Learning Goals

During this lab, you will:

• 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.

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 $u \rightsquigarrow v$ and a path $v \rightsquigarrow u$.

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 tree, each of which is a strongly connected component of $G$.

**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. (True/False) Back edges are never encountered when performing the depth-first search subroutines of Kosaraju’s Algorithm.
   
   *Solution*. False. You can have cycles.

4. 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.
5. 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 \(C_u \leadsto C_v\), then \((u, v)\) does not change the total number of components, since it is redundant. But if instead \(C_v \leadsto C_u\), then via \((u, v)\) we have \(C_u \leadsto C_v\). 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.

Problem 2

[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.

```python
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.

```python
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?

Problem 3

[CLRS 22-4] Let \(G = (V, E)\) be a in which each vertex \(u \in V\) is labeled with a unique integer \(L(u)\) from the set \(\{1, 2, ..., n\}\). For each \(u \in V\), let \(R(u) = \{v \in V \mid u \leadsto v\}\) be the set of vertices reachable from \(u\). Define \(\text{min}(u)\) to be the vertex in \(R(u)\) whose label is minimum. Give an \(O(|V| + |E|)\) algorithm to compute \(\text{min}(u)\) for each \(u \in V\). (Hint: How would you do it if it were a DAG?)
Solution. If it was a DAG, we start by doing a topological sort. This should make a lot of sense since we mention the reachable set from each node. Now we process the nodes in order from ‘right to left’, meaning we start from the node with out-degree 0. Initialize our $\min[u] = u$ for all $u$ in $V$. As we move along the ordering, we update $\min[u] = \min \min[v], \min[u]$ for all $e = (u, v)$ leaving vertex $u$.

Now we consider how it works if it has cycles. We know that we just solved it for the acyclic case, so how can we transform our graph to be able to use that algorithm? Our favorite Kosaraju’s algorithm! The SCC kernel graph is a DAG. However, we need to make some small modifications to make it work. Now the label of each SCC will be the $\min(v)$ for all $v$ in the component.

Correctness. The algorithm for a DAG works because the topological ordering guarantees that at the time we process a node $u$, we have already processed all the nodes reachable from $u$. Each neighbor $v$ already encapsulates all the additional data of nodes reachable from $v$.

When converting to using the SCC graph, we know that we can label it in this way because all nodes in the SCC are reachable from each other. This comes straight from the definition of a SCC. Therefore, we can label it with the min since it is guaranteed to be in the reachable set of all vertices in that SCC.

Running time. Steps:

1. Topological sort: $O(V + E)$
2. Processing the vertices and setting mins: $O(V + E)$
3. Creating SCC graph: $O(V + E)$
4. Labelling all SCCs with min of component: $O(V)$

Therefore, this algorithm is $O(V + E)$. 