How can we make this faster?

Notice in every step, we update indegrees + look for node with indegree = 0.

When a node has indegree = 0, store it to be next node to be deleted. Store in queue.

TopSort (G) // G=(V,E). Prints nodes of G in top-sort order.
1. For all v ∈ V
2.   if (v.indegree == 0)
3.     Q.enqueue (v)
4. while (!Q.empty())
5.   u = Q.dequeue()
6.   print (u)
7.   //update indegrees of neighbors
8.   for all neighbors, v, of u
9.     v.indegree --
10.    if (v.indegree == 0)
11.       Q.enqueue (v).

```
0 3 2 1 0
A  B  C  D  E
```

printed: A B E F G D C

```
0 3 2 1 0
A B C D E
```

printed: A B E F G D C
RunTime:

Steps 1-3: \( O(V^3) \)
Steps 4-10: \( O(V^2) \) + (not *)
Step 7: \( O(E+V) \)

Consider each vertex linear times, each edge once: \( O(V^2 + E) \)

Recall: runtimes for Dijkstra's: \( O(V \log V + E \log V) \) & BF \( O(V^2 + E) \)

Can we do better for DAGs?

For DAGs, top sort can help us find shortest paths. How?

Just put nodes in top sort order.

For Shortest paths in DAGs:

Proceed nodes in top sort order (no MinHeap needed)

Why? For a node \( \alpha \), only nodes that appear before \( \alpha \) in top sort order can affect \( \alpha \).dist

* Does this work for graphs with negative edge weights? Yes!

Recall: problem with negative edges \( + \text{Dijkstra's } \): future node could help improve distance to already processed node

If nodes processed in top sort order, a negative edge cannot improve the distance of a processed node (only unprocessed nodes).
TopSort - SP (G, s) // G is DAG : (V, E), s ∈ V.

// Finds shortest path distances from s to all other nodes.
1. Q = TopSort(V, E) // Store vertices in top-sort order
2. For all v ∈ V, v.dist = ∞, v.pred = null. (No more processed)
3. s.dist = 0
4. While (!Q.empty())
5. u = Q.dequeue()
6. For each edge (u, v)
7. Update (u, v) = if (v.dist > u.dist + w_{u,v})

Q: r s t x y

r is "thrown" away (can't get to it):
Run Time:
Step 1 (Top-Sort): $O(V + E)$
Steps 2-3: $O(V)$
Steps 4-7: $O(V + (\text{not } *) O(E)$

Total: $O(2(V + E)) = O(V + E)$

vs. $O(V\log V + 1E\log V)$ for Dijkstra's, $O((V + E)E)$ for BFS.

Correctness: Similar to Dijkstra's correctness proof.

When node $u$ is dequeued, $u\text{-dist}$ is optimal distance to $u$.

1. Only edges $(v,u)$ where $v$ comes before $u$ in the topsort order can improve the distance to $u$.
2. Algorithm finds optimal distance to all such nodes before setting distance for $u$.
   - Distance to $u$ must be optimal.

Why processed not needed:

Cyclic:
- $u$ most recently deleted
- Acyclic (nodes deleted in topsort order)
  - $u$ most recently deleted

edge $(u,v)$ exists where
- All nodes adjacent to $u$ will have dist > $u\text{-dist}$
- $v\text{-dist} < u\text{-dist}$ so don't update $v$.
Minimum Spanning Tree

Suppose we have set of cities, roads.

- Big snowstorm hits
- Can't afford to have all roads open but need access b/w every pair of cities
- Roads have costs

Goal: find set of roads of minimum total cost s.t. there is a path b/w every pair of cities.

What can we say about this set?
⇒ will make up a tree ⇒ No cycles!

Why?

All three edges cannot be in minimum cost set since any two allows access to all three nodes.
Minimum Spanning Tree (MST)

\[ \text{G} = (V, E) \]
\[ V = \{ v_1, v_2, \ldots, v_n \} \]
\[ E = \{ (v_i, v_j) : v_i, v_j \in V \} \]

For \( (v_i, v_j) \in E \), \( c(v_i, v_j) > 0 \) is cost of \( (v_i, v_j) \)

Goal: Find a subset of edges \( T \subseteq E \) such that \( (V, T) \) is connected and total cost \[ \sum_{(v_i, v_j) \in T} c(v_i, v_j) \]

is minimized.

Two Greedy Approaches work:

1. Start with any node, continuously choose the cheapest node to add to current connected graph
2. Continuously choose the cheapest edge that doesn't cause a cycle

Start with (1) or how to implement?