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 \( G \).

\[
\text{TopoSort}(G)  
\begin{align*}  
\text{Perform a DFS on } G 
\end{align*}  
\text{Return the nodes in decreasing order of finish time}
\]

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:
Kahn’s Algorithm for Topological Ordering

*Input:* A DAG $G = (V, E)$, given as an adjacency list

*Output:* A topological ordering of the nodes in $G$.

**TopoSort(G):**

1. Let $L$ be a new queue // nodes of in-degree 0
2. Let $OUT$ be a new list // topological ordering
3. // compute in-degree of each node
4. for each $v \in V$
5.   for each $u \in Adj[v]$
6.     $in[u] = in[u] + 1$
7. // initially populate $L$
8. for each $v \in V$
9.   if $in[v] = 0$ then
10.      $L$.enqueue($v$)
11. while $L$ is not empty do
12.   $v = L$.dequeue()
13.   $OUT$.append($v$)
14.   for each $u \in Adj[v]$
15.     $in[u] = in[u] - 1$
16.     if $in[u] = 0$ then
17.        $L$.enqueue($u$)
18. return $OUT$

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:
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.

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.** Consider the counterexample below. On the left, a DFS traversal of $G$ starting at $x$ and processing the $(x, z)$ edge first yields the starting/finishing times as annotated. On the right in $G_{SCC}$, we can see that $x$ and $z$ form one SCC and $y$ forms the other with a $(xz, y)$ edge. Observe that the finishing time of $z$ is 3, which is not greater than that finishing time of an SCC reachable from it, since $y$ has a finishing time of 5. However, it is true that at least one vertex must have a larger finishing time than those of the SCCs reachable from $s$ due to the recursive nature of 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?

**Solution**

1. 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\(_u\) \(\rightarrow\) 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\(_v\) \(\rightarrow\) SCC\(_u\). Via \((u, v)\), we have SCC\(_u\) \(\rightarrow\) SCC\(_v\), so all SCCs reachable with a path starting at SCC\(_u\) and ending at SCC\(_v\) (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.

2. Nope, this algorithm will not always be correct. Again, consider the counterexample above from Problem 1. If in the first DFS traversal of \(G\), we start at \(x\) and process the \((x, z)\) edge first, then in the second DFS traversal, processing the vertices in order of increasing finishing times in \(G\) will start at the source SCC, \(xz\) and also discover the sink SCC, \(y\) all at once, outputting both in the same DFS tree — incorrectly declaring the graph as just one SCC.

At a high-level, the intuition behind why this simpler algorithm fails is because visiting vertices in \(G\) in order of increasing finishing times in the second DFS does not always correspond to visiting the SCCs in order of increasing finishing times — meaning that we cannot guarantee we are visiting SCCs of \(G\) in reverse topologically sorted order, leading to incorrect outputs. More specifically, this is because the finishing time of a vertex (in the first DFS) being minimum does not imply that the finishing time of its SCC will be minimum. In contrast, by definition, the finishing time of a vertex (in the first DFS) being maximum does imply that the finishing time of its SCC will be maximum. This is why Kosaraju’s works: transposing \(G\) and visiting vertices in decreasing order of finishing times (aka starting from the maximum) in the second DFS guarantees that we are visiting the SCCs of \(G^T\) in reverse topologically sorted order.

**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 by running Kosaraju’s again on \(G’\) and checking if there is only one SCC or by picking an arbitrary 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 \(x \rightarrow v \rightarrow y\). This thus implies that \(G’\) is strongly connected.
(⇒) If our algorithm returns true, then our new graph $G'$ was strongly connected. Since we only added a single edge in our algorithm, by definition, it follows that our original graph $G$ was almost strongly connected.

(⇐) If our algorithm returns false, then our new graph $G'$ was 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 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.