Solution Set

CIS 121—Data Structures and Algorithms—Fall 2020

Topological Sort, Strongly Connected Components—Monday, October 26 / Tuesday, October 27

Readings

- Lecture Notes Chapter 18: DAGs and Topological Sort
- Lecture Notes Chapter 19: Strongly Connected Components

Problems

Problem 1

1. (True/False) Every DAG has exactly one topological ordering.
2. (True/False) If a graph has a topological ordering, then a depth-first traversal of the same graph will not see any back edges.

1. False. Take for example, a graph with no edges. Any ordering is valid.
2. True. If we are able to topologically sort it, then it must be a DAG. That means there will not be any back edges.

Problem 2

Consider a graph $G = (V, E)$ 'almost strongly connected' if adding a single edge could make the entire graph strongly connected. Design an algorithm to determine whether a graph is almost strongly connected.

Solution

Algorithm: First, use Kosaraju's algorithm to create the graph of SCCs, G_{SCC}. Then topologically sort the graph, since it is a DAG. If the graph is 'almost strongly connected', then adding a single edge will connect the graph.
Add an edge from the last component to the first, and check if the graph is now strongly connected using DFS/BFS.

Correctness. If the algorithm returns true, meaning our new graph was strongly connected, since we only added a single edge, it follows that the original graph was almost strongly connected.

In the case that our algorithm returns false: For a graph to be almost strongly connected, every vertex must have a path to vertex s, the source of the edge to add, and a path from t, the second vertex in the new edge. If for contradiction it didn't then the new graph must clearly have a vertex with no path to s, or no path from t, as adding an edge from a vertex does not affect its reachability. t must be in the first component in G_{SCC}, as it it wasn't, then any vertex earlier in the topological order is clearly not reachable from t. s must be in the last component in G_{SCC}, as if it wasn't, then any vertex later in the topological order clearly can not reach s.

Running time. Steps:

1. Creating SCC kernel graph: $O(|V| + |E|)$
2. Topological sort: $O(|V| + |E|)$
3. Checking if strongly connected: $O(|V| + |E|)$

Therefore, this algorithm is $O(|V| + |E|)$.

Problem 3

1. (True/False) The finish times of all vertices in a SCC s must be greater than the finish times of other SCCs reachable from s during the first DFS.

2. How does the number of SCC's of a graph change if a new edge is added?

3. (CLRS 22.5) Professor Bacon claims that Kosaraju's algorithm would be simpler if it used the original (instead of the transpose) graph in the second depth-first search and scanned the vertices in order of increasing finishing times. Does this simpler algorithm always produce correct results?

Solution

1. False, consider the first vertex the DFS visits in s. Consider a path from that vertex within s that only has edges to other vertices in that SCC. If DFS takes this path before taking an edge out of s, the vertices on the path will finish first. Since the SCC graph is a DAG, we will never revisit s if we take an edge out. It is true though that at least one vertex must have a larger finish time than those SCCs reachable from s.

2. Consider a new directed edge (u, v). We have two cases. Either u and v are in the same component, in which case the total number of components does not change and we are done; or u and v are in different components. Let u and v be in components C_u and C_v respectively. Consider the component graph. If $C_u \rightsquigarrow C_v$, then (u, v) does not change the total number of components, since it is redundant. But if instead $C_v \rightsquigarrow C_u$, then via (u, v) we have $C_u \rightsquigarrow C_v$. Thus, all components reachable with a path starting at C_u and ending at C_v (including C_u and C_v) are contracted into a single component.

3. No, consider the first connected connected component having the vertex with the smallest finish time (see first true/false). Then a DFS would start from this vertex and discover the whole graph, declaring it incorrectly as a single connected component.