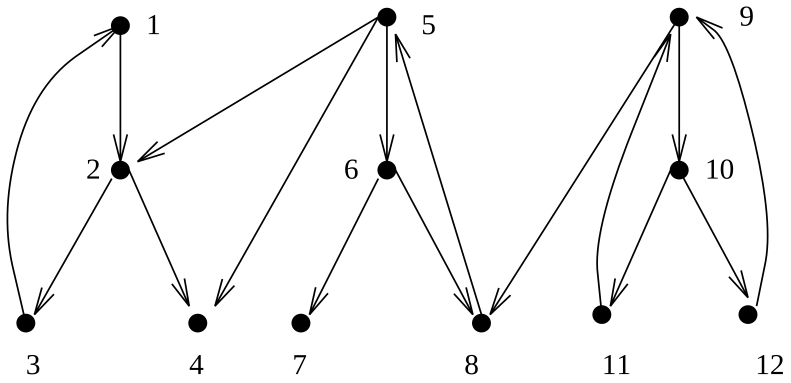
9, 10, 11, 12 can reach 9, oldest remaining finished is 5.

5, 6, 7 can read 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