CS 31: Algorithms (Spring 2019): Lecture 12

Date: 2nd May, 2019

Topic: Graph Algorithms 2: Applications of Depth First Search

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please notify errors on Piazza/by email to deeparnab@dartmouth.edu.

1 Applications of DFS

In this lecture, we see two applications of DFS on directed graphs. The first deals with directed acyclic graphs, also called DAGs. The second is a magical application of DFS to find the strongly connected components of a directed graph.

1.1 Topological Ordering of DAGs

Throughout this subsection, G is a directed acyclic graph (DAG). Recall from the previous lecture, this means that if we run DFS on G, there are no back edges. DAGs are very useful and appear in many applications some of which we may visit in the problem set. Here is a property of DAG which follows from definition.

Claim 1. Every DAG G has at least one sink vertex v with $\deg^+(v) = 0$ and one source vertex v with $\deg^-(v) = 0$.

Exercise: *Complete the proof*

A topological ordering of a DAG is an ordering $\sigma[1:n]$ of the vertices such that for any $i < j$, there is **no** edge from $\sigma[j]$ to $\sigma[i]$. That is, if we write down the numbers from left to right in the σ order, then *all* edges go from left to right. If one thinks of an edge (u, v) as v being "bigger" than u, then the topological ordering is a linearization of the graph according to this (partial) order. Of course, not every pair of vertices may be comparable. Note that the first vertex in the topological order must be a source and the last vertex must be a sink.

Exercise: *Is the topological ordering of a DAG G unique? If not, is there any special case when it is?*

TOPOLOGICAL ORDERING
Input: Directed Acyclic Graph G.
Output: A topological ordering of G.

Consider running DFS in any arbitrary order on the DAG G.

Lemma 1. Let σ be the ordering of the vertices in *decreasing* order of $\mathsf{last}[v]$. Then, σ is a topological order.

Proof. Let x and y be two vertices with $\mathsf{last}[x] < \mathsf{last}[y]$. We need to show (x, y) can't be an edge in G. Suppose it is. If $\mathsf{first}[x] < \mathsf{first}[y]$, then the edge property would imply $\mathsf{last}[y] < \mathsf{last}[x]$ which contradicts the choice. Therefore, we get $\mathsf{first}[y] < \mathsf{first}[x]$. But then (x, y) is a *back edge* which contradicts the acyclicity of G. $\square$

Theorem 1. TOPOLOGICAL ORDERING of any DAG G can be found in $\Theta(n + m)$ time.

Proof. There is one extra thing needed to argue about following the previous lemma. We need the (decreasing) sorted order of lasts; how do we get that in $\Theta(n + m)$ time. Two answers.

One, the $\mathsf{last}[v]$'s are integers between 1 and $2n$; so we can sort in $\Theta(n)$ time using Count-Sort. Two, and this is less modular, but we can read out the numbers in increasing order of $\mathsf{last}[v]$ as DFS as being run; after all, it is precisely the order in which the $\mathsf{last}[v]$'s are set. The topological order is the reverse of that. $\square$

1.2 Strongly Connected Components (SCCs) using DFS

This is a deep application which really illustrates the power of DFS. Let's try to illustrate the main ideas of the algorithm before giving the final description and analysis. To do so, consider the graph in Figure 1.

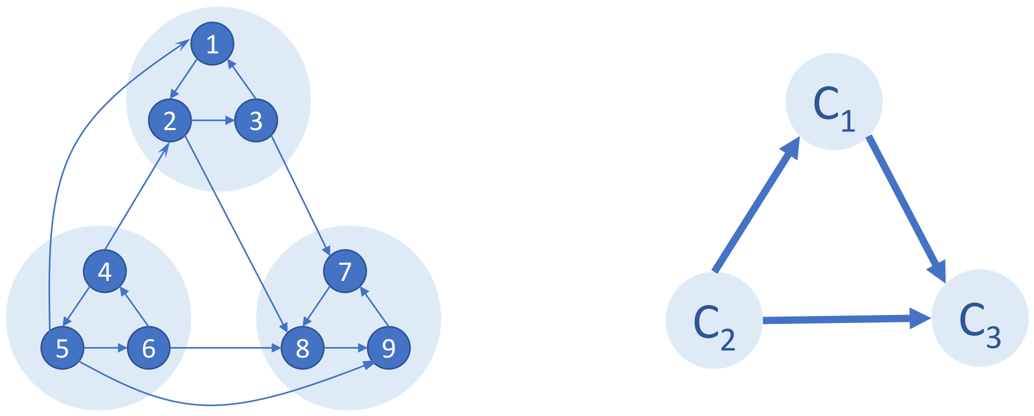

Figure 1: Illustrative example for SCC.

In the graph on the left, there are three strongly connected components marked in light blue circles. The graph on the right is one whose vertices are these three components, and we have an edge between two components (for instance from C_1 to C_2) if and only if there is an edge (u, v) in the original graph with $u \in C_1$ and $v \in C_2$. Note there could be, and in this example there are, multiple such edges. We require that there be at least one. In general, given a graph G as on the left, then the graph on the right is called G^{scc}; note we don't have this graph up front but is useful for analysis and designing the algorithm.

Observation 1. G^{scc} is a DAG.

Exercise: *Complete the proof.*

Before we move on to discovering the SCCs, let us see why the algorithm for undirected graphs is not enough. Recall what we did for undirected graphs; we ran DFS on G in any order and returned the connected components of the forest. Why doesn't it work? Well, in the graph above consider what happens when we run DFS from the vertex 1. You see that *all* the vertices are reachable from 1 and thus end up in the tree rooted at 1. The resulting vertices are not strongly connected. To stress why this is not an issue in undirected graphs note that in undirected graphs if there is a path from a vertex 1 to a set of vertices S, then there is a path from any vertex in S to 1 as well. This is patently false in directed graphs.

An Encouraging Idea. Suppose instead we ran DFS from vertex number 9. Then, we would definitely discover all the vertices that 9 can reach. But these are precisely the ones in C_3, the strongly connected component connecting 9. Why was this? This is because, there is no edge which starts from inside C_3 and goes outside. That is, because C_3 is a *sink* component of G^{scc}. But this is wonderful; there is some vertex from which if we start DFS we get what we want. Let us make this our goal for now.

From our understanding of topological ordering in DAGs, our experience is that the vertex with the smallest $\mathsf{last}[v]$ is a sink vertex in a DAG. Perhaps, we could conjecture it for a general graph: in any graph G and any DFS run, the vertex with the smallest $\mathsf{last}[v]$ must lie in a sink component of G^{scc}. *Unfortunately, this idea has a hole!*

Exercise: *Find an example disproving the above conjecture.*

A philosophical interlude. In research, you often think you have an understanding of objects, and this leads you to make some conjectures. Just like we did above. And often they are wrong. I'll not lie – disappointment is usually the first response. But what really defines a researcher is resilience. Counterexamples are the world's ways of telling us, "Your understanding was incomplete. Refine them. Think harder." And when you do get back to the drawing board, or square one, the world often rewards you with *epiphanies*.

Epiphany 1. Although the vertex with the *smallest* $\mathsf{last}[v]$ may not be in a SINK component of G^{scc} (assuming you did the exercise above), it is in fact true that the vertex with the *largest* $\mathsf{last}[v]$ *does indeed* lie in the SOURCE component of G^{scc}.

In fact, more is true. For any component $C \in G^{\mathsf{scc}}$, define

$$f(C) = \max_{v \in C} \mathsf{last}[v]$$

Lemma 2. If (C_i, C_j) is an edge in G^{scc}, then $f(C_i) > f(C_j)$.

Proof. Before we prove the lemma, let's take a look at the definition again. For any component C, let x be the vertex in C with the largest last. Indeed, this vertex also must have the *smallest* first.

Claim 2. For any strongly connected component C if $x \in C$ has the largest last, then it also has the smallest first. In particular, x's interval contains all the intervals of every other vertex in C.

Proof. Suppose not. Suppose $y \in C$, $y \neq x$ has the smallest first. Since C is strongly connected, there is a path from y to x. y has the smallest first among all vertices in this path, so by the *path property*, it must have the the largest last among all vertices in this path. In particular, $\mathsf{last}[y] > \mathsf{last}[x]$. Contradiction. □

Now the proof of the lemma is simple. Let x be the vertex in C_i with the largest last and y be the vertex in C_j with the largest last. For the sake of contradiction, assume $\mathsf{last}[x] < \mathsf{last}[y]$. By the Nested Interval Property, either (a) $\mathsf{first}[y] < \mathsf{first}[x]$, that is, x's interval is completely contained in y's interval, or (b) $\mathsf{last}[x] < \mathsf{first}[y]$, that is x's interval is disjoint and lies before y's interval.

We will reach a contradiction in both cases. Case (a) is easy: if x's interval is completely contained in y's interval, then there is a path from y to x in the DFS forest. In particular, that would imply an edge from C_j to C_i in the G^{scc} contradicting the DAG nature of G^{scc}.

In Case (b), x's interval finishes before y's interval. By the claim above, this means that the interval of *every* vertex in C_i finishes before the interval of *any* vertex in C_j starts. Now since (C_i, C_j) is an edge, there is some $u \in C_i$ and $v \in C_j$ such that (u, v) is an edge in G. From the claim, we see $\mathsf{first}[u] < \mathsf{last}[u] < \mathsf{first}[v] < \mathsf{last}[v]$; this contradicts the *edge property*. □

The above lemma means that arranging the components by increasing order of $f(C_j)$ gives the topological order of G^{scc}; a generalization of the topological ordering theorem for DAGs where $G = G^{\mathsf{scc}}$. It also implies that the largest $\mathsf{last}[v]$ must lie in a source vertex of G^{scc}. Suppose not, then if it lies in C_j and there is an edge (C_i, C_j), the above lemma implies $f(C_i) > f(C_j)$ contradicting the choice of v. Coming back to the problem at hand, why is the above useful in finding a vertex in the *sink* component? A second epiphany answers this.

***Epiphany* 2.** Let G_{rev} be the graph where all edges of G have been reversed. Observe that the strongly connected components of G_{rev} are the precisely the same as those in G, and that $(G_{\mathsf{rev}})^{\mathsf{scc}} = (G^{\mathsf{scc}})_{\mathsf{rev}}$. In other words, the source components of $(G)^{\mathsf{scc}}$ are precisely the sink components of $(G_{\mathsf{rev}})^{\mathsf{scc}}$. Therefore, if we run DFS on G and look at the vertex with the largest $\mathsf{last}[v]$ that is guaranteed to be in the *sink* component of $(G_{\mathsf{rev}})^{\mathsf{scc}}$. Which will allow us to find the strongly connected components of (G_{rev}). Which is the same as the strongly connected components of G. Done!

```
1: procedure StronConnComp($G$):
2:     ▷ Returns the strongly connected components of G
3:     Run DFS($G, \{1, 2, \ldots, n\}$) to get $\mathsf{last}[v]$ for every vertex.
4:     $\pi$ be the decreasing order of $\mathsf{last}[v]$'s. ▷ Can be found in $\Theta(n)$ time a la Top. Ord.
5:     Obtain $G_{\mathsf{rev}}$. ▷ This takes $\Theta(n + m)$ time.
6:     Run DFS($G_{\mathsf{rev}}, \pi$) and return the connected components of the forest $F$.
```

Lemma 3. The above algorithm returns the strongly connected components correctly.

Proof. Let $C_1, \ldots, C_k$ be the components of G^{scc}. We claim that the components of the final forest F returned in Line 6 are these components returned in topological order of G^{scc}. We assume this is true for the first i components returned in Line 6 and we argue about the $(i + 1)$th component; we start with $i = 0$ (so in the beginning we have a vacuous statement).

Note that the vertex v picked as root at the $(i + 1)$th step is the first vertex in π not in $C_1 \cup \ldots \cup C_i$. That is, $\mathsf{last}[v] = \max_{u \notin C_1, \cdot, C_i} \mathsf{last}[u]$. Suppose v lies in component C_j of G^{scc}. Lemma 2 implies C_j is a *source* component in $G^{\mathsf{scc}} \setminus (C_1 \cup \cdots \cup C_i)$. Suppose not; then there is some (C_k, C_j) edge, $k > i$, which implies $f(C_k) > f(C_j)$ contradicting the choice of v. Therefore C_j is a source component of $G^{\mathsf{scc}} \setminus (C_1 \cup \cdots \cup C_i)$.

That is, C_j is a sink component of $(G_{\mathsf{rev}})^{\mathsf{scc}} \setminus (C_1 \cup \cdots \cup C_i)$. Now notice that the DFS run from vertex v on G_{rev} will only discover vertices in C_j as it is a sink component. That is, the $(i + 1)$th step discovers a sink component of $(G_{\mathsf{rev}})^{\mathsf{scc}} \setminus (C_1 \cup \cdots \cup C_i)$, and thus consistent with the reverse topological order in $(G_{\mathsf{rev}})^{\mathsf{scc}}$ which is the topological order of G^{scc}. □