Solution Set

CIS 1210—Data Structures and Algorithms—Fall 2024

Topological Sort & Strongly Connected Components—Tuesday, October 22/Wednesday, October 23

Readings

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

Review: Topological Sort

A topological sort of a directed acyclic graph (DAG) $G = (V, E)$ is an ordering of the vertices such that for each directed edge $(u, v) \in E$, u appears before v in the ordering. As described in the below algorithms, topologically sorting a DAG only takes $O(m + n)$ time, so **given a DAG, it is helpful to topologically sort it,** since most graph algorithms take $\Omega(m + n)$ time anyway. In other words, topologically sorting a DAG is usually a free step, and if not necessary for your algorithm, it can make reasoning/thinking about the problem easier since it gives you a visual. Below are two algorithms to find a topological sort:

Kahn's Algorithm

Every DAG has a source node, or a node with no incoming edges. Kahn's algorithm relies on this intuition — at a high-level, the algorithm operates by repeatedly finding a source node, putting it next in the topological sort, removing the node and all of the edges incident on it from the graph, and repeating this process.

Kahn's algorithm runs in $O(m+n)$ time. As seen in the pseudocode, the first step to compute the in-degree of each node takes $O(m + n)$ time since for each node, we scan through its neighbors; the second step to populate the queue takes $O(n)$ time since we iterate through all of the vertices. In our while loop, note that we enqueue each node exactly once and scan through each of its neighbors, performing constant work for each, which takes $O(m + n)$ time.

Tarjan's Algorithm

Tarjan's algorithm leverages the finishing times of DFS as shown in the pseudocode by just running DFS and then returning the nodes in decreasing order of finishing times. Thus, it also runs in $O(m + n)$ time.

Review: Kosaraju's Algorithm

Given a directed graph $G = (V, E)$, a **strongly connected component (SCC)** is a maximal set $S \subseteq V$ such that for all $u, v \in S$, there exists a path $u \rightsquigarrow v$ and a path $v \rightsquigarrow u$. Thus, we can decompose a directed graph G into its SCCs, yielding G^{SCC} or our kernel graph. Formally, $G^{SCC} = (V^{SCC}, E^{SCC})$. Each vertex v_i in G^{SCC} represents a single SCC C_i in G, and an edge (v_i, v_j) exists in G^{SCC} if G contains the directed edge (x, y) where x is in SCC C_i and y is in SCC C_j. Observe that G^{SCC} is a DAG, meaning that we can topologically sort it to make the problem easier to think about.

Kosaraju's algorithm is an algorithm that we can use to compute the SCCs of a graph, and by extension, to obtain G^{SCC}. It operates by running two DFS traversals, one on G and another on G^T, the transposed graph obtained by reversing the direction of edges in G; in the latter, we consider vertices in order of decreasing finishing times.

Thus, Kosaraju's runs in $O(m+n)$ time. As seen in the pseudocode, the first step is just DFS, which takes $O(m + n)$ time; the second step is computing G^T, but this can be done in $O(m + n)$ time since we just reverse the direction of edges; and the third step is also just DFS, which takes $O(m + n)$ time.

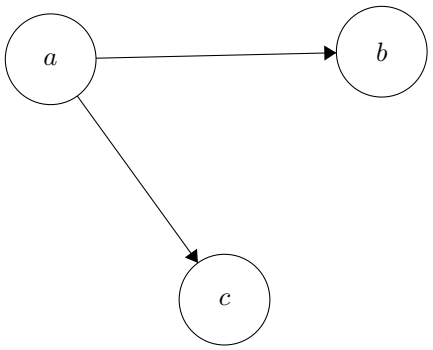
Problems

Problem 1: True or False

1. Every DAG has exactly one topological sort.
2. If a graph has a topological sort, then a DFS traversal of the graph will not find any back edges.
3. Given a DAG G Tarjan's and Kahn's algorithm will always output the same topological ordering.

Solution

1. **False.** As a counterexample, consider a graph with no edges. Any order of its vertices is a valid topological sort.
2. **True.** If there is a topological sort, then the graph is a DAG, which by definition does not have any directed cycles. Since the presence of a back edge during a DFS traversal on a directed graph indicates a directed cycle, a DFS traversal on a graph with a topological sort will not find any back edges.
3. **False.** As a counterexample, consider the following directed graph:



Depending on how Khan's Algorithm selects vertices with in-degree 0, it could output (a, b, c). However, depending on the order of vertices processed in DFS, Tarjan's algorithm could output (a, c, b). While these are both valid topological orderings, they are not the same.

Problem 2

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

Solution

Consider a new directed edge (u, v). We have two cases. First, if u and v are already in the same SCC, the number of SCCs does not change. Otherwise, let u and v be in SCC_u and SCC_v, respectively. Consider G_{SCC}. If $SCC_u \rightsquigarrow SCC_v$ or there is no directed path between the two SCCs, then adding (u, v) will not change the number of SCCs. However, consider the case where $SCC_v \rightsquigarrow SCC_u$. Via (u, v), we have $SCC_u \rightsquigarrow SCC_v$, so all SCCs reachable with a path starting at SCC_v and ending at SCC_u (including SCC_u and SCC_v) will be contracted into a single SCC. Therefore, adding a single edge may not change the number of SCCs at all, but it could also contract the entire graph into a single SCC.

Problem 3

A graph $G = (V, E)$ is "almost strongly connected" if adding a single edge makes the graph strongly connected. Design an $O(|V| + |E|)$ algorithm to determine whether a graph is almost strongly connected.

Solution

Algorithm: Use Kosaraju's algorithm to create G_{SCC}; then, topologically sort it. Add an edge from the last SCC to the first SCC in the topological sort and check if the new graph G' is now strongly connected either by running Kosaraju's again on G' and checking if there is only one SCC or by picking an arbitary vertex v; running BFS/DFS starting at v on **both** G' and G'^T; and seeing if v can reach all other vertices in both cases. If there is only one SCC, return true; otherwise, return false.

Proof of Correctness: We want to show that our algorithm returns true iff G is almost strongly connected. First, however, we show that our last step properly checks if G' is strongly connected. If Kosaraju's yields one SCC, then, by definition, G' is strongly connected. In the alternate method, we know v has a path to all other vertices in G', and we also know that all other vertices have a path to v because we ran BFS/DFS on G'^T as well. So, for any two vertices, x, $y \in G$, there exists a path from x to y by following $x \rightsquigarrow v \rightsquigarrow y$, implying G' is strongly connected. We now turn to proving the biconditional statement:

($\Rightarrow$) If our algorithm returns true, then our new graph G' is strongly connected. Since we only added a single edge in our algorithm, by definition, it follows that our original graph G is almost strongly connected.

($\Leftarrow$) We'll prove the contrapositive: If our algorithm returns false, then our new graph G' is not strongly connected. We want to show that only the addition of an edge from the last SCC to the first SCC is necessary for checking whether G is almost strongly connected. For G to be almost strongly connected, every vertex in G must have a path to vertex s, the source of the edge to add, and a path to itself from t, the second vertex/endpoint of the new edge. If it didn't, then the new graph G' would have a vertex with no path to s and/or no path from t. Hence, t must be in the first SCC, as if it wasn't, then any vertex earlier in the topological sort is not reachable from t; similarly, s must be in the last SCC, as if it wasn't, then any vertex later in the topological sort would not be able to reach s. Therefore, adding a single edge from the last SCC to the first SCC to obtain G' and finding that G' is not strongly connected is sufficient to conclude that G is not almost strongly connected.

Runtime Analysis: Using Kosaraju's and generating G_{SCC} takes $O(|V|+|E|)$ time. Topologically sorting G_{SCC} also takes $O(|V|+|E|)$, since we can use Kahn's or Tarjan's. Checking if the resulting graph is strongly connected requires running Kosaraju's or BFS/DFS, both of which take $O(|V|+|E|)$ time. Therefore, our algorithm runs in $O(|V|+|E|)$ time.