**15-451/651: Design & Analysis of Algorithms**
**Lecture #8 (draft!!)** September 18, 2013

Today's specials:

- A reminder of the depth-first search (DFS) algorithm and its properties.
- Arranging the nodes of a DAG so that edges go from left to right.
- An algorithm for finding stongly connected components in a digraph.

---

## 1 Introduction

Depth First Search: you've seen this before, in 15-210, if not even earlier. Given a graph, it is a procedure that visits all the vertices of a graph. And you can build on this in many ways, to give algorithms to:

- Determine the connected components of a graph.
- (*) Find cycles in a directed or undirected graph.
- Find the biconnected components of an undirected graph.
- (*) Topologically sort a directed graph.
- Determine if a graph is planar, and finding an embedding of it if it is.
- (*) Find the strong components of a directed graph.

If the graph has $n$ vertices and $m$ edges then depth first search can be used to solve all of these problems in time $O(n + m)$, that is, linear in the size of the graph. In this lecture we will do the final item on this list: find strong components of the graph[1] – though along the way we'll touch upon the other starred topics while reviewing some basic properties of DFS.

## 2 Depth First Search

First, some notation. We have a directed or undirected graph $G = (V, E)$. We assume $n$ nodes/vertices and $m$ edges. If it is directed, we refer to the edges as *arcs* to emphasize they are ordered. We write an arc as $(u, v)$ or just $uv$ — the arrow goes from $u$ to $v$. Node $u$ is the *tail* of the arc $uv$ and $v$ the *head*.

The input will be assumed to be in the adjacency-list format, where the adjacency lists are linked lists. (We won't need the linked list structure in this lecture.) Let `adj`$(v)$ denote the adjacency list of vertex $v$ — for directed graphs we assume it contains all the edges going out of $A$.

OK. DFS. Depth First Search. The simplest graph algorithm in the book, but it has a lot of power to it. In this lecture, we'll only consider directed graphs, the version for undirected graphs is almost identical.

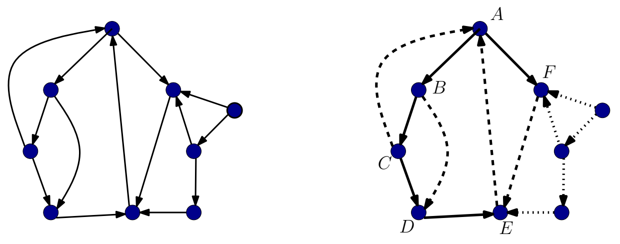
```
Initialize: for each v in V
  mark(v) = F

DFS(v)     // Invariant: v has not been marked
  mark(v) = T
```

[1]Strong components are directed analogs of connected components; we'll define them later in this lecture.

```
  for each arc (vw) in adjacency-list(v) {
    if mark(w) == F then DFS(w)
  }
```

Basically, we look at each arc and if the other side has not already been visited yet, we recursively visit it. Here's an example. The labeled nodes are the ones visited by calling DFS($A$). The dashed edges are the ones not traversed, the dotted ones were not even looked at.



A node $w$ is *reachable* from $v$ in $G$ if there is a path $v = v_0, v_1, v_2, \ldots, v_k = w$ such that each $(v_i, v_{i+1})$ is an arc of $G$.

**Fact 1** *When DFS($v$) terminates, it has visited (marked) all the nodes that can be reached from $v$.*

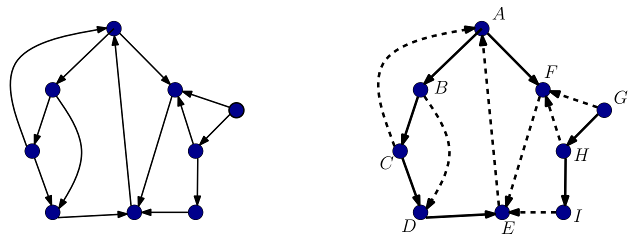
**Proof:** The simple proof is by induction. We will terminate because every call to DFS(v) is to an unmarked node, and each such call marks a node. There are n nodes, hence n calls, before we stop.

Now suppose some node w that is reachable from v and is not marked when DFS(v) terminates. Since w is reachable, there is a path $v = v_0, v_1, v_2, \ldots, v_k = w$ from $v$ to $w$, and a first node $v_i$ on this path that is not marked. But this is impossible, because we marked $v_{i-1}$ and would have examined the arc $(v_{i-1}, v_i)$. ■

Of course, it may be the case that not all the nodes in G are reachable from v. So really we should do the following

```
DFS-graph(graph G)
  for all v in V, mark(v) = F.
  While there exists an unmarked node v
     DFS(v)
```

This process will visit all the nodes of the graph (just by the definition of the procedure). Here's the old example.



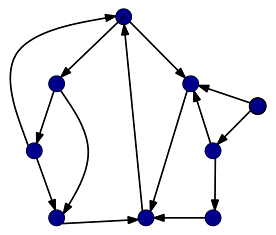
It will help to have a few more pieces of data defined, which will make reasoning about DFS much easier. One is `active`($v$), which is a flag that indicates that $v$ is currently on the recursion stack. Two other numbers are `pre`($v$) and `post`($v$) which are "times" at which we add $v$ to the recursion stack, and when we remove $v$ from it. (In 15-210, these were the times at which you ENTER $v$ and EXIT $v$.)

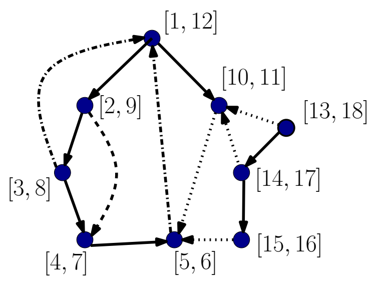
Here is the depth first search procedure:

```
$i \leftarrow 0$
for all $x \in V$ do pre($v$) $\leftarrow 0$, post($v$) $\leftarrow 0$, active($v$) $\leftarrow 0$
for all $x \in V$ do
    if pre($x$) $= 0$ then DFS($x$)

DFS($v$)
    $i \leftarrow i + 1$
    pre($v$) $\leftarrow i$
    active($v$) $\leftarrow 1$
    for all $w \in adj(v)$ do
        if pre($w$) $= 0$ then DFS($w$)              [$(v, w)$ is a tree arc ]
        else if pre($w$) $>$ pre($v$) then           [$(v, w)$ is a forward arc ]
        else if active($w$) $= 0$ then               [$(v, w)$ is a cross arc]
        else                                         [$(v, w)$ is a back arc]
    active($v$) $\leftarrow 0$
    $i \leftarrow i + 1$
    post($v$) $\leftarrow i$
end DFS
```

Below is our running example with the node labelings and the arc classification: the solid arcs are tree arcs, dashed arcs are forward arcs, the dotted arcs are cross arcs, and the dot-dashed arcs are back-arcs.





Just as above, this process examines all arcs and vertices. The call DFS($v$) is made exactly once for each vertex of the graph. Each arc is placed into exactly one of four classes by the algorithm: *tree* arcs, *forward* arcs, *cross* arcs, and *back* arcs. This classification of the arcs is not a property of the graph alone. It also depends on the ordering of the vertices in $adj(v)$ and on the ordering of the vertices in the loop that calls the DFS procedure.

The tree arcs have the property that either zero or one of them points to a given vertex. Therefore, they define a collection of trees, called the *depth-first spanning forest* of the graph. The root of each tree is the vertex with the lowest `pre` number (the one that was searched first). These rooted trees allow us to define the ancestor and descendant relations among vertices. The four types of arcs are related to the spanning forest as follows:

- The *forward* arcs are arcs from a vertex to a descendant of it that are not tree arcs. This is because the test pre($w$) > pre($v$) indicates that $w$ was explored after the call to DFS($v$). Since the call to DFS($v$) is not yet complete $w$ must be a descendant of $v$. Moreover, the exploration of $w$ must be complete (it must have been popped from the execution stack for us to return to $v$), so we'll have post($w$) < post($v$). To summarize, *forward* arcs have

$$\texttt{pre}(v) < \texttt{pre}(w) < \texttt{post}(w) < \texttt{post}(v)$$

  This pattern is also true for *tree* arcs.

- The *cross* arcs are arcs from a vertex $v$ to a vertex $w$ such that the subtrees rooted at $v$ and $w$ are disjoint. This follows because $\texttt{active}(w) = 0$ so the exploration of $w$ is complete, and was complete before the call to DFS($v$), so we'll have $\texttt{post}(w) < \texttt{post}(v) < \texttt{pre}(v)$. Therefore $v$ is not in a subtree rooted at $w$. Vertex $w$ is not in a subtree rooted at $v$ because $\texttt{pre}(w) < \texttt{pre}(v)$. To summarize, for *cross* arcs $vw$

$$\texttt{pre}(w) < \texttt{post}(w) < \texttt{pre}(v) < \texttt{post}(v)$$

- The *back* arcs are arcs from a vertex to an ancestor of it. The fact that $\texttt{active}(w) = 1$ indicates that the $w$ is on the recursion stack and is thus an ancestor of $v$. So for *back* arcs, we'll have

$$\texttt{pre}(w) < \texttt{pre}(v) < \texttt{post}(v) < \texttt{post}(w).$$

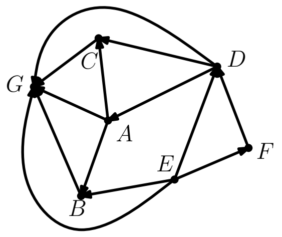
Observe that from this discussion that just looking at the pre/post numbers for two vertices $x$ and $y$, we can correctly answer the question:

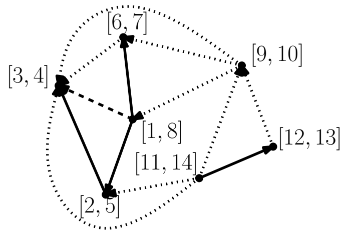
*Is $x$ an ancestor of $y$ in the DFS tree? Or is $y$ an ancestor of $x$? Or are they unrelated?*

**Exercise:** A tree $T$ rooted at some node $r$ naturally defines an ancestor-descendent relation on the nodes. Show that you can label each node with labels of size $O(\log n)$ bits so that you can answer queries of the form "is $x$ an ancestor of $y$ in the $T$, or is $y$ an ancestor of $x$, or are they unrelated?" in constant time. (Assume comparing $O(\log n)$-bit integers takes constant time.)

## 3 Topological Ordering for Directed Acyclic Graphs

You know what a *directed acyclic graph (DAG)* is, right? A directed graph that does not have any directed cycles. I.e., no directed path that starts at $v$ and ends at $v$, except of course the path of length 0. No "non-trivial" path. Below's a DAG and its DFS.





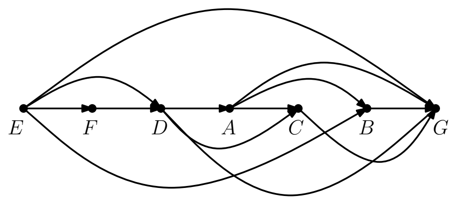
**Lemma 2** *A digraph $G$ is a DAG if and only if DFS on the graph has no back arcs.*

**Proof:** One direction is immediate: if there is a back arc (which goes from an descendent to an ancestor), it creates a cycle. On the other hand, if there is a cycle $C$ in $G$, consider the earliest node in $C$ visited by DFS (the one with the lowest $\texttt{pre}$ number). Call it $v$. Since all nodes in $C$ are reachable from $v$, they will be visited before time $\texttt{post}(v)$, and will be descendents of $v$. But then $v$'s predecessor $u$ on the cycle would have resulted in the back arc $uv$. ■

Look back at the classification of arcs into tree/forward, back, and cross. In a DAG there are no back arcs. This means all the arcs $vw$ in the graph have the property that $\texttt{post}(w) < \texttt{post}(v)$. In words, the head of each arc has a smaller $\texttt{post}$ number than the tail. So here's a way to output a total ordering of the vertices in $G$, so that all arcs in $G$ go from left to right:

*Run DFS on G.*
*Output the vertices in decreasing order of their* $\texttt{post}$ *numbering.*

For example, the DAG above will result in the following ordering.



**Exercise:** Naively, the above algorithm asks us to sort the `post` numbers, which would take $O(n \log n)$ time. Show that the algorithm can be implemented in $O(m+n)$ time by slightly altering the DFS procedure.

**Exercise:** Now suppose you forgot to alter the DFS procedure. You ran DFS, got the `post` numbers, and now you read the second line. You don't want to spend $O(n \log n)$ time to sort the $n$ `post` numbers. How can you sort them in $O(n)$ time?

Such an ordering on the nodes where all the arcs point in the same direction (say from left to right) is called a *topological ordering.* We've just shown that the graphs for which a topological ordering exists is precisely DAGs. And the exercise shows we can output a topological ordering in $O(m+n)$ time.

**Exercise:** A *source* in a digraph is a node that has no arcs coming into it. A *sink* is a node that has no arcs leaving it. Prove that each DAG must have at least one source and at least one sink. Show a digraph (that is not a DAG) that has no sources and no sinks.

**Exercise:** Prove that the following algorithm also outputs a topological ordering: *Find a sink $v$ in the current DAG. Recursively output an ordering on $G - v$ (the graph with $v$ deleted). Output $v$ at the end.*

**Exercise:** Prove that the following algorithm also outputs a topological ordering: *Find a source $v$ in the current DAG. Output $v$. Recursively output an ordering on $G - v$ (the graph with $v$ deleted).*

**Exercise:** Can you implement the algorithms outlined in the above exercises in $O(m+n)$ time?

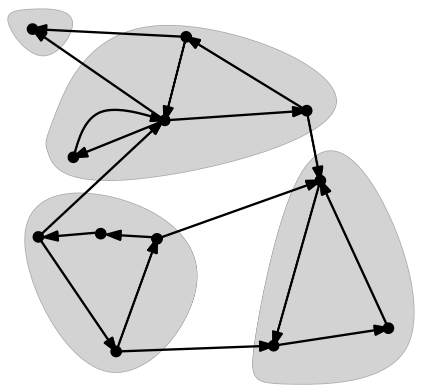
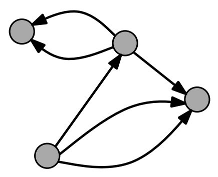
## 4 Strong Connected Components

Call two vertices $v$ and $w$ *equivalent*, denoted $v \equiv w$ if there exists a path from $v$ to $w$ and one from $w$ to $v$. The relation "$\equiv$" so defined is an *equivalence relation* because it satisfies the following three properties:

1. Reflexivity: $w \equiv w$. This is because a path of length zero goes from $w$ to $w$ and vice versa.
2. Symmetry: If $v \equiv w$ then $w \equiv v$. This follows immediately from the definition.
3. Transitivity: If $v \equiv w$ and $w \equiv x$ then $v \equiv x$. If there is a path from $v$ to $w$, and one from $w$ to $x$, then there is a path from $v$ to $x$. The same reasoning shows that there is a path from $x$ to $v$.

This equivalence relation induces a partitioning of the vertices of the graph into components in which each pair of vertices in a component are equivalent. These are called the *strongly connected components* or *strong components* of the graph. It is standard to abbreviate this as SCCs.

On the left is an example of a directed graph partitioned into its SCCs. Then if we contract the SCCs into single nodes, we get the graph on the right, which you notice is a DAG with one source and two sinks. This DAG structure is not a fluke, as we show next.

**Fact 3** *Shrinking each SCC of a graph $G$ into a single node gives us a DAG.*

**Proof:** (Sketch) If there were a cycle in the shrunk graph containing the nodes corresponding to SCCs $S_1$ and $S_2$ then we could reach from some node in $S_1$ to some node in $S_2$ (and vice versa). And hence we could reach from all nodes in $S_1$ to all nodes in $S_2$ and vice versa. So $S_1$ and $S_2$ should have been in the same SCC to begin with. ■

Our goal is to devise an algorithm which will compute the SCCs of a graph. There is a very close relationship between the strong components of a graph and the depth first spanning forest of the graph. We shall present an algorithm that uses depth first search to find the strong components of the graph.

Here's the idea behind our algorithm (called Kosaraju's algorithm).[2] There's this DAG of SCCs of $G$. Suppose we knew a vertex $v$ in the *sink* SCC. If we run DFS from $v$, we would see *exactly* the nodes in the sink SCC.[3] We could then delete these nodes. This would leave a smaller graph $G'$ with the same set of SCCs as $G$ (except the one we just deleted). We repeat — find a vertex $v'$ in a sink SCC in $G'$, use DFS to find the entire sink SCC, delete it and keep going.

A nice and clean idea. Two things to worry about. (a) How do we find a node in this sink SCC? (b) How to find this in linear time? (c) And then, how to run the entire algorithm in linear time? OK, one thing at a time.

### 4.1 Finding a Node in the Sink SCC

The crucial idea here is the following observation.

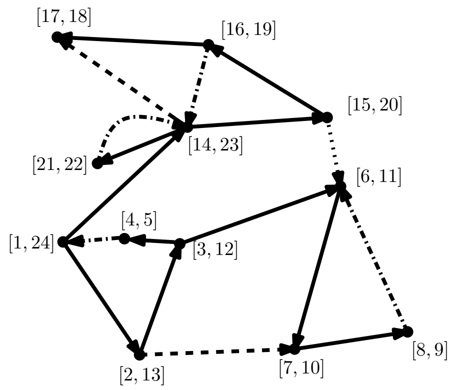
**Lemma 4** *In a run of DFS on digraph $G$, the vertex with the highest* `post` *number belongs to the* <u>*source*</u> *SCC.*

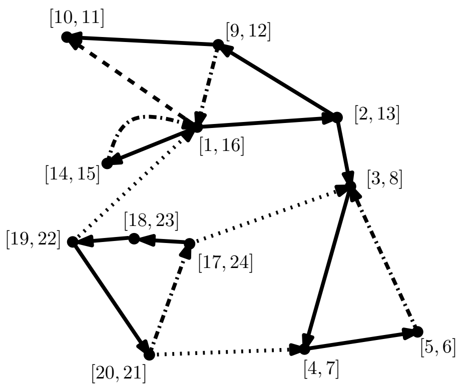
Here are two different DFS runs on the same graph with the highest `post` number in the source component.

---

[2]Kosaraju's algorithm is due to S. Rao Kosaraju. There's no paper associated with it, though — it just appears in the classic textbook of Aho, Hopcroft, and Ullman, attributed to him.

[3]Why? Clearly we'd see at least the nodes in the same SCC as $v$. The worry is that we might see more — nodes which are reachable from $v$ but can't reach $v$. That is, we might see nodes that lie in a different SCC. This is why we chose $v$ from a sink SCC. There *are* no other SCCs reachable from $v$'s SCC. So we'll see exactly the nodes in $v$'s SCC.





Oh, that sounds goo ... Wait, what? The *source* SCC? Didn't we want the *sink* SCC? For our scheme the source SCC won't do at all. The whole point was that starting DFS from a node in a sink SCC will explore exactly that SCC (which we can peel off and repeat). Starting it from a source SCC might explore a lot more.

This is actually easy to fix. Look at the *inverted digraph* $G^{\leftarrow}$ which is obtained by just reversing the directions of all arcs in $G$. A sink SCC in $G$ is then a source SCC in the inverted graph $G^{\leftarrow}$. This trick allows us to find a sink SCC in $G$.

So back to proving Lemma 4. The proof is immediate once you've proved the following claim.

**Claim 5** *If $C_1$ and $C_2$ are SCCs in $G$ with an edge from some node in $C_1$ to some node in $C_2$, then the highest* `post` *number in $C_1$ is greater than the highest* `post` *number in $C_2$.*

**Proof:** Look at the first node in $C_1 \cup C_2$ that DFS visits. Say it is node $v_1 \in C_1$, then $\texttt{post}(v_1)$ will have the highest `post` number among all these nodes. Else we visit some $v_2 \in C_2$ before any of the nodes in $C_1$ — then DFS will visit all nodes in $C_2$ before it exits $v_2$, and none in $C_1$ (because none of the nodes in $C_1$ are reachable from $v_2$). In this latter case even all the `pre` numbers of $C_1$ will be later than all the `post` numbers of $C_2$. ■

Good, good. So this says that a single DFS suffices to find a node in a source component of $G$. And the inverted digraph idea tells us that we can use this to find some node in a sink component $C_s$ of $G$. You remember the inverted digraph idea, right? We took the inverted digraph $G^{\leftarrow}$ and ran DFS on it to get the `post` numbers on the nodes. For each $v \in V$, let's denote $\texttt{post}(v)$ on this inverted graph by $\texttt{post}_{G^{\leftarrow}}(v)$.

Look at the vertex in $G$ with the highest value of $\texttt{post}_{G^{\leftarrow}}(v)$, call this $v_s$. Call the associated sink CSS $C_s$. Now run DFS($v_s$) — this will get stuck precisely after exploring $C_s$, and this DFS will just take time proportional to the number of arcs and nodes within the $C_s$.

### 4.2 Finishing the Rest in Linear Time Too

OK, so one SCC down, all the rest to go. And what about the rest?

Well, the final piece of the puzzle is also simple. Let $G'$ be the graph obtained by deleting the nodes and arcs of $C_s$ from the graph $G$. The vertex with the highest $\texttt{post}_{G^{\leftarrow}}(v)$ number among those remaining in $G'$ will be in a sink CSS of $G'$. (You see why? It again follows from Claim 5.) We can now peel off this component, and repeat. The time to peel off each SCC is proportional to the number of arcs within the SCC. Adding over all the SCCs, the total time is again $O(m + n)$.

The algorithm is now easily summarized.

```
Compute the post numbers in the graph $G^{\leftarrow}$, call the numbers $\texttt{post}_{G^{\leftarrow}}(\cdot)$.
Initialize $H \leftarrow G$
```

```
Repeat until $H$ is empty
    Let $v$ be the vertex in $H$ with the highest $\texttt{post}_{G^{\leftarrow}}(\cdot)$ value
    Run DFS($v$) in $H$ to find the associated sink SCC $C$, and delete it from $H$.
```

Again, the first step is just DFS, and each step of the loop can be charged to the nodes and edges of the component removed from $H$ in that iteration.

Short and simple. And it's all achieved just by using the simple properties of the DFS algorithm and numbering scheme.

**Exercise:** Note that when we are running DFS, we have the flexibility of choosing the start node (and also every time DFS stops before it explores the entire graph, choosing the next node to begin DFS at). Use this observation to implement the above algorithm using 2 DFSs.