CS 341: ALGORITHMS (S18) — LECTURE 13

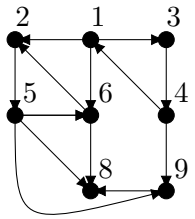
DEPTH-FIRST SEARCH IN DIRECTED GRAPHS

ERIC BLAIS

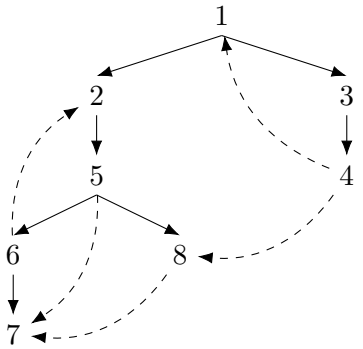
In the first lecture of this section of the course, we introduced directed (or *ordered*) graphs, but so far we have only considered algorithms on undirected graphs. In this lecture, we now turn our attention to directed graphs.

1. DFS tree in directed graphs

Recall that a directed graph is a graph where each edge is directed between a *source* vertex and a *destination* vertex. We can run the DFS algorithm as-is on directed graphs (following edges only in the direction from their source to their destination), and the algorithm again builds a DFS tree. For example, on the input graph



the DFS algorithm builds the following DFS tree



As the example shows, in the directed graph setting, we are no longer guaranteed to have non-tree edges connecting only ancestors and descendants. In this setting, the non-tree edges belong to one of three categories: *forward edges* that lead from an ancestor to a descendant, *backward edges* that lead from a descendant to one of its ancestors, and *cross edges* to lead from a vertex to another one that is neither its ancestor nor its descendant. In the DFS tree above, for example, the edge $(5, 7)$ is a forward edge, the edges $(4, 1)$ and $(6, 2)$ are backward edges, and the edges $(4, 8)$ and $(8, 7)$ are cross edges.

2. Directed acyclic graphs

A *cycle* in a directed graph is a circular path $v_0 \to v_1 \to \cdots \to v_k \to v_0$. (Note that unlike in undirected graphs, the path $v_0 \to v_1 \to v_0$ is a cycle.) A directed graph that contains no cycle is known as a *directed acyclic graph*, or *dag* for short: these graphs are particularly important, and appear in many different contexts. It's therefore important to be able to tell efficiently when a graph is a dag.

Definition 13.1 (Testing DAGs)**.** Given a directed graph $G = (V, E)$, determine if it is a directed acyclic graph.

We can solve the problem of testing dags using depth-first search.

Theorem 13.2. *A directed graph $G = (V, E)$ contains a cycle if and only if its DFS tree contains a backward edge.*

Proof. If (u, v) is a backward edge in the DFS tree for G, then the path from v to u in the tree along with the edge (u, v) forms a cycle.

If G contains a cycle, we can write it as $v_0 \to v_1 \to \cdots \to v_k \to v_0$ where v_0 is the first vertex in the set $\{v_0, v_1, \ldots, v_k\}$ to be discovered by the DFS. Then each vertex $v_1, \ldots, v_k$ is a descendant of v_0 in the DFS since the subtree rooted at v_0 includes all the vertices reachable from v_0 in G. So the edge (v_k, v_0) must be a backward edge. □

We then have a simple algorithm for checking if a directed graph is a DAG: run the DFS algorithm, and check if any of the non-tree edges encountered along the way is a backward edge. (Exercise: see how you can use the $post[v]$ values to determine if an edge (u, v) is a backward edge or not.)

3. Topological sorting of DAGs

DAGs can be used to represent chains of dependencies. For example, if I want to make a fancy cup of coffee, I need to grind some beans, boil water, pour the water over the beans, prewarm the cups, etc. Some of these tasks can be done in any order—whether I grind the beans first or boild the water first does not (really) matter—but other tasks can only be done after others have been completed—pouring the water over the beans before you boil it will not give the same result! A directed graph can represent the tasks (as nodes) and the dependencies between the tasks (as directed edges); when the graph is a DAG, it means that there are no circular dependencies.

If the DAG represents a set of tasks that need to be completed and the edges represent dependencies between the tasks, the problem of determining in what order we should complete all the tasks (so that we complete a task only when all the ones that it depends on have already been completed) corresponds to the problem of *linearizing*, or *ordering*, a DAG.

Definition 13.3 (Linearizing DAGs problem)**.** Given a directed acyclic graph $G = (V, E)$, find an order $v_1, \ldots, v_n$ of the vertices in V such that for every directed edge $(v_i, v_j) \in E$, we have $i < j$.

It's not immediately clear that every DAG can be linearized, much less that we can find a linearization easily. But, as it turns out, the DFS algorithm already does so!

Theorem 13.4. *For any directed acyclic graph $G = (V, E)$, after we run the DFS algorithm, every edge $(u, v) \in E$ satisfies $post[u] > post[v]$.*

Proof. PostVisit(u) will be called only when we have finished exploring all of the vertices that are reachable from u (including v itself). □

We can then sort the vertices of G by their post number to linearize G or, alternatively, build the sorted order as we perform the DFS itself with a slight modification of the PostVisit function.

There are two types of nodes in a DAG worth introducing here because they will appear again in later lectures:

Definition 13.5 (Source and sink vertices)**.** A *source vertex* in a dag $G = (V, E)$ is a vertex with in-degree 0. A *sink vertex* in G is a vertex with out-degree 0.

In a linearization of G, the first vertex must be a source vertex, and the last vertex must be a sink vertex.

4. Strong connectivity

A directed graph $G = (V, E)$ is *strongly connected* if for every pair of vertices $u, v \in V$, there is a path from u to v and a path from v to u in G. Can we efficiently determine whether a directed graph is strongly connected or not?

Definition 13.6 (Testing strong connectivity)**.** Given a directed graph $G = (V, E)$, determine whether it is strongly connected.

When we run the DFS algorithm on a directed graph G from the start vertex s, we find the set of vertices in V that are reachable from s in G—but it's not always the case that there is also always a path from those vertices to s in G.

However, it is easy to find the set of vertices $v \in V$ for which there is a path from v to s in G: run DFS on the directed graph G^R obtained by flipping the direction of every edge in G!

So by running the DFS algorithm on G and on its reversal G^R, we can test whether s is strongly connected to every vertex in the graph. The following lemma shows that this is enough to test if the graph G is strongly connected.

Lemma 13.7. *Fix any vertex $s \in V$. The directed graph $G = (V, E)$ is strongly connected if and only if for every $v \in V$, there is a path from s to v and from v to s in G.*

Proof. If G is strongly connected, the conclusion is true for every pair of vertices $u, v \in V$, so it is also true for every pair where $u = s$.

Conversely, if there is a path from every v to s and from s to every v, then for any pair $u, v \in V$, there is a path from u to v obtained by following the path from u to s and then the path from s to v, and there is a path from v to u by going from v to s and from s to u, so G is strongly connected. □

4.1. Finding strongly-connected components.

In the setting of undirected graphs, we saw that every graph can be partitioned into disjoint connected components. And it is easy to find the set of connected components of a graph: run a depth-first or breadth-first search from any initial vertex $s \in V$ to find the first connected component, then repeat the same process on any vertex that is not covered by any of the connected components identified so far until no such vertices remain.

In the setting of directed graphs, there is a natural analogue of connected components: a *strongly-connected component* of a directed graph $G = (V, E)$ is a maximal set $S \subseteq V$ of

vertices where for every pair $u, v \in S$, there is a path from u to v and a path from v to u in G.

Can we modify the strong connectivity test above to find the strongly-connected components of a directed graph? This is not nearly as simple as in the undirected setting! In fact, there are quite a few clever algorithms that have been designed for this problem, using some of the ideas that we have seen in today's lecture. I highly recommend trying this yourselves before you look at our textbooks to see those solutions:

Challenge 1. *Design an algorithm that identifies the strongly-connected components of a directed graph in time $O(n + m)$.*