Learning Goals

During this lab, you will:

- Review Topological Sort
- Gain intuition for finding strongly connected components through topological ordering.
- Explore applications of strongly connected components.

Topological Sort

A few weeks ago we covered an algorithm called topological sort. This is motivated by many problems encountered in the real world. For example, you are running an assembly line where there are a number of tasks required to create a product. Some of the tasks must come before others. You want to maximize the amount of parallel tasks you can complete at once. How can you obtain an ordering of these tasks to make sure the product is assembled properly? The answer is a topological sort!

**Definition 1** (Topological ordering). A topological ordering of a directed acyclic graph $G = (V, E)$ is a linear ordering of $V$ such that whenever $G$ contains a directed edge $(u, v)$, then $u$ appears before $v$ in the ordering.

There are two canonical algorithms for this. It is good for you to understand both of them.

**Using depth-first search (Tarjan’s algorithm)**

- Call DFS and compute finish times for each vertex $v$.
- As each vertex finishes, push each onto a stack.
- Return the stack.

From most recently pushed to the eldest element, the stack contains the nodes in order of decreasing finishing times.

This is equivalent to a reverse postorder traversal.

You should think carefully about the correctness of this algorithm!

**Kahn’s algorithm**

- Maintain a set $S$ of nodes with in-degree 0.
- While $S$ is not empty, remove a node from $S$ and add to the end of ordering.
- Remove all edges going out of that node and update $S$ accordingly.

This is perhaps the more intuitive algorithm based on your understanding of topo sort.
Problems

Problem 1
Conceptual questions:

1. (True/False) Every DAG has exactly one topological ordering.
   Solution. False. Take for example, a graph with no edges. Any ordering is valid.

2. (True/False) A preorder traversal always produces a topological ordering on a tree.
   Solution. True. Recall that in a tree, every vertex has exactly one path to every other vertex.

3. If a graph has a topological ordering, then a depth-first traversal of the same graph will not see any back edges.
   Solution. 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

Problem (CLRS 22.4-2). Give a linear-time algorithm that takes as input a directed acyclic graph \( G = (V, E) \) and two vertices \( s \) and \( t \), and returns the number of simple paths from \( s \) to \( t \) in \( G \). You only need count the simple paths, not list them. (An example can be found in the textbook.)

Solution. First, we see it is a DAG, so we should immediately think of topological sort. In this case, they ask for linear time, so we know that asymptotically this is fine. We can now reason about the graph in a more reasonable way.

We make the observation that the number of paths from \( s \) to \( t \) can be counted by using intermediate nodes. For each \( u \) that has an edge \( e = (u, t) \), we can count the paths to \( t \) as the sum of the number of paths to each of the \( u \) nodes. We know this because we ended up at each of those \( u \) nodes by some number of paths, then took the last edge \( e \) to get to \( t \). Therefore, we only need to consider how we got to \( u \).

From this observation, we can now build an algorithm.

```
function pathCount(G):
    Topologically sort the vertices, v_1 . . . v_n
    return pathCount(v_n, 0)

function pathCountHelper(v, accumulator):
    for incoming edge e = (u, v)
        accumulator += pathCountHelper(u, accumulator)
    return accumulator
```

Now we look at this algorithm and you should be able to reason that the running time is not optimal! We are doing a lot of overlapping work on the recursive call. It seems very likely that we will be rerunning the same recursive call multiple times (for all nodes that have edges from that node), so let’s try to eliminate doing that work again.

```
function pathCount(G):
    Topologically sort the vertices, v_1 . . . v_n
    arr = new array of size n
    arr[0] = 1
    for each i from 1 to n-1
        for each e = (v_k, v_i)
            arr[i] += arr[k]
```

This works going from ‘left to right’ on the topological ordering, counting up the paths based on the observation we made. Note that you can also go the other direction—can you figure that out?
Strongly Connected Components

**Definition 2** (Strongly connected component). 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 $v \rightarrow u$.

Note: We consider only directed graphs here because in undirected graphs, every connected component is trivially strongly connected.

**Definition 3** (Component graph (kernel graph)). The strongly connected component graph of a directed graph $G$ is the directed graph $G' = (V', E')$ where each vertex of $V'$ represents a strongly connected component of $G$, and the new edges $E'$ consist of the directed edges between the SCCs of $G$.

In other words, we can contract every edge whose incident vertices are in the same SCC to produce the component graph.

Notice that the component graph is a directed, acyclic graph. This is useful because we can now we can topologically order its vertices. This idea is crucial to many linear time graph algorithms.

Kosaraju’s algorithm

**Description**

Run DFS on $G$, noting finish times. Then, in decreasing order of finish time, run DFS on the vertices of $G^T$ ($G$ with its edges reversed). The output is a DFS forest where each tree in the forest is an SCC of $G$.

**Running time**

DFS takes $O(n + m)$ time. We perform it twice, for a total of $O(2(n + m)) = O(n + m)$. Computing $G^T$ requires simply iterating over $G$’s adjacency list once, $O(n + m)$ time. Thus, the total running time is $O(n + m)$.

**Correctness sketch**

In the DFS of $G$, after we visit a node $x$, we visit its SCC $C$ and some edges out of $C$. We observe that if there is a path from $x \rightarrow u$ in $G$, then $u$ and $x$ are strongly connected only if there is also a path from $x \rightarrow u$ in $G^T$. Because $G$ and $G^T$ have the same strongly connected components, there will be a path in $G^T$ from $x$ to every vertex in $C$ but the edges out of $C$ will have been reversed and they will not be followed before the algorithm finishes processing $C$. When $C$ is finished, the part of the DFS starting from the vertex with the next highest finish time will, by logic similar to the above, only reach vertices in its own component. Continuing to apply this logic, we see that the output of the algorithm is a forest of DFS trees, each of which is a strongly connected component of $G$.

**Problems**

**Problem 1**

Conceptual questions:

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.

   *Solution*. 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 on the path. 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 $s$. 

3
2. (True/False) Back edges are never encountered when performing the depth-first search subroutines of Kosaraju’s Algorithm.

Solution. False. You can have cycles.

3. If a graph has a topological ordering, then a depth-first traversal of the same graph will not see any back edges.

Solution. True. If we are able to topologically sort it, then it must be a DAG. That means there will not be any back edges.

4. How does the number of SCCs of a graph change if a new edge is added?

Solution. 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 \(C_u \rightarrow C_v\), then \((u, v)\) does not change the total number of components, since it is redundant. But if instead \(C_v \rightarrow C_u\), then via \((u, v)\) we have \(C_u \rightarrow C_v\). 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.

5. (CLRS 22.5) Professor Bacon claims that the algorithm for strongly connected components 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. 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.

Problem 1

Problem. 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. 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.

\(s\) 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 \(s\).

\(t\) 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 \(t\).

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 $O(|V| + |E|)$. 

\[\square\]