U.C. Berkeley — CS170: Intro to CS Theory | Handout N5
Professor Luca Trevisan | September 11, 2001

Notes for Lecture 5

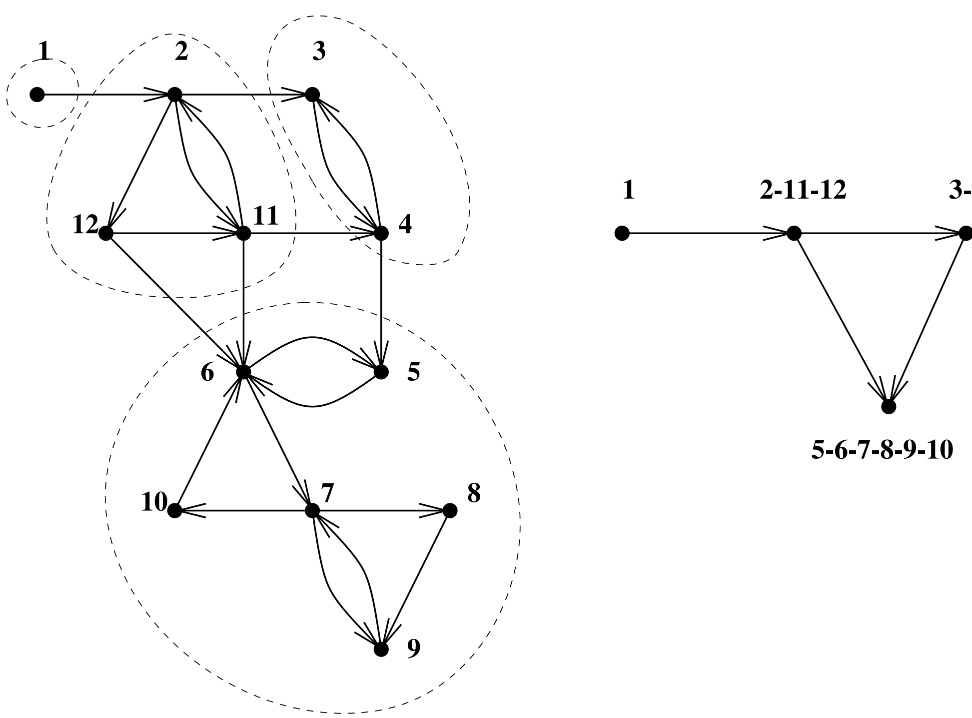
1 Strongly Connected Components

Connectivity in undirected graphs is rather straightforward: A graph that is not connected is naturally and obviously decomposed in several *connected components*. As we have seen, depth-first search does this handily: Each call of `explore` by `dfs` (which we also call a "restart" of `explore`) marks a new connected component.

In *directed graphs*, however, connectivity is more subtle. In some primitive sense, the directed graph above is "connected" (no part of it can be "pulled apart," so to speak, without "breaking" any edges). But this notion does not really capture the notion of connectedness, because there is no path from vertex 6 to 12, or from anywhere to vertex 1. The only meaningful way to define connectivity in directed graphs is this:

Call two vertices u and v of a directed graph $G = (V, E)$ *connected* if there is a path from u to v, *and* one from v to u. This relation between vertices is reflexive, symmetric, and transitive (check!), so it is an equivalence relation on the vertices. As such, it partitions V into disjoint sets, called the *strongly connected components* of the graph. In the figure below is a directed graph whose four strongly connected components are circled.

Strongly Connected Components



If we shrink each of these strongly connected components down to a single vertex, and draw an edge between two of them if there is an edge from some vertex in the first to some vertex in the second, the resulting directed graph has to be a *directed acyclic graph (dag)*—that is to say, it can have no cycles (see above). The reason is simple: A cycle containing several strongly connected components would merge them all to a single strongly connected component. We can restate this observation as follows:

CLAIM 1
Every directed graph is a dag of its strongly connected components.

This important decomposition theorem allows one to fathom the subtle connectivity information of a directed graph in a two-level manner: At the top level we have a dag, a rather simple structure. For example, we know that a dag is guaranteed to have at least one *source* (a vertex without incoming edges) and at least one *sink* (a vertex without outgoing edges), and can be topologically sorted. If we want more details, we could look inside a vertex of the dag to see the full-fledged strongly connected component (a tightly connected graph) that lies there.

This decomposition is extremely useful and informative; it is thus very fortunate that we have a very efficient algorithm, based on depth-first search, that finds it *in linear time!* We motivate this algorithm next. It is based on several interesting and slightly subtle properties of depth-first search:

LEMMA 2
If depth-first search of a graph is started at a vertex u, then it will get stuck and restarted precisely when all vertices that are reachable from u have been visited. Therefore, if depth-first search is started at a vertex of a sink strongly connected component (a strongly connected component that has no edges leaving it in the dag of strongly connected components), then it will get stuck after it visits precisely the vertices of this strongly connected component.

For example, if depth-first search is started at vertex 5 above (a vertex in the only sink strongly connected component in this graph), then it will visit vertices 5, 6, 7, 8, 9, and 10—the six vertices form this sink strongly connected component—and then get stuck. Lemma 2 suggests a way of starting the sought decomposition algorithm, by finding our first strongly connected component: Start from any vertex in a sink strongly connected component, and, when stuck, output the vertices that have been visited: They form a strongly connected component!

Of course, this leaves us with two problems: (A) How to guess a vertex in a sink strongly connected component, and (B) how to continue our algorithm after outputting the first strongly connected component, by continuing with the second strongly connected component, etc.

Let us first face Problem (A). It is hard to solve it directly. There is no easy, direct way to obtain a vertex in a sink strongly connected component. *But* there is a fairly easy way to obtain a vertex in a *source* strongly connected component. In particular:

LEMMA 3
The vertex with the highest `post` *number in depth-first search (that is, the vertex where the depth-first search ends) belongs to a source strongly connected component.*

Lemma 3 follows from a more basic fact.

Lemma 4
Let C and C' be two strongly connected components of a graph, and suppose that there is an edge from a vertex of C to a vertex of C'. Then the vertex of C visited first by depth-first search has higher `post` *than any vertex of C'.*
Proof: There are two cases: Either C is visited before C' by depth-first search, or the other way around. In the first case, depth-first search, started at C, visits all vertices of C and C' before getting stuck, and thus the vertex of C visited first was the last among the vertices of C and C' to finish. If C' was visited before C by depth-first search, then depth-first search from it was stuck before visiting any vertex of C (the vertices of C are not reachable from C'), and thus again the property is true. $\Box$

Lemma 4 can be rephrased in the following suggestive way: *Arranging the strongly connected components of a directed graph in decreasing order of the highest* `post` *number topologically sorts the strongly connected components of the graph!* This is a generalization of our topological sorting algorithm for dags: After all, a dag is just a directed graph with singleton strongly connected components.

Lemma 3 provides an indirect solution to Problem (A): Consider the *reverse* graph of $G = (V, E)$, $G^R = (V, E^R)$—G with the directions of all edges reversed. G_R has precisely the same strongly connected components as G (why?). So, if we make a depth-first search in G^R, then the vertex where we end (the one with the highest `post` number) belongs to a source strongly connected component of G^R—that is to say, a sink strongly connected component of G. We have solved Problem (A)!

Onwards to Problem (B): How to continue after the first sink component is output? The solution is also provided by Lemma 4: After we output the first strongly connected componentand delete it from the graph, the vertex with the highest `post` from the depth-first search of G^R among the remaining vertices belongs to a sink strongly connected component of the remaining graph. Therefore, we can use the same depth-first search information from G^R to output the second strongly connected component, the third strongly connected component, and so on. The full algorithm is this:

1. Perform DFS on G^R.
2. Run DFS, labeling connected components, processing the vertices of G in order of decreasing `post` found in step 1.

Needless to say, this algorithm is as linear-time as depth-first search, only the constant of the linear term is about twice that of straight depth-first search. (*Question:* How does one construct G^R—that is, the adjacency lists of the reverse graph—from G in linear time? And how does one order the vertices of G in decreasing `post[v]` also in linear time?) If we run this algorithm on the directed graph above, Step 1 yields the following order on the vertices (decreasing post-order in G^R's depth-first search—watch the three "negatives" here): 5, 6, 10, 7, 9, 8, 3, 4, 2, 11, 12, 1, (assuming the are visited taking the lowest numbered vertex first). Step 2 now discovers the following strongly connected components: component 1: 5, 6, 10, 7, 9, 8; component 2: 3, 4; component 3: 2, 11, 12; and component 4: 1.

Incidentally, there *is* more sophisticated connectivity information that one can derive from *undirected* graphs. An *articulation point* is a vertex whose deletion increases the number of connected components in the undirected graph (in the graph below there are four articulation points: 3, 6, and 8. Articulation points divide the graph into *biconnected components* (the pieces of the graph between articulation points); this is not quite a partition, because neighboring biconnected components share an articulation point. For example, the graph below has four biconnected components: 1-2-3-4-5-7-8, 3-6, 6-9-10, and 8-11. Intuitively, biconnected components are the parts of the graph such that any two vertices have not just one path between them but two vertex-disjoint paths (unless the biconnected component happens to be a single edge). Just like any directed graph is a dag of its strongly connected components, any undirected graph can be considered *a tree of its biconnected components*. Not coincidentally, this more sophisticated and subtle connectivity information can also be captured by a slightly more sophisticated version of depth-first search.