CS170 Discussion Section 4: *2/8*

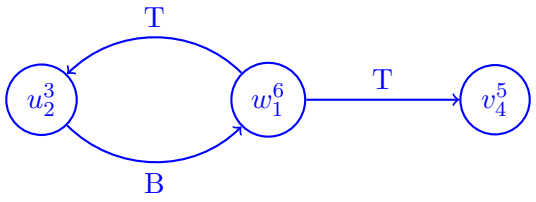
1. *Short answer.* For each of the following, either prove the statement is true or give a counterexample to show it is false.

   (a) If (u, v) is an edge in an undirected graph and during DFS, $\text{post}(v) < \text{post}(u)$, then u is an ancestor of v in the DFS tree.

   True. There are two possible cases: $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$ or $\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)$. In the first case, u is an ancestor of v. In the second case, v was popped off the stack without looking at u. However, since there is an edge between them and we look at all neighbors of v, this cannot happen.

   (b) In a directed graph, if there is a path from u to v and $\text{pre}(u) < \text{pre}(v)$ then u is an ancestor of v in the DFS tree.

   False. Consider the following case:



   (c) In any connected undirected graph G there is a vertex whose removal leaves G connected.

   True. Remove a leaf of a DFS tree of the graph.

2. *Alien alphabet.*
   Suppose you have a dictionary of an alien language which lists words in some sorted lexicographical ordering. For example, given the following list of words:

$$[\texttt{baa} \quad \texttt{abcd} \quad \texttt{abca} \quad \texttt{cab} \quad \texttt{cad}]$$

   You can conclude the ordering of the alphabet is

$$\texttt{b} < \texttt{d} < \texttt{a} < \texttt{c}$$

   Give an efficient algorithm to determine the lexicographical ordering for any input list.

   Begin by setting up a graph with one node for each symbol in the alphabet.

   Then walk through the list of words linearly and consider each adjacent pair (u, v). Look through the characters in (u, v) one by one until you find the first index i in which they differ. Then draw an edge from symbol $u[i]$ to $v[i]$, which denotes that $u[i]$ is before $v[i]$ in this ordering.

After drawing in the edges, linearize the dag and you have a topological ordering.

The overall runtime is linear in the number of words given and the size of the alphabet; $\Theta(|W| + |A|)$.

3. *True Source.* Design an efficient algorithm that given a directed graph G determines whether there is a single vertex v from which every other vertex can be reached. Hint: first solve this for directed acyclic graphs. Note that running DFS from every single vertex is not efficient.

   In directed acyclic graphs, this is easy to check. We just need to see if the number of source nodes (zero indegree) is 1 or more than 1. Certainly if it is more than 1, there is no true source, because one cannot reach either source from the other. But if there is only 1, that source can reach every other vertex, because if v is any other vertex, if we keep taking one of the incoming edges, starting at v, we have to either reach the source, or see a repeat vertex. But the fact that the graph is acyclic means that we can't see a repeat vertex, so we have to reach the source. This means that the source can reach any vertex in the graph.

   Now for general graphs, we first form the SCCs, and the metagraph. Now if there is only one source SCC, any vertex from it can reach any other vertex in the graph, but if there are more than one source SCCs, there is no single vertex that can reach all vertices.

4. *Shortest Cycle.*
   Give an algorithm that takes as input an undirected, unweighted graph, and returns the length of the shortest cycle in the graph (if the graph is acyclic, it should say so). Your algorithm should take time at most $O(|V|^2 + |V| \cdot |E|)$.

   Consider the simpler problem of finding the shortest cycle through vertex s in a graph. We can solve it with a version of BFS (or DFS) starting from s and modified as follows: when exploring a vertex v and checking edges, if the edge (v, w) is encountered and w *has* been explored, instead of skipping it, add the cycle $s..v, w..s$ as a possible cycle. At the end of the BFS exploration, return the shortest of the candidate cycles.

   We can be sure this returns the length of the shortest cycle containing s because all cycles from s take the form of some $s - v$ path, plus some $w - s$ path (where w can be equal to s in which case this path has length 0), plus the edge (v, w), so it considers all the cycles. We know our `depth`(v) and `depth`(w) values will be correct, so the length it computes is correct.

   The complete algorithm is to run this modified BFS from each vertex in the graph, and return the minimum of all the results (the shortest cycle must go through some vertex, and that vertex was one of those we checked). This runs in $\Theta(|V|(|V| + |E|)) = \Theta(|V|^2 + |V| \cdot |E|)$ time.

5. *Dijkstra and negative edge weights.*
In general, Dijkstra's algorithm doesn't work on graphs with negative edge weights. Here is one attempt to fix it:

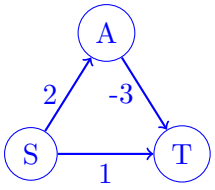
   (a) Add a large number M to every edge so that there are no negative weights left.

   (b) Run Dijkstra to find the shortest path in the new graph.

   (c) Return the path Dijkstra's algorithm found, but with the old edge weights (subtract M from the weight of each edge).

   Show that this algorithm doesn't work — even when the graph is guaranteed not to contain a negative weight cycle — by finding such a graph for which the algorithm must give the wrong answer. What is your intuition for why this algorithm does not work?

   The intuition here is that this scheme increases the length of a path containing more edges by a larger amount. Thus, in our counterexample, we would like to have a shortest path that has many more edges than some other path of larger weight. Building on this, we come up with the following counterexample:



   The shortest $S \to T$ path is $S \to A \to T$ (weight -1), but after we add $M = 3$ to each edge, the shortest path changes to $S \to T$ (weight 4, while that of the original one becomes 5), so we end up returning the wrong path.