CSE 101: Homework 2

1. For a given G, let's consider decomposing it into it's strongly-connected components (SCCs). Let G' be the meta-graph of G, meaning it has a vertex for every SCC of G and a directed edge $e = (C_1, C_2)$ if it's possible to travel from C_1 to C_2. We know that G' must be a DAG since it represents the SCCs of G. Also, note that if there exists some s in G from which every other vertex is reachable, then s must be part of a SCC that is a source in G'. For the sake of contradiction, consider if s were not part of a source SCC in G'. Then, since every DAG must have at least one source, s cannot reach any nodes in the source SCC of G' because it has no incoming edges! Therefore, if an s exists in G, it must be part of some source in G. The second thing to note from this argument is that if G' has more than one source, then no s will exist for the same reasoning as above. So, to test if an s exists, we simply test to see that exactly one source exists in the meta-graph G', and that when you run DFS from this source, all other SCCs are reached. Then, there must exist an s in this source SCC that can reach all other nodes in G. Here is the pseudo-code:

   ```
   SeeAll(G):
     (a) Run SCC algorithm to create $G'$.
     (b) Check that there is exactly one source in $G'$, call it $C_s$. If there are more than one source, return "No s can reach all other vertices" and halt.
     (c) Run explore at $C_s$ in $G'$. If all other vertices of $G'$ are visited return "Yes an s exists!", otherwise return "No s can reach all other vertices."
   ```

   Step 1) takes $O(|V| + |E|)$, 2) takes at worst $O(|E|)$, and 3) takes at worst $O(|V| + |E|)$. This gives a total running time of $O(|V| + |E|)$. The correctness claim is that SeeAll() returns s iff all other vertices in G can be reached from s. The correctness of both directions was established in the arguments above.

2. Let's say that we topologically sort G so that we have an ordering $n(v)$ such that edges only point towards higher ordering numbers when sorted this way. Since edges only point down the list, if a path exists that touches every node exactly once, it must start at the node with the lowest $n(v)$, call it s. This is because if it didn't start at s, then there is no path that can start at some other node and come back to s in G. The path then must go to the node who has the next largest $n(v)$, because if it didn't there would be no way to get back to this node and include it on the path. Here is the pseudocode:

   ```
   TouchAll(G):
     (a) Topologically sort G, giving an ordering $n(v) \in \{1, ..., |V|\}$.
     (b) For i from 1 to $|V|-1$: Let u be the node that has $n(u) = i$, and v be the one that has $n(v) = i+1$.
         If $(u, v) \notin E$ output "No path!" and halt.
     (c) Output "There is a path!".
   ```

   The runtime of this is dominated by TopSort call which gives a total runtime of $O(|V| + |E|)$. The correctness claim is that TouchAll output that there is a path iff there is a path that touches all nodes. Assume that TouchAll outputs that there is a path, then each (u, v) that it checks in 2) is part of the edge set, so there is a path that touches every vertex. For the other direction of the proof, assume there is a path that touches every vertex exactly once, then it must be revealed from the topological sorting because of the reasons stated in the first paragraph.

3. Here's what the dist vector looks like before each pop from the priority queue:

popped	A	B	C	D	E	F	G	H
-	0	∞	∞	∞	∞	∞	∞	∞
A	0	1	∞	∞	4	8	∞	∞
B	0	1	3	∞	4	7	7	∞
C	0	1	3	4	4	7	5	∞
D	0	1	3	4	4	7	5	8
E	0	1	3	4	4	7	5	8
G	0	1	3	4	4	6	5	6
F	0	1	3	4	4	6	5	6

The shortest path tree has edges as follows $A \to B \to C \to D \to G \to H$, and $G \to F$, and $A \to E$.

4. Consider running BFS staring at u, and say that v is reached on level k (if it's not reached at all then there are 0 SPs between u and v). So all SPs between u and v must start at u on level 0, then go somewhere on level 1, then level 2, and continue as so until they reach v on level k. Consider all edges that go from level $k-1$ to v on level k. All (u,v)-SPs must use one of these edges. Let $w_1, ..., w_i$ be the vertices on level $k-1$ that have edges that connect to v. So, the number of SPs from u to v is going to be the sum over i of the number of SPs from u to w_i (each SP from u to w_i can be extended by edge (w_i, v) to be a SP from u to v). This suggests a recursive formulation of this problem. Let $nsp(x)$ be the number of shortest paths from u to node x. We know $nsp(x) = \sum_i nsp(w_i)$ where the sum is over all w_i such that $(w_i, x) \in E$. We know that there is exactly one shortest path from u to itself, namely the path that goes nowhere, so $nsp(u) = 1$. We can update the vector nsp as we run BFS. For each tree-edge (x, y) encountered during the BFS process, update $nsp(y) += nsp(x)$. Here's the pseudocode:

```
NumSPs(G,u,v):
  (a) For all $x \in V$ initialize $nsp(x) = 0$. Set $nsp(u) = 1$.
  (b) Run BFS on G starting at $u$, and for each tree-edge $(x, y)$ encountered during the process do the
      update $nsp(y) = nsp(y) + nsp(x)$.
  (c) Return $nsp(v)$.
```

The runtime here is dominated by BFS which is $O(|V| + |E|)$. For the correctness it suffices to prove the sub-claim that $nsp(x)$ will hold the number of shortest paths from u to x after NumSPs is executed. Let's prove this by induction. As a base case, the number of shortest paths from u to itself is 1, which is what we set it to in step 1. Now assume that $nsp(x)$ is the number of shortest paths from u to x for all x below level k of the BFS. We now would like to show that we correctly set $nsp(x)$ for an x on level k of the BFS. Each edge (w_i, x) where w_i is on level $k-1$ must be part of the SPs to x, and each SP from u to w_i can be extended by edge (w_i, x) to be a SP to x. This justifies the sum in step 2. The overall correctness claim that NumSPs outputs K iff the number of shortest paths from u to v is exactly K is shown because of the sub-claim.

5. Let the edge in question be $e = (u, v)$. If we run Dijkstra's algorithm from both u and v, we will have calculated as a result two vectors $\delta_u(x)$, which holds the lengths of the shortest paths from u to every other vertex, and $\delta_v(x)$ which is the same but from v. Since the graph is undirected the SP from any x to y is the same length as from y to x. So, to find a cycle, we need to find an intermediate node m such that $\delta_u(m)+\delta_v(m)$ is minimized. Then, the cycle will be $u \to m \to v \to u$. Here is the pseudocode:

ShortestCycle(G,e=(u,v)):

(a) Run Dijkstra(G,u) and Dijkstra(G,v) to get $\delta_u(x)$ and $\delta_v(x)$.

(b) Return $l_e + \min\{\delta_u(m) + \delta_v(m) : m \in V,\ m \neq u,\ m \neq v\}$.

The first step takes $O(|V|^2)$ using an array implementation of the priority queue in Dijkstra. The second step takes $O(|V|)$, so we have overall an $O(|V|^2)$ algorithm. The correctness claim is that ShortestCycle returns K iff K is the length of the shortest cycle containing $e = (u, v)$ (note: a $K = \infty$ is OK if no such cycle exists). Assume that ShortestCycle returns K, meaning that there exists a m such that minimizes the expression in 2. and where there is a cycle through m using e. For the other direction, assume that G has a shortest cycle through e of length K. We must show that there must exist some m where the cycle edges are the same as the shortest path from u to m and from v to m. Assume that there was a shortest cycle where an m didn't exist as such. Then the shortest paths from u to m or from v to m could be made shorter which is a contradiction. This concludes the correctness argument.