Readings

- [Lecture Notes Chapter 18: DAGs and Topological Sort]
- [Asynchronous Recording: Proof of “Every DAG has a source node”]
- [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. In lecture, you already learned Tarjan’s Algorithm, which leverages the finishing times of DFS as shown below:

```
Tarjan’s Algorithm for Topological Ordering

Input: A DAG $G = (V, E)$, given as an adjacency list
Output: A topological ordering of the nodes in $G$.

TopeSort(G)
    Perform a DFS on G
    Return the nodes in decreasing order of finish time
```

Since we just run DFS and then return the nodes in decreasing order of finishing times, Tarjan’s algorithm runs in $O(m + n)$ time.

Another algorithm for computing the topological sort is Kahn’s algorithm, which also runs in $O(m + n)$ time as described below. Therefore, since topologically sorting a DAG only takes $O(m + n)$ time, if you are ever given a DAG, it is really helpful to topologically sort it because most graph algorithms take $\Omega(m + n)$ time anyway. In other words, topologically sorting a DAG is usually a free step, and if it is not necessary for your algorithm, it can make reasoning/thinking about the problem easier since it gives you a visual.

Kahn’s Algorithm

Observe that every DAG has a source node, or a node with no incoming edges (a full proof can be found in the course notes and in Arvind’s asynchronous recording). Kahn’s algorithm is an algorithm that relies on this intuition to compute the topological sort of a DAG. At a high-level, the algorithm operates by repeatedly finding a source node, putting it next in the topological sort, and removing the node and all of the edges incident on it from the graph. The pseudocode is below:
Runtime Analysis

Kahn’s algorithm also runs in $O(m + n)$ time, similar to Tarjan’s. 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. Therefore, Kahn’s runs in $O(m + n)$ time.

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 $u \rightarrow v$ and a path $v \rightarrow u$. 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 at least make the problem easier to think about it.

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 which is obtained by reversing the direction of edges in $G$. The pseudocode is as follows:
Runtime Analysis

Kosaraju’s runs in $O(m + n)$ time. The first step is a DFS traversal, which takes $O(m + n)$ time; the second step requires us to compute $G^T$, but this can be done in $O(m + n)$ time since we just reverse the direction of edges; and the third step is also a DFS traversal, 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. The finishing times of all vertices in a SCC $s$ must be greater than the finishing times of other SCCs reachable from $s$ during the first DFS.

Problem 2
1. How does the number of SCC’s of a graph change if a new edge is added?
2. (Adapted from CLRS 22.5) Consider a “simpler” version of Kosaraju’s algorithm, where we use the original (instead of the transposed) graph in the second DFS traversal but process vertices in order of increasing finishing times. Is this algorithm always correct?

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.