CS 3510 Algorithms 2/9/2023

Lecture 7: Strongly Connected Compeonts

Lecturer: Abrahim Ladha *Scribe(s): Joseph Gulian*

Reminder for tests, don't start running algorithms on the sinks. Run algorithms how we tell you to run algorithms (often in alphabetical order). When you look at a picture of a graph, you are secretly and subconsciously computing things.

1 Connected Components

Most graphs we cover in this class are connected because if we had a graph that was not connected, we could decompose it into it's connected parts and study those. It's trivially easy to find connected components in an undirected graph: run DFS. [1] The nodes found in each explore call will be the connected components; if you wanted to count the number of components, you could count the number of top level explore calls.

In a directed graph, nodes are strongly connected if there exists a path to and from the nodes with respect to all other nodes. In other words, if there is a cycle. This means that your graph won't always be disconnected, sometimes it may simply be that a node can reach another node, but the second node can not go back.

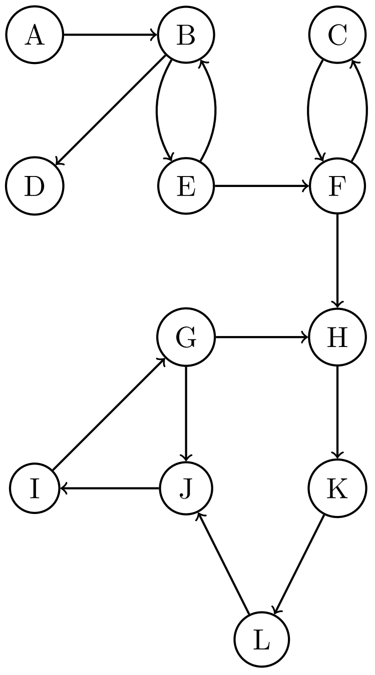

Figure 1: A directed graph.

[1] There's an important distinction explore and dfs in the context of disconnected graphs. The explore routine that we've outlined will only explore within one connected component; dfs will explore all components. Confusing these is a common mistake, don't make it.

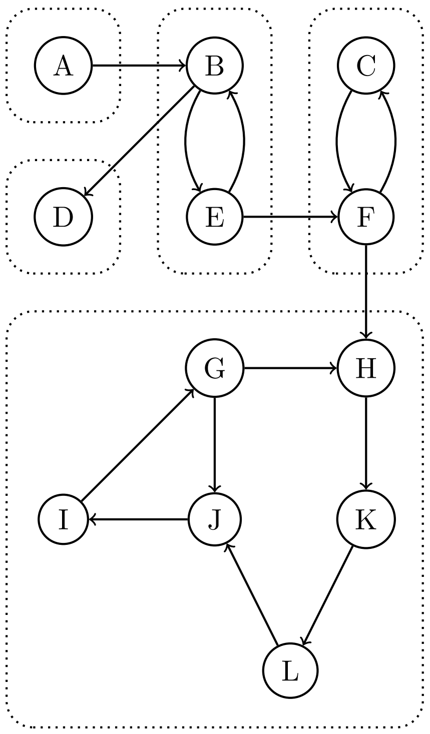

Figure 2: The components in a directed graph.

In this graph, the strongly connected components can be found by looking at it. Note that a vertex can not be contained in two components. This is because if a vertex is part of one cycle and part of another cycle, there is a cycle between all components. Now, let's construct the metagraph, a graph with these components as vertices, and edges between components where they appear in the original graph.

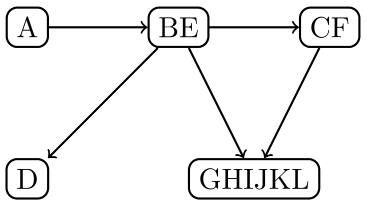

Figure 3: Components in a directed graph.

2 Metagraphs and Graph properties

Here's our metagraph; it happens to be a DAG. It's interesting to consider whether all metagraphs are DAGs, so let's try to prove it. Assume to the contrary that there exists a metagraph with a cycle. This would mean, the two components would themselves be strongly connected as we've deduced earlier. This means those nodes should have been in the same metanode, a contradiction to our correctness.

We want to create an algorithm to find strongly connected components in directed graphs. This isn't that simple. Say you call DFS on the graph like we did for connected components with undirected graphs. [2] Instead we use an algorithm which is a little more advanced than dfs called the SCC algorithm.

As an aside, many people have this intuition that there's an entrance and an exit to a graph. This could be formalized by the concept of sources and sinks. A source is a vertex with in-degree 0 (no edges entering), and a sink is a vertex with out-degree 0 (no edges exiting). Think about why this could be useful along with some of the algorithms we've previously devised by running explore from the sink and source (although we already ran it from the sink). Doing this you might notice that running explore from the sink of the metagraph reveals one strongly connected component of the graph.

Of course now, we have another problem, how do we get only the sinks of a graph. Recall what we did for top-sort: noting when a vertex was entered and exited. Performing this on the above graph, we get the following tree.

$$\begin{tikzcd}
 & & A^{1,24} \arrow[d] & & \\
 & & B^{2,23} \arrow[dl] \arrow[dr] & & \\
 & D^{3,4} & & E^{5,22} \arrow[d] & \\
 & & & F^{6,21} \arrow[dl] \arrow[dr] & \\
 & & C^{7,8} & & H^{9,20} \arrow[dl] \\
 & & & K^{10,19} \arrow[dl] & \\
 & & L^{11,18} \arrow[dl] & & \\
 & J^{12,17} \arrow[dl] \arrow[dr] & & & \\
G^{13,14} & & I^{15,16} & &
\end{tikzcd}$$

Figure 4: The dfs traversal of the graph.

If you thought about how you would top-sort this (ignoring cycles), it would put the source in the back. We're not guaranteed the sink would be the last element. Consider ordering by post label the following graph with a dfs which explores in alphabetical order. You'd get the ordering A, B, and C which is not correct because C is actually in the source of the graph.

[2] Think about this. Calling on the first graph from the letter A would produce A, B, C, D, E, F, G, H, I, J, K, and L. Clearly this is not what we want as we can discover by looking at the answer.

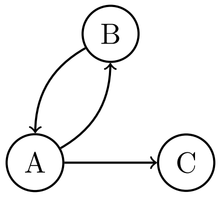

Figure 5: A graph whose loose topological sort does not yield a sink as the last vertex.

3 Proof and Algorithm

We will need to prove one thing before we can actually get a correct algorithm: If C and C' are strongly connected components, and there is an edge from a node in C to a node in C', then the highest post number in C is bigger than the highest post number in C'. We'll prove this by cases. Say DFS finds C before C', it will explore $C \to C'$, and C' will have a smaller post label. Say DFS finds C' before C, C' will be added and then C will be added later through a separate explore call, so it will have a higher post label.

Considering that the first element in a "top-sort" will be a source, but we're looking for a sink. Consider what reversing all the edges will do; it will make the sources sinks and the sinks sources. Now, we can sort by post labels, and our last element will be a source of the reverse graph G^R, but a sink in our real graph G.

An interesting proof arises with G and G^R wherein you might think that G and G^R have the same metagraph, simply with the reversed edges. This is correct and consider why. Consider the definition of a strongly connected component: there are cycles between all the vertices in a strongly connected component. Now consider if you reverse all the edges. All the edges in the cycle will be reversed but it will still form a cycle (simply in the reverse direction). All other edges will be reversed.

Considering all of this we come to the following algorithm for computing strongly connected comonents.

```
def scc(G):
    compute the reverse graph GR from G
    post-label-sort = top-sort(GR)
    initialize a list of sets of vertices result
    for element in post-label-sort:
        result += explore(G,element)
    return result
```

Figure 6: Strongly connected components algorithm

This algorithm works how you might think given the ideas we've devised. We find a source of `GR`, so its guaranteed to be in the sink of `G`. We then explore this entire sink component in `G`. We remove this explored component from `G`, and its nodes from our topsort, and repeat until we have all components. A common confusion is that even though top sort doesn't work on cyclic graphs, as we've devised top-sort, it will sort by post labels which is all we care about. **Note carefully that we call our "top-sort" on `GR`, but we explore on `G`**.

It's also worth noting that this is essentially doing the same amount of work as two DFS calls, meaning strongly connected components run in linear time.