15-750: Graduate Algorithms February 8, 2016

Lecture 10: Strongly Connected Components, Biconnected Graphs

Lecturer: David Witmer *Scribe: Zhong Zhou*

1 DFS Continued

We have introduced Depth-First Search (DFS) earlier in the previous lecture. There are three ways of traversing the graph:

1. Depth-First Search (DFS)
2. Breadth-First Search (BFS)
3. Random Walk

DFS, Strongly Connected Components, and Biconnected Components are our focus for this lecture. Before we proceed to Strongly Connected Components, we need to prove a important theorem about Depth-First Search (DFS).

For any graph $G = (V, E)$, DFS, the strategy of exhaustively searching every path to the end before exploring the next one, takes time $O(|V| + |E|)$. The edges traversed by DFS form a forest.

Input: $G = (V, E)$
Initialize $\forall u \in V, u.c =$ white;
Initialize $t = 0$;
for $u \in V$ **do**
  if $u.c = white$ **then**
    DFS-visit(u, G);
  end
end

Algorithm 1: DFS

The subroutine of DFS is defined as DFS-visit:

Input: u, G
$u.c =$ grey;
$t = t + 1$;
$disc(u) = t$;
for $v \in adj(u)$ **do**
  if $v.c = white$ **then**
    DFS-visit(v, G);
  end
end
$u.c =$ black;
$t = t + 1$;
$finish(u) = t$;
return F;

Algorithm 2: DFS-visit

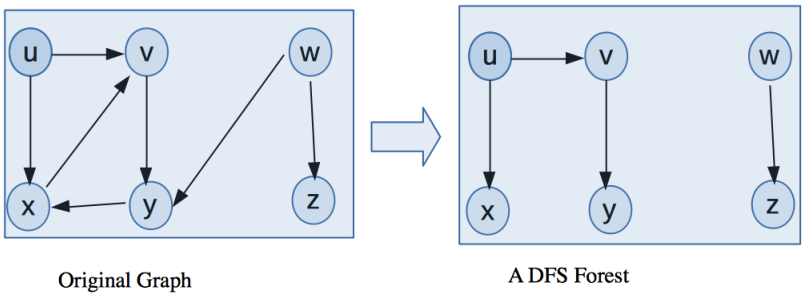

Figure 1: A DFS Forest Formed by DFS Algorithm (Adapted from an example in [CLRS09] Section 22.3)

Theorem 1.1. *In a DFS-forest, v is a descendant of u if and only if $\exists$ a path of white vertices from u to v at time $disc(u)$.*

Proof. ($\Rightarrow$) $\forall v$ which is a descendant of u, we have

$$disc(u) < disc(v)$$

Therefore, v must be white at time $disc(u)$.
Since we have proven this for all v which is any descendant of u, all descendants of u must be white at time $disc(u)$.
All vertices in DFS tree path from u to v are white at $disc(u)$.

($\Leftarrow$) Suppose $\exists$ a white path from u to v at $disc(u)$.

From our assumption we have $disc(u) < disc(v)$.

Assume v is the first vertex on path that is not descendant of u, let w be its predecessor. Then we have $finish(w) \leq finish(u)$. Then

$$disc(u) < disc(v) < finish(w) \leq finish(u)$$

From the nesting theorem we have proven earlier, we finish v before we finish u, therefore, v is a descendant of u.

Therefore, we have proven both directions of the statement. □

2 Strongly Connected Components

In this section, we will see that any graph $G = (V, E)$ can be partitioned into strongly connected components. For a component to be strongly connected, every vertex in the component must be reachable from every other vertex in the same component. We will define an equivalence relation on the graph G and strongly connected components are induced by this equivalence relation. The formal definitions are as follows.

Definition 2.1 (Equivalence of nodes)**.** For a directed graph, $G = (V, E)$, $\forall u, v \in V$, $u \equiv v$ if $\exists$a path from u to v and a path from v to u.

In mathematics, an equivalence relation satisfies reflexivity, symmetry and transitivity. As a consequence of these properties, an equivalence relation provides a partition of a set into equivalence classes.

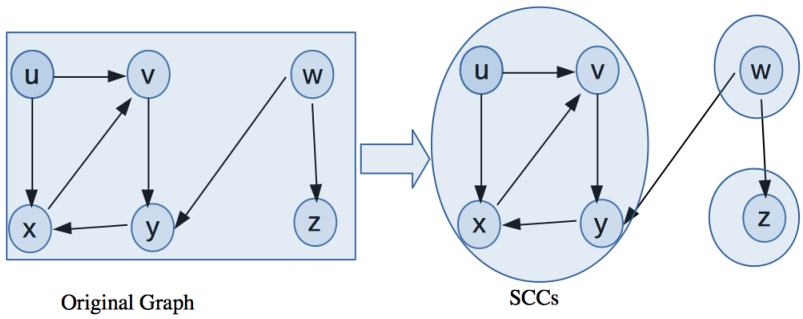

Figure 2: Strongly Connected Components (Adapted from an example in [CLRS09] Section 22.3)

Definition 2.2 (Equivalence Relation)**.** A given binary relation R on a set X is said to be an equivalence relation if and only if it is reflexive, symmetric and transitive. That is, for all a, b and c in X:

1. aRa. (Reflexivity)
2. aRb if and only if bRa. (Symmetry)
3. If aRb and bRc then aRc. (Transitivity)

Our claim is that the relation $\equiv$ that we defined on nodes of a directed graph is also an equivalence relation. The proof is straightforward.

Definition 2.3 (Strongly Connected Components of G)**.** Strongly connected components of G are equivalent classes induced by $\equiv$. Each strongly connected component is a maximal set S such that $\forall u, v \in S, u \equiv v$.

Definition 2.4 (Component Graph)**.** If G has set of strongly connected components, $C_1, C_2, ..., C_k$, the component graph $G^{scc} = (V^{scc}, E^{scc})$ has $V^{scc} = 1, 2, ..., k$ and $E^{scc} = \{(i, j) \mid G \text{ has an edge } C_i \rightarrow C_j\}$.

Claim 2.5. *Every component graph is a DAG.*

Proof. We will prove by contradiction. Suppose $\exists$ strongly connected components $C_1, C_2, ..., C_k$ such that the component graph contains a cycle.

Then all components in this cycle can form a bigger strongly connected component by definition. This violates our assumption that strongly connected components are maximal. Therefore, every component graph is a DAG. □

Therefore, G^{scc} has a topological ordering.

Kosaraju's Algorithm uses DFS to find strongly connected components. If a DFS returned a forest such that the trees of the forest were exactly the strongly connected components, we would be all set: we could just run DFS and output the vertices of each DFS tree as a strongly connected component. We then want to order the vertices of our graph so that the trees of the resulting DFS forest are exactly the strongly connected components. That is, we don't want our DFS to traverse edges between strongly connected components. These edges, corresponding to edges of G^{scc}, should be cross edges in our DFS forest. In order for an edge (u, v) to be a DFS cross edge, we must discover the head of the edge v before we discover its tail u so that we don't traverse it in our search. We therefore want a search order corresponding to a reverse topological sort of G^{scc}.

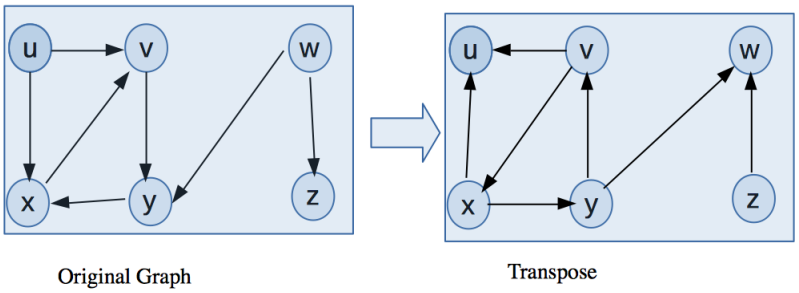

Figure 3: Transpose of Graph

Recall from the previous lecture, we have shown that in a DAG, reverse finish time gives a topological sort. Since we need a reverse topological sort, it will be convenient to consider the transpose graph, in which every edge is reversed.

Definition 2.6 (Transpose of a Graph)**.** Given $G = (V, E)$, the transpose of G is $G^T = (V, E^T)$ with $E^T = \{(u, v) \mid (v, u) \in E\}$

We can now formally state Kosaraju's algorithm (1978) for finding strong components in a graph:

Input: $G = (V, E)$
1. Perform a DFS of G and number the vertices in order of completion of the recursive calls, compute finish(u) $\forall u$;
2. Construct a new directed graph G^T by reversing the direction of every edge in G;
3. Perform a DFS on G^T starting the search from the highest numbered vertex according to the decreasing order of finish time;
4. return DFS trees;

Algorithm 3: GCC with DFS

Now we need to formally prove that this algorithm is correct. We will only consider discovery times and finish times for the first DFS on G; $disc$ and $finish$ will refer to these times. Let U be a subset of V, and define the following:

$$disc(U) = \min_{x \in U} disc(x), \quad finish(U) = \max_{x \in U} finish(u)$$

Lemma 2.7. *Let C and C' be distinct SCCs. If $\exists$ an edge (u, v) in E where u is in C and v is in C', then $finish(C) > finish(C')$*

Proof.
Case 1: $disc(C) < disc(C')$
Let x be the first vertex discovered in C. We know that at time $disc(x)$, all vertices in C and C' are white.
For any vertex $w \in C'$, $\exists$ a white vertex path from x to w. Therefore, all vertices in C and C' are descendants of x by Theorem 1.1. Thus we have

$$finish(x) = finish(C) > finish(C')$$

Case 2: $disc(C) > disc(C')$
Let y be the first vertex discovered in C'. All vertices of C' are descendants of y in our DFS tree.

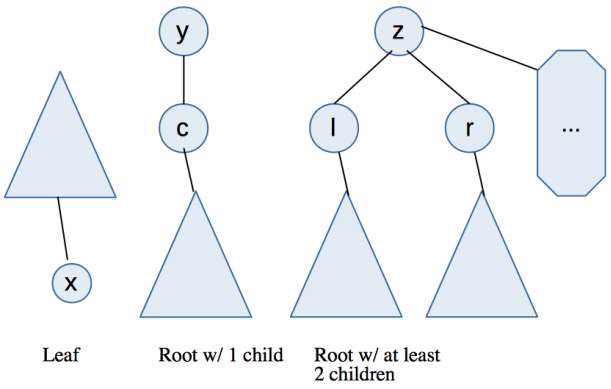

Figure 4: Examples of nodes in undirected graph

Therefore, $finish(y) = finish(C')$.
Since G^{scc} is a DAG, there does not exists a path from C' to C, which means all vertices in C are still white when we finish. Thus, $finish(C) > finish(C')$.

Therefore, in both cases, we have proven $finish(C) > finish(C')$. □

This lemma will be reversed if we look at the transpose:

Corollary 2.8. *Let C and C' be distinct SCCs.*
If $\exists$ an edge (u, v) in E^T where u is in C and v is in C', then $finish(C) < finish(C')$.

Theorem 2.9. *Kosaraju's algorithm is correct.*

Proof. We will do induction on number of DFS tree we have found so far. The base case is trivial.

In the induction step, assume the first k trees found are SCCs. We assume the $(k+1)^{st}$ tree is T, u is the first vertex of T discovered by our DFS, and u is in SCC C. We would like to show that vertices of T are exactly the same as vertices in C; this requires proofs in both directions.

($\Leftarrow$) All vertices in C are in T. By our inductive hypothesis, we have that all vertices of C are still white at $disc(u)$. Theorem 1.1 gives that all vertices of T are descendants of u. Therefore, all vertices in C are in T.

($\Rightarrow$) All vertices in T are in C. The DFS order ensures that $finish(u) = finish(C) > finish(C')$ $\forall$ SCCs C' that have not been reached. By Corollary 2.8, there are no edges from C to C', so no vertices in C' can be in T. Therefore, all vertices in T are in C.

The vertices of T are exactly the vertices of C, and we have proven the induction for $k+1$. Thus, Kosaraju's algorithm is correct. □

3 Biconnected Components

Before we introduce biconnected components, we need to first introduce the notion of articulation points.

Definition 3.1 (Articulation Point)**.** For a undirected graph, v is an articulation point if $\exists$ distinct x, y, such that all x, y path go through v.

Definition 3.2 (Biconnected Graph)**.** G is biconnected if there is no articulation point.

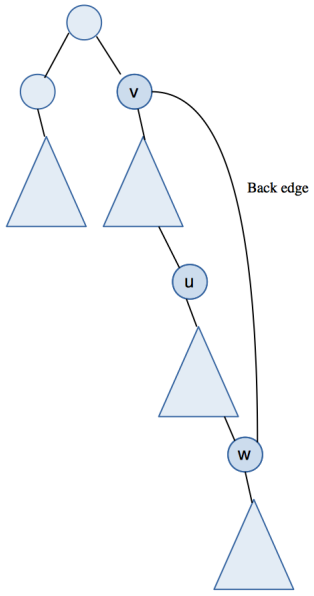

Figure 5: Examples of non-leaf, non-root node in undirected graph

An undirected graph will have only tree edges or back edges. Can a leaf be an articulation point? The answer is no because we cannot disconnect a graph by removing leaves. Take a look at Figure 4; the leaf x cannot be an articulation point. If we have a root that only has one child just like the node y at Figure 4, it cannot be an articulation point either becasue we cannot disconnect the graph by removing this node. Now, what kind of root node can be an articulation point? When a root node has at least two children like node z at Figure 4, its removal will result in a disconnected graph; therefore, a root node with at least two children is an articulation point. And now, consider a non-leaf, non-root node. Such a node could be an articulation point if there is not any back edge from a descendent to an ancestor in its DFS tree. Take a look at Figure 5; the node u is not an articulation point because there is an edge from its descendent w to its ancestor v.

Definition 3.3 (Low of a vertex)**.** $\forall v$ in a graph, $low(v)$ is the vertex with the earliest discovery time that is reachable from v by using a directed path that uses just one back edge, i.e.,

$$low(v) = \min\{\{disc(v)\} \cup \{disc(w) \mid \exists u \text{ such that } u \text{ is descendant of } v \text{ and } (u, w) \text{ is a back edge}\}\}.$$

Theorem 3.4. *When using DFS to find articulation points, we have 3 cases:*

1. *a leaf is never an articulation point*
2. *a roots is an articulation point if and only if it has at least two children*
3. *a non-leaf, non-root vertex v is articulation point if and only if $\exists$ child u of v such that $low(u) \geq disc(v)$*

References

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009. 1, 2