Adapted From Virginia Williams' lecture notes.

COMP 285 (NC A&T, Spr '22) Lecture 18

Strongly Connected Components

1 Connected components in undirected graphs

A connected component of an undirected graph $G = (V, E)$ is a maximal set of vertices $S \subset V$ such that for each $u \in S$ and $v \in S$, there exists a path in G from vertex u to vertex v.

Definition 1.1 (Formal Definition)**.** Let $u \equiv v$ if and only if G has a path from vertex u to vertex v . This is an equivalence relation (it is symmetric, reflexive, and transitive). Then, a connected component of G is an equivalence class of this relation $\equiv$. Recall that the equivalence class of a vertex u over a relation $\equiv$ is the set of all vertices v such that $u \equiv v$.

1.1 Algorithm to find connected components in a undirected graph

In order to find a connected component of an undirected graph, we can just pick a vertex and start doing a search (BFS or DFS) from that vertex. All the vertices we can reach from that vertex compose a single connected component. To find all the connected components, then, we just need to go through every vertex, finding their connected components one at a time by searching the graph. Note however that we do not need to search from a vertex v if we have already found it to be part of a previous connected component. Hence, if we keep track of what vertices we have already encountered, we will only need to perform one BFS for each connected component.

Proof. When searching from a particular vertex v , we will clearly never reach any nodes outside the connected component with DFS or BFS. So we just need to prove that we will in fact reach all connected vertices. We can prove this by induction: Consider the vertices at minimum distance i from vertex v. Call these vertices "level i" vertices. If BFS or DFS successfully reaches all vertices at level i, then they must reach all vertices at level $i + 1$, since each vertex at distance $i + 1$ from v must be connected to some vertex at distance $i from v$. This is the inductive step, and for the base case, DFS or BFS will clearly reach all vertices at level 0 (just v itself). So indeed this algorithm will find each connected component correctly. □

The searches in the above algorithm take total time $O(|E| + |V|)$, because each BFS or DFS call takes linear time in the number of edges and vertices for its component, and each component is only searched once, so all searches will take time linear in the total number of edges and vertices.

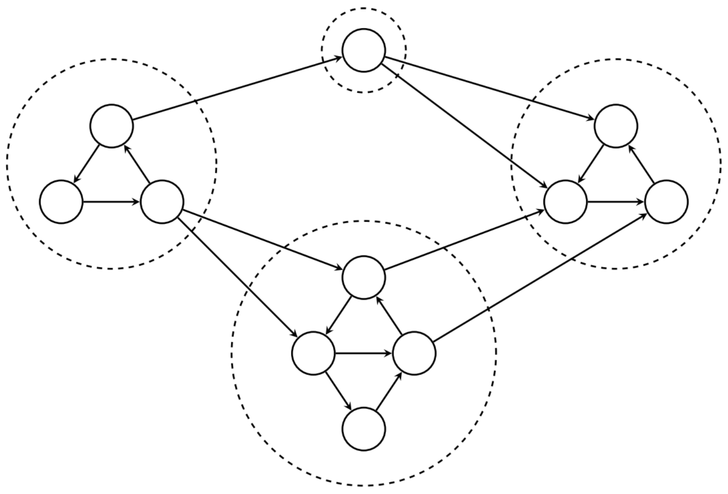
Figure 1: The strongly connected components of a directed graph

2 Connectivity in directed graphs

How can we extend the notion of connected components to directed graphs?

Definition 2.1 (Strongly connected component (SCC))**.** A strongly connected component in a directed graph $G = (V, E)$ is a maximal set of vertices $S \subset V$ such that each vertex $v \in S$ has a path to each other vertex $u \in S$. This is the same as the definition using equivalence classes for undirected graphs, except now $u \equiv v$ if and only if there is a path from u to v AND a path from v to u.

Definition 2.2 (Weakly connected compnent)**.** Let $G = (V, E)$ be a directed graph, and let G' be the undirected graph that is formed by replacing each directed edge of G with an undirected edge. Then the weakly connected components of G are exactly the connected components of G'.

3 Algorithm to find strongly connected components of a directed graph

The algorithm we present is essentially two passes of depth-first search, plus some extremely clever additional book-keeping. The algorithm is described in a top-down fashion in Algorithms 1 to 3. Algorithm 1 describes the top level of the algorithm, and Algorithm 2 and Algorithm 3 describe the subroutines DFS-Loop and DFS. Read these procedures carefully before proceeding to the next section.

Remark 1. *The algorithm in Algorithm 1 is a bit different than the one in CLRS/Lecture! The difference is that in these notes, we first run DFS on the reversed graph, and then we run it again on the original; in CLRS, we first run DFS on the original, and then the second*

Algorithm 1: The top level of our SCC algorithm. The f-values and leaders are computed in the first and second calls to DFS-Loop, respectively (see below)

```
INPUT: A directed graph $G = (V, E)$, in adjacency list representation. Assume that the vertices $V$ are labeled $1, 2, 3, \cdots, n$.
$G^{\text{rev}} \leftarrow$ the graph $G$ after the orientation of all arcs have been reversed.
Run the DFS-Loop subroutine on $G^{\text{rev}}$, processing vertices in any arbitrary order, to obtain a finishing time $f(v)$ for each vertex $v \in V$.
Run the DFS-Loop subrouting on $G$, processing vertices in creasing order of $f(v)$, to assign a "leader" to each vertex $v \in V$. The leader of a vertex $v$ will be the source vertex that the DFS that discovered $v$ started from.
The strongly connected components of $G$ correspond to vertices of $G$ that share a common leader.
```

Algorithm 2: The DFS-Loop subroutine

```
INPUT: A directed graph $G = (V, E)$, in adjacency list representation.
Let global variable $t \leftarrow 0$ /* This keeps track of the number of vertices that have been fully explored */
Let global variable $s \leftarrow$NULL /* This keeps track of the vertex from which the last DFS call was invoked */
for $i = n, n-1, \cdots, 1$ do
  // In the first call, vertices are labeled $1, 2, \cdots, n$ arbitrarily. In the second call, vertices are labeled by their $f(v)$-values from the first call.
  if $i$ not yet explored then
    Let $s \leftarrow i$ /* Set the current source $s$ to $i$ All vertices discovered from the below DFS call will have their leaders set to $s$ */
    DFS(G, i)
  end if
end for
```

Algorithm 3: The DFS subroutine. The f-values only need to be computed during the first call to DFS-Loop, and the ledaer values only need to be computed during the second call to DFS-Loop.

```
INPUT: A directed graph $G = (V, E)$, in adjacency list representation, and source vertex $i \in V$.
Mark $i$ as explored. /* It remains explored for the entire duration of the DFS-Loop call */
leader($i$) $\leftarrow s$
for arc($i, j$) in $G$ do
  if $j$ noto yet explored then
    DFS($G, j$)
  end if
end for
$t \leftarrow t + 1$
Let $f(i) \leftarrow t$
```

time on the reversed graph. Is it the case that one of these two textbooks has messed it up? In fact, it doesn't matter: the SCCs ofG are the same as the SCCs of G^{rev} *, so both algorithms find exactly the same SCC decomposition.*

As we've seen, each invocation of DFS-Loop can be implemented in linear time (i.e., $O(|E| + |V|)$), so this whole algorithm will take linear time (the bookkeeping of leaders and finishing times just adds a constant number of operations per each node).

4 An Example

But why on earth should this algorithm work? An example should increase its plausibility (though it certainly doesn't constitute a proof of correctness). Figure 2 displays a reversed graph G^{rev}, with its vertices numbered arbitrarily, and the f-values computed in the first call to DFS-Loop. In more detail, the first DFS is initiated at node 9. The search must proceed next to node 6. DFS then has to make a choice between two different adjacent nodes; we have shown the f-values that ensue when DFS visits node 3 before node 8.[1]. When DFS visits node 3 it gets stuck; at this point node 3 is assigned a finishing time of 1. DFS backtracks to node 6, proceeds to node 8, then node 2, and then node 5. DFS then backtracks all the way back to node 9, resulting in nodes $5, 2, 8, 6$, and 9 receiving the finishing times $2, 3, 4, 5$, and 6, respectively. Execution returns to DFS-Loop, and the next (and final) call to DFS begins at node 7.

Figure 3 shows the original graph (with all arcs now unreversed), with nodes labeled withtheir finishing times. The magic of the algorithm is now evident, as the SCCs of G present

[1]Different choices of which node to visit next generate different sets of f-values, but our proof of correctness will apply to all ways of resolving these choices

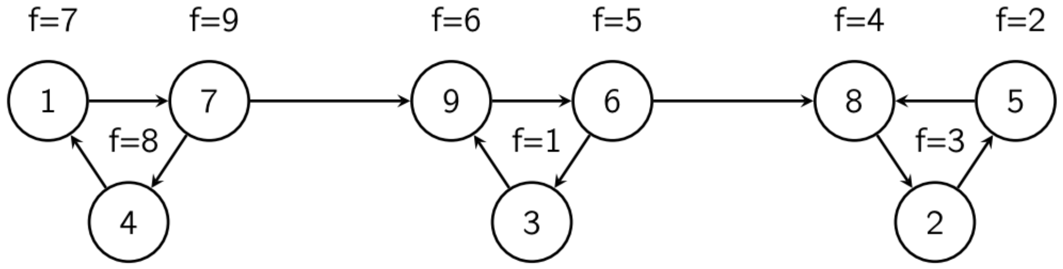

Figure 2: Example exuection of the strongly connected components algorithm. Nodes tare labaled arbitrarily and their finishing times are shown.

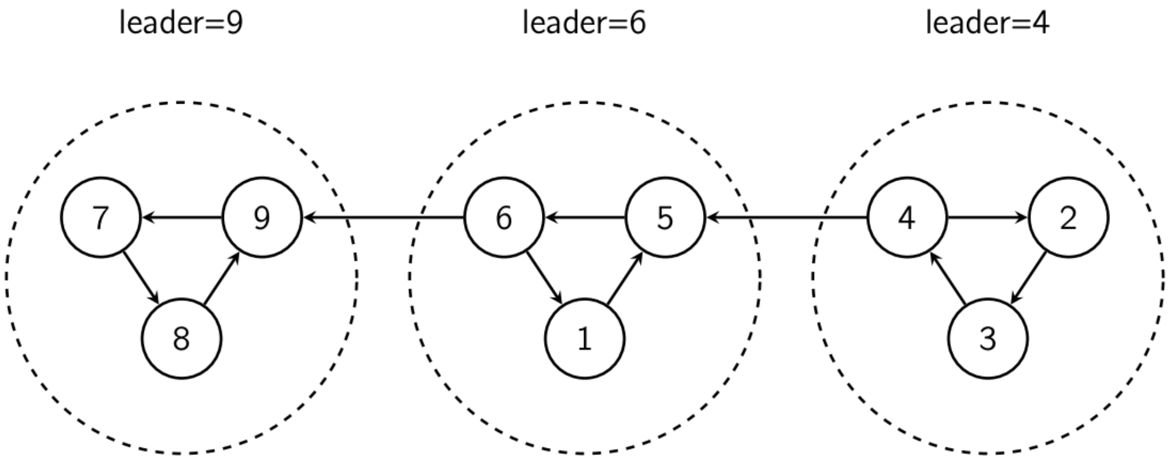

Figure 3: Example execution of the strongly connected components algorithm. Nodes are labeled by their finishing times and their leaders are shown.

themselves to us in order: since we call DFS on the nodes in decreasing order of their finishing times, the first call to DFS discovers the nodes 7-9 (with leader 9); the second the nodes 1,5, and 6 (with leader 6); and the third the remaining three nodes (with leader 4).