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.
2. (True/False) A preorder traversal always produces a topological ordering on a tree.
3. If a graph has a topological ordering, then a depth-first traversal of the same graph will not see 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.)

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 \( u \leadsto v \) and a path \( v \leadsto u \).

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 \( x \leadsto u \) in \( G \), then \( u \) and \( x \) are strongly connected only if there is also a path from \( x \leadsto u \) 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.

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

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

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

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?

**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.