TA Applications are Out!

If you want to apply to be a intro CIS TA, that is for 110, 120, 160, or 121, the applications are out. If you are even slightly considering it then you should apply! http://tinyurl.com/cis-spring-19

Timestamped DFS

Recall the DFS graph traversal algorithm from last week’s recitation. This week, we will extend this algorithm to include timestamps for every vertex. Each vertex $v$ now has two timestamps. The first timestamp, $v.d$, records when $v$ is first discovered in the search. The second timestamp, $v.f$, records when $v$ is finished, that is, when it’s adjacency list has been examined completely. Below is the pseudocode for a timestamped algorithm built upon the recursive version of DFS. Note that we can similarly modify the iterative version of DFS to include timestamps.

```
DFS(G)
1  time = 0
2  for each vertex $v \in G.V$
3      $v.visited = false$
4  for each vertex $v \in G.V$
5      if $v.visited == false$
6        DFS-VISIT($v$)

DFS-VISIT($v$)
1  time = time + 1
2  $v.d = time$
3  $v.visited = true$
4  for each vertex $w \in G.Adj(v)$
5      if $w.visited == false$
6        DFS-VISIT($w$)
7  time = time + 1
8  $v.f = time$
```

Figure 1: An example graph for the timestamped DFS algorithm. Note that the source node, $s$, has a start time of 1. Discovery and finish times are indicated inside each node. The edges traversed by DFS have been highlighted.
Strongly Connected Components

Definition 1 (Transpose graph). For a graph $G = (V, E)$, the transpose of $G$ is defined to be $G^T = (V, E^T)$, where $E^T = \{(u, v) : (v, u) \in E\}$. That is, $E^T$ consists of the edges of $G$ with their directions reversed. Given an adjacency-list representation of $G$, the time to create $G^T$ is $O(V + E)$.

Definition 2 (Strongly connected component). Given a directed graph $G = (V, E)$, a strongly connected component (SCC) is a maximal set $S \subseteq V$ such that for all $u, v \in S$, there exists a path $u \rightsquigarrow v$ and a path $v \rightsquigarrow u$.

Note: We consider only directed graphs here because in undirected graphs, every connected component is trivially strongly connected.

Definition 3 (Component graph (kernel graph)). The strongly connected component graph of a directed graph $G$ is the directed graph $G^{SCC} = (V^{SCC}, E^{SCC})$ where each vertex of $V^{SCC}$ represents a strongly connected component of $G$, and the new edges $E^{SCC}$ consist of the directed edges between the SCCs of $G$.

Formally, the definition of the component graph is as follows: suppose that $G$ has strongly connected components, $C_1, C_2, ..., C_k$. The vertex set $V^{SCC}$ is $\{v_1, v_2, ..., v_k\}$, and it contains a vertex $v_i$ for each strongly connected component $C_i$ of $G$. There is an edge $(v_i, v_j) \in E^{SCC}$ if $G$ contains a directed edge $(x, y)$ for some $x \in C_i$ and some $y \in C_j$. Looked at another way, by contracting all edges whose incident vertices are within the same strongly connected component of $G$, the resulting graph is $G^{SCC}$. Figure 2(c) shows the component graph of the graph in Figure 2(a). In other words, we can contract every edge whose incident vertices are in the same SCC to produce the component graph.

Notice that the component graph is a directed, acyclic graph. This is useful because we can now we can topologically order its vertices. This idea is crucial to many linear time graph algorithms.

Kosaraju’s algorithm

Description

Kosaraju’s is a linear time algorithm for finding the strongly connected components of a graph. The algorithm is described below:

KOSARAJU’S(G)
1 call DFS(G) to compute the finishing times $u.f$ for each vertex $u$
2 compute $G^T$
3 call DFS($G^T$), but in the main loop of DFS consider the vertices in order of decreasing $u.f$ as computed in line 1
4 output the vertices of each tree in the depth-first forest formed in line 4 as a separate strongly connected component

Notes: We call DFS on an arbitrary initial node in line 1; which node we pick does not affect the correctness of the algorithm. In line 3, we would initially call DFS on the node with the highest finish time, and whenever we must restart our call to DFS we start on the unvisited node which has the highest finish time out of all unvisited nodes.

Running time

DFS takes $O(n + m)$ time. We perform it twice, for a total of $O(2(n + m)) = O(n + m)$. Computing $G^T$ requires simply iterating over $G$’s adjacency list once, which takes $O(n + m)$ time. Thus, the total running time is $O(n + m)$. 

2
Figure 2: (a) represents the run of DFS($G$, $c$) with the start and end times labeled. The SCCs are shaded. The edges traversed in the DFS call are highlighted in grey. (b) is the transpose $G^T$, with the vertices of highest end time in (a) shaded in dark grey. The SCCs are also shaded. The edges traversed in the DFS call are highlighted in grey. (c) represents $G^{SCC}$, which is the strongly connected component graph of $G$.

Plan for proving Correctness

Here is the intuitive look as to why Kosaraju’s is actually giving us the proper SCCs. Let’s examine what the $G^{SCC}$ graph looks like. We know it is some DAG, which means we can topologically sort that DAG. Let the nodes of $G^{SCC}$ in topologically sorted order be $C_1, C_2, ... C_k$. We are certain that $C_k$ has no outgoing edges, because it is last in our topological sort. We know by the definition of a strongly connected component all pairs of nodes $(u, v)$ in $C_k$ have paths between them in both directions. Therefore if we perform a DFS starting at any node in $C_k$, we will discover all nodes in $C_k$, and because we don’t have any outgoing edges, we will only discover the nodes in $C_k$.

Now we perform a DFS on node $C_{k-1}$. The only outgoing edges this component could have would be to $C_k$ by the properties of topological sort. So by the same logic as before, when we perform DFS on any node in $C_{k-1}$, our resulting DFS tree will just contain the nodes within $C_{k-1}$. We can see now that these repeated calls to DFS in reverse topologically sorted order will give us the SCCs of $G$. This principle is why Kosaraju’s functions as it does.

Now we look at how to actually prove this argument. Using Lemma 1, we show $G^{SCC}$ actually is a DAG, which is necessary for us to topologically sort it. Using Lemma 2, we prove that components which have a
higher max finish time are earlier in the topological sort of $G^{SCC}$. This is important, as we know we want to iterate over our vertices in reverse topologically sorted order, so we need some way to know what this order is. In Corollary 2, we apply Lemma 2 to show that components which have a higher max finish time are later in the topological sort of $(G^T)^{SCC}$. That is, iterating in decreasing finish time accomplishes our goal of iterating in reverse topologically sorted order. Because we know that the SCCs of $G^T$ and $G$ are the same, returning the SCCs of $G^T$ will give us the correct SCCs. Finally, using Theorem 1, we formally prove that the union of all of our steps is correct: that is, iterating with DFS in decreasing finish time actually returns us the correct SCCs.

**Proof of Correctness**

Our algorithm for finding strongly connected components of a graph $G = (V, E)$ uses the transpose of $G$, defined as $G^T$. It is interesting to observe that $G$ and $G^T$ have exactly the same strongly connected components: $u$ and $v$ are reachable from each other in $G$ if and only if they are reachable from each other in $G^T$. The figure above in (b) shows the transpose of the graph in (a), with the strongly connected components labeled.

Here is some simple intuition for that fact. If $u$ and $v$ are in the same SCC in $G$, then there exists a path, call it $p_u$, such that $u \sim v$ and a path, call it $p_v$, such that $v \sim u$. In $G^T$, all edges have their direction reversed. This means that $p_u$ now connects $v$ to $u$, and $p_v$ connects $u$ to $v$. We can see then that there still exists a path connecting $u$ to $v$ and $v$ to $u$. Therefore any two nodes which were in the same SCC in $G$ will be in the same SCC in $G^T$.

The idea behind this algorithm comes from a key property of the component graph: the component graph is a DAG, which the following lemma implies:

**Lemma 1**

Let $C$ and $C'$ be distinct strongly connected components in directed graph $G = (V, E)$, let $u, v \in C$, let $u', v' \in C'$, and suppose that $G$ contains a path $u \sim u'$. Then $G$ cannot also contain a path $v' \sim v$.

**Proof:** If $G$ contains a path $v' \sim v$, then it contains paths $u \sim u' \sim v'$ and $v' \sim v \sim u$. Thus, $u$ and $v'$ are reachable from each other, thereby contradicting the assumption that $C$ and $C'$ are distinct strongly connected components.

We shall see that by considering vertices during the second depth-first search in decreasing order of the finishing times that were computed in the first depth-first search, we are, in essence, visiting the vertices of the component graph (each of which corresponds to a strongly connected component of $G$) in topologically sorted order. Because we are computing two depth-first searches in our algorithm, there is some potential for ambiguity when discussing $u.d$ and $u.f$. Whenever we say $u.d$ or $u.f$, we mean the discovery and finish times of $u$ in the first call of DFS in line 1.

We extend the notation for discovery and finishing times to sets of vertices. If $U \subseteq V$, then we defined, $d(U) = \min_{u \in U} \{u.d\}$ and $f(U) = \max_{u \in U} \{u.f\}$. That is, $d(U)$ and $f(U)$ are the earliest discovery time and latest finishing time, respectively, of any vertex in $U$. The following lemma and its corollary give a key property relating strongly connected components and finishing times in the first depth-first search.

**Lemma 2**

Let $C$ and $C'$ be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$. 


Proof: We consider two cases, depending on which strongly connected component, \( C \) or \( C' \), had the first discovered vertex during the depth-first search. If \( d(C) < d(C') \), let \( x \) be the first vertex discovered in \( C \). At time \( x.d \), all vertices in \( C \) and \( C' \) have not been visited. At that time, \( G \) contains a path consisting only of unvisited vertices from \( x \) to each vertex in \( C \). Because \((u, v) \in E\), for any vertex \( w \in C' \), there is also a path in \( G \) at time \( x.d \) from \( x \) to \( w \) consisting only of unvisited vertices: \( x \rightarrow u \rightarrow v \rightarrow w \). Intuitively, we can see how all vertices in \( C \) and \( C' \) become descendants of \( x \) in the depth-first tree—because \( x \) has a path made up of unvisited vertices to every vertex in \( C \) and \( C' \), it follows that DFS will visit each of those vertices before \( x \) is finished. This implies that \( x.f \) will be greater than the finishing times of all other vertices in \( C \) and \( C' \). Therefore, we know that \( f(C) > f(C') \).

If instead we have \( d(C) > d(C') \), let \( y \) be the first vertex discovered in \( C' \). At time \( y.d \), all vertices in \( C' \) have not been visited. At that time, \( G \) contains a path consisting only of unvisited vertices from \( y \) to each vertex in \( C' \). Since there is an edge \((u, v) \in E \) from \( C \) to \( C' \), Lemma 1 implies that there cannot be a path from \( C' \) to \( C \). Hence, no vertex in \( C \) is reachable from \( y \). At time \( y.f \), therefore, all vertices in \( C \) are still unvisited. Thus, for any vertex \( w \in C \), we have \( f(w) > f(y) \), which implies that \( f(C) > f(C') \).

The following corollary tells us that each edge in \( G_T \) that goes between different strongly connected components goes from a component with an earlier finishing time (in the first depth-first search) to a component with a later finishing time.

Corollary 2

Let \( C \) and \( C' \) be distinct strongly connected components in directed graph \( G = (V, E) \). Suppose that there is an edge \((u, v) \in E_T\), where \( u \in C \) and \( v \in C' \). Then \( f(C) < f(C') \).

Proof: Since \((u, v) \in E_T\), we have \((v, u) \in E\). Because the strongly connected components of \( G \) and \( G_T \) are the same, Lemma 2 implies that \( f(C) < f(C') \).

Corollary 2 provides the key to understanding why the strongly connected components algorithm works. Let us examine what happens when we perform the second depth-first search, which is on \( G_T \). We start with the strongly connected component \( C \) whose finishing time \( f(C) \) is maximum. The search starts from some vertex \( x \in C \), and it visits all vertices in \( C \). By Corollary 2, \( G_T \) contains no edges from \( C \) to any other strongly connected component, and so the search from \( x \) will not visit vertices in any other component. Thus, the tree rooted at \( x \) contains exactly the vertices of \( C \). Having completed visiting all vertices in \( C \), the search in line 3 selects as a root a vertex from some other strongly connected component \( C' \) whose finishing time \( f(C') \) is maximum over all components other than \( C \). Again, the search will visit all vertices in \( C' \), but by Corollary 2, the only edges in \( G_T \) from \( C' \) to any other component must be to \( C \), which we have already visited. In general, when the depth-first search of \( G_T \) in line 3 visits any strongly connected component, any edges out of that component must be to components that the search already visited. Each depth-first tree, therefore, will be exactly one strongly connected component. The following theorem formalizes this argument.

Theorem 1

The KOSARAJU’S procedure correctly computes the strongly connected components of the directed graph \( G \) provided as its input.

Proof: We argue by induction on the number of depth-first trees found in the depth-first search of \( G_T \) in line 3 that the vertices of each tree form a strongly connected component. The inductive hypothesis is that the first \( k \) trees produced in line 3 are strongly connected components. The basis for the induction, when \( k = 0 \), is trivial. In the inductive step, we assume that each of the first \( k \) depth-first trees produced in line 3 is a strongly connected component, and we consider the \((k + 1)\)st tree produced. Let the root of this tree be vertex \( u \), and let \( u \) be in strongly connected component \( C \). Because of how we choose roots in the depth-first search in line 3, \( u.f = f(C) > f(C') \) for any strongly connected component \( C' \) other than \( C \) that has yet to be visited. By the inductive hypothesis, at the time that the search visits \( u \), all other vertices of \( C \) have not been visited. By the intuition explained in Lemma 2, this implies that all other vertices of \( C \) are
descendants of \( u \) in its depth-first tree. Moreover, by the inductive hypothesis and by Corollary 2, any edges in \( G^T \) that leave \( C \) must be to strongly connected components that have already been visited. Thus, no vertex in any strongly connected component other than \( C \) will be a descendant of \( u \) during the depth-first search of \( G^T \). Thus, the vertices of the depth-first tree in \( G^T \) that is rooted at \( u \) form exactly one strongly connected component, which completes the inductive step and the proof.

Here is another way to look at how the second depth-first search operates. Consider the component graph \((G^T)_{SCC}\) of \( G^T \). If we map each strongly connected component visited in the second depth-first search to a vertex of \((G^T)_{SCC}\), the second depth-first search visits vertices of \((G^T)_{SCC}\) in the reverse of a topologically sorted order. If we reverse the edges of \((G^T)_{SCC}\), we get the graph \(((G^T)_{SCC})^T\). Because \(((G^T)_{SCC})^T = G_{SCC}\) as explained above, the second depth-first search visits the vertices of \( G_{SCC} \) in topologically sorted order.

### Problems

**Problem 1**

Conceptual questions:

1. (True/False) The finish times of all vertices in a SCC \( s \) must be greater than the finish times of other SCCs reachable from \( s \) during the first DFS.

   **Solution.** False, consider the first vertex the DFS visits in \( s \). Consider a path from that vertex within \( s \) that only has edges to other vertices in that SCC. If DFS takes this path before taking an edge out of \( s \), the vertices on the path will finish first. Since the SCC graph is a DAG, we will never revisit \( s \) if we take an edge out. It is true though that at least one vertex must have a larger finish time than those SCCs reachable from \( s \).

2. How does the number of SCCs of a graph change if a new edge is added?

   **Solution.** Consider a new directed edge \((u, v)\). We have two cases. Either \( u \) and \( v \) are in the same component, in which case the total number of components does not change and we are done; or \( u \) and \( v \) are in different components. Let \( u \) and \( v \) be in components \( C_u \) and \( C_v \) respectively. Consider the component graph. If \( C_u \sim C_v \), then \( (u, v) \) does not change the total number of components, since it is redundant. But if instead \( C_v \sim C_u \), then via \( (u, v) \) we have \( C_u \sim C_v \). Thus, all components reachable with a path starting at \( C_u \) and ending at \( C_v \) (including \( C_u \) and \( C_v \)) are contracted into a single component.

3. (CLRS 22.5) Professor Bacon claims that Kosaraju’s algorithm would be simpler if it used the original (instead of the transpose) graph in the second depth-first search and scanned the vertices in order of increasing finishing times. Does this simpler algorithm always produce correct results?

   **Solution.** No, this algorithm would not always return the correct answer. Consider the first connected component having the vertex with the smallest finish time (see first true/false). Then a DFS would start from this vertex and discover the whole graph, declaring it incorrectly as a single connected component. For example, picture the graph with nodes \( a, b, c \). The edges are \((a, b), (b, a), (b, c)\). Let’s say we start at \( b \) and discover \( a \) first. Then \( a \) will finish with a finish time of 3, which is the first finish time. When we start our DFS in increasing finish time we start at \( b \), which then discovers the whole graph. This is incorrect, as the components are \((a, b)\) and \((c)\).

**Problem 2**

**Problem.** Consider a graph \( G = (V, E) \) 'almost strongly connected' if adding a single edge could make the entire graph strongly connected. Design an algorithm to determine whether a graph is almost strongly connected.
**Solution.** First, use Kosaraju’s algorithm to create the graph of SCCs, \( G_{SCC} \). Then topologically sort the graph, since it is a DAG.

If the graph is ‘almost strongly connected’, then adding a single edge will connect the graph.

Add an edge from the last component to the first, and check if the graph is now strongly connected using DFS/BFS.

**Correctness.** If the algorithm returns true, meaning our new graph was strongly connected, since we only added a single edge, it follows that the original graph was almost strongly connected.

In the case that our algorithm returns false:

For a graph to be almost strongly connected, every vertex must have a path to vertex \( s \), the source of the edge to add, and a path from \( t \), the second vertex in the new edge. If for contradiction it didn’t then the new graph must clearly have a vertex with no path to \( s \), or no path from \( t \), as adding an edge from a vertex does not affect its reachability.

\( s \) must be in the first component in \( G_{SCC} \), as it it wasn’t, then any vertex earlier in the topological order is clearly not reachable from \( s \).

\( t \) must be in the last component in \( G_{SCC} \), as if it wasn’t, then any vertex later in the topological order clearly can not reach \( t \).

**Running time.**

Steps:

1. Creating SCC kernel graph: \( O(|V| + |E|) \)

2. Topological sort: \( O(|V| + |E|) \)

3. Checking if strongly connected: \( O(|V| + |E|) \)

Therefore, this algorithm is \( O(|V| + |E|) \).