Strongly Connected Components (Tarjan's Algorithm)

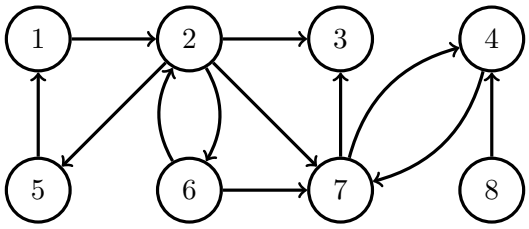
Samuel Hsiang

October 16, 2015

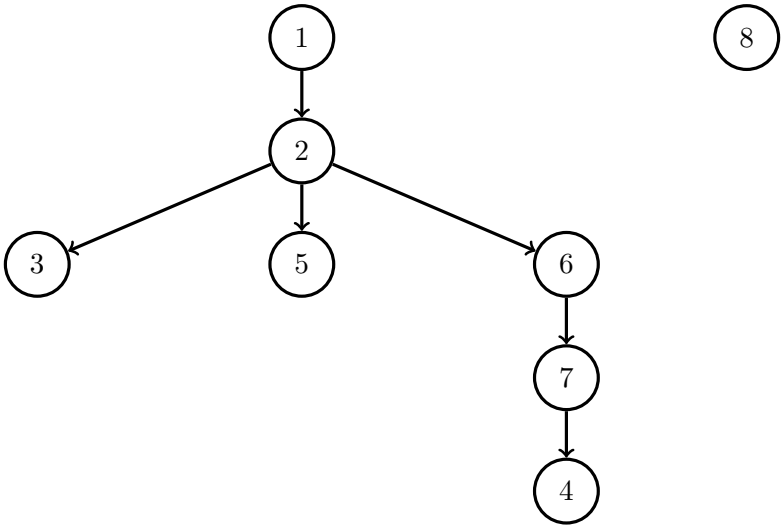
1 Strongly Connected Components

This section covers Tarjan's algorithm detects strongly connected components in a directed graph. We begin with the DFS traversal of the graph, building a forest (collection of trees). In a DFS, we only traverse each vertex once; this means when we encounter an edge to a vertex we already visited, we move on without calling the DFS on that node. Once we obtain the forest, we can split it into subgraphs that represent our strongly connected components.

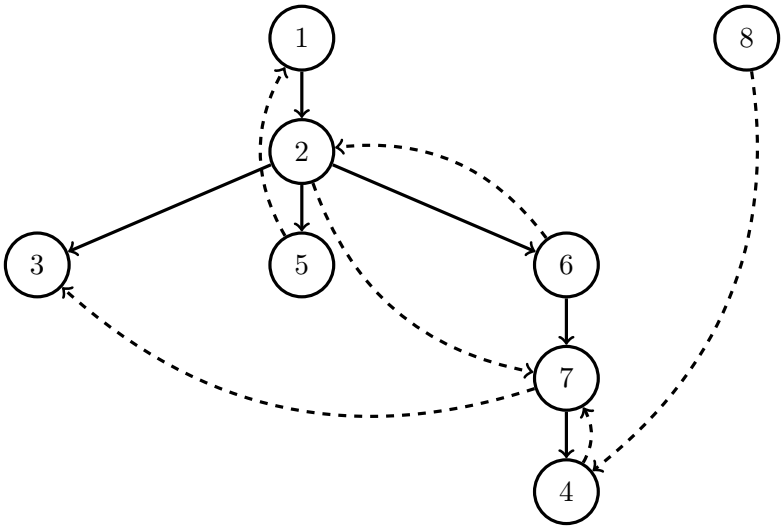
Let's work through how we can get the subgraphs representing the strongly connected components from our DFS. Here's the graph from the beginning of this section.



It doesn't matter from which node we begin our DFS or the order in which we choose children; in our example, we'll simply choose the node with the minimum label to traverse first. This will result in two trees in our forest, one rooted at 1 and the other rooted at 8.



Now this by itself is not very useful, as in order to find strongly connected components, we'll need the edges in the directed graph that aren't included in our tree. Here we add the other edges as dashed pointers.



Note that strongly connected components are represented by subtrees in the graph. We call the root of the subtree called the root of the strongly connected component.

One edge here is immediately useless. We already know that 2 can reach 7; 7 is in 2's subtree. The fact that there is an edge from 2 to 7 doesn't change anything. Then we have a crucial observation – the only possible useful extra edges are those that go up to a previous node in the subtree, like the edge from 5 to 1, or those that go "left" to a previously-visited vertex, either to a previous branch in the tree, like from 7 to 3, or to a previous subtree entirely, like from 8 to 4.

If a node v has an edge to a direct ancestor in the tree, that means we immediately have a cycle, and therefore the node, its ancestor, and every vertex along the way must be in the same strongly connected component.

Naturally, the "left" case is trickier. Suppose v has a left edge to a vertex u. We somehow need a way to find out if u has a path back to v. We know that u cannot go down the tree to v as v is not in the subtree of u by the way DFS constructed the tree. Therefore, we want to know whether u has a path back up to some common ancestor with v. However, again by the way DFS traverses the graph, the entire subtree of u has already been searched before the DFS reaches v. We want to exploit this fact with some kind of memoization.

If vertex v was the nth vertex visited in the DFS, we'll mark v with the label $order(v) = n$. We'll also keep track of the "least" vertex $link(v) = u$ that we know up to that point that v can visit, or the vertex u with the minimum $order(u)$ that v can reach so far.

As we're using a DFS, we'll use a stack S to keep track of nodes we've visited. In a normal DFS on a tree, once we finish exploring a vertex v, we pop off v from the stack. This will not be the case for us. A node remains on the stack iff it has a path to some node earlier in the stack.

This means as we explore the descendants of a vertex v, we'll know if v has a path back to a previous vertex. That is, if $link(v) < order(v)$, it stays on the stack. If $link(v) = order(v)$, we take it off the stack.

Now we describe Tarjan's algorithm. Here, *num* represents a global variable that indicates how many vertices have been visited so far.

Algorithm 1 Tarjan

```
function StrongConnect(vertex u)
    num ← num + 1                                   ▷ increment num
    order(u) ← num                                  ▷ set order(u) to smallest unused number
    link(u) ← order(u)                              ▷ least order(v) accessible is u itself
    push u on S
    for all neighbors v of u do
        if order(v) is undefined then               ▷ v has not been visited
            StrongConnect(v)
            link(u) ← min(link(u), link(v))
        else if v is on stack S then                ▷ v is in current component
            link(u) ← min(link(u), order(v))
    if link(u) = order(u) then                      ▷ u is root of component, create SCC
        create new strongly connected component
        repeat
            v ← top of S
            add v to strongly connected component
            pop top from S
        until u = v
function Tarjan(G(V, E))
    num ← 0
    initialize new empty stack S
    for all vertices v ∈ V do
        if order(v) is undefined then               ▷ v has not been visited
            StrongConnect(v)
```

This algorithm runs in $O(V + E)$.