CS 457, Data Structures and Algorithms I
Fifth Problem Set Solutions

1. Consider a variation of STRONGLY-CONNECTED-COMPONENTS(G) (see Page 617 of your textbook) which, rather than calling DFS(G^T) in Step 3, calls DFS(G) instead (everything else remains the same). Provide a graph $G = (V, E)$ for which this variation is incorrect, i.e., it does not output the correct set of strongly connected components of G. For full, credit explain what the variation of the algorithm would do for this graph G:

   (a) provide $v.d$ and $v.f$ for all $v \in V$ for Step 1 (you can choose the initial ordering of the vertices),

   Solution:

   Consider the directed graph of 2 vertices, s and t, with an edge (s, t). We select the ordering of vertices to be $[s, t]$. Thus, $s.d = 1$, $s.f = 4$, $t.d = 2$, $t.f = 3$.

   (b) provide $v.d$ and $v.f$ for all $v \in V$ for Step 3,

   Solution:

   We begin with the same vertex (s) for the second depth-first search, so the d and v values for the vertices are the same.

   (c) provide the set of connected components that it would output, and

   Solution:

   This algorithm would output that both s and t belong to the same connected component. Thus, the set of connected components is s, t.

   (d) explain why this output is incorrect.

   Solution:

   This is clearly incorrect since there exists no path from t to s. Thus, s and t lie in separate connected components.

   Further, it is easy to see that there are many graphs for which this algorithm will return the incorrect result. Consider any graph with vertices u and v such that there exists a path from u to v but no path from v to u. Suppose DFS discovers u first. Thus, since there exists a path from u to v, v will have a lower finishing time than u. Suppose v is discovered first. Since there exists no path from v to u, v will finish before u is discovered and thus v will finish before u does. Therefore, when performing the depth first search in decreasing order of finishing times, we will begin at u and find a path to v. The algorithm would then incorrectly conclude that u and v are in the same connected component.

2. Provide an algorithm that, given a directed acyclic graph $G = (V, E)$ and two vertices $s, t \in V$, returns the number of simple paths from s to t in G. Your algorithm's worst-case running time should be $O(m + n)$.

   Solution:

   We can first perform topological sort on the graph and begin a depth-first search at vertex s. For every vertex, we store an additional field which corresponds to the number of paths from that vertex to t. The number of paths from a vertex v to t is equal to the sum of the number of paths to t of all the vertices adjacent to v plus 1 if v is connected to t itself. We can write this directly as a recursive algorithm and we may stop recursion for a node if it lies "to the right of t" in a topological sort or if it has no vertices adjacent to it. Since we store the number of paths from each vertex to t, we avoid the need to recalculate (and therefore retread) edges which we have already explored. Thus, we observe that since topological sort is $O(m + n)$ and we traverse every vertex and edge at most once the total cost of operation is at most $O(m + n)$.

3. A Eulerian tour of a strongly connected directed graph $G = (V, E)$ is a directed cycle that contains *every* edge in E exactly once (at least one and no more than once). First, prove that G has a Eulerian tour *if and only if* every vertex V has its in-degree equal to its out-degree. Then, using this fact, provide a $O(m)$ algorithm that returns a Eulerian tour of G, if one exists, or reports that no Eulerian tour exists. Hint: your algorithm can merge edge-disjoint cycles.

Solution:

We want to prove that a strongly-connected digraph G has a directed Eulerian cycle if and only if each vertex has its indegree equal to its outdegree. We begin by showing that G has a directed Eulerian cycle *only if* each vertex has its indegree equal to its outdegree. Assume that a Eulerian cycle may exist in a graph G that contains some vertex v whose indegree is different than its outdegree. Starting from some other vertex u, we traverse the Eulerian cycle, and we have to visit v at some point, thus using one of its incoming edges. Then, since the cycle returns to u, we need to use one of v's outgoing edges to leave v. This remains true whenever the cycle re-visits v, so each visit uses the same number of incoming and outgoing edges. As a result, since v has a different outdegree and indegree, there has to exist some incoming or outgoing edge that the Eulerian cycle does not visit, which contradicts the definition of a Eulerian cycle, i.e., the fact that it contains each edge *exactly* once.

To complete the first part of the proof, we now also show that G has a directed Eulerian cycle *if* each vertex has its indegree equal to its outdegree. We prove this fact by constructing an Eulerian cycle. Starting from some vertex v, we arbitrarily choose one of its outgoing edges, hence visiting one of its neighbors, and repeat this process until we return to v. Since every vertex that we visit has the same indegree and outdegree, whenever we use an edge to visit that vertex, there always exists some unused outgoing edge that we can choose in order to leave that vertex, until we return to v, which has already used one of its outgoing edges without using an incoming one. Note that, of course, this cycle need not be Eulerian, and there may be several edges that the initial cycle did not use. If this is the case, then starting from a vertex u of the initial cycle that has unused edges (it must have the same number of outgoing and incoming edges), we choose one of its outgoing edges and repeat the same process, using only unused edges, until we return back to u. The initial cycle, combined with the new cycle now form a longer cycle that starts from v, uses the detour via u and then returns to v. Using the same arguments as above, we can repeat this process until we have used all the edges exactly once, thus forming a Eulerian cycle.

Having proved that a strongly connected digraph G has a directed Eulerian cycle *if and only if* each vertex has its indegree equal to its outdegree, we can now proceed to design an algorithm for this problem. Given some graph G, the algorithm begins by running the STRONGLY-CONNECTED-COMPONENTS(G) algorithm from Page 617. If this algorithm returns more than one strongly connected components, then a Eulerian cycle *cannot exist* since there is no cycle that visits more than one components. To verify this fact note that if there existed a cycle that visits two connected components, then every vertex in one component would be reachable from the other, and vice versa, which would imply that all these vertices actually belong to the same connected component. If, on the other hand, the STRONGLY-CONNECTED-COMPONENTS(G) algorithm returns a single connected component, we could go over all the edges in that component and compute the indegree and outdegree of every vertex. Using the equivalence that we proved above, if there exists a vertex with different indegree and outdegree, the algorithm reports that "a Eulerian cycle does not exist in G". If, on the other hand, all vertices have the same indegree and outdegree, the algorithm could use the constructive argument from above in order to generate a Eulerian cycle.

4. (25 pts) Provide a DFS-based algorithm that takes as input a graph $G = (V, E)$ and determines if there exists an edge $e \in E$ such that removing e from E increases the number of connected components of G. If such an edge exists, your algorithm should return all of the edges in E with this property. For instance, if $V = \{v_1, v_2, v_3\}$ and $E = \{(v_1, v_2), (v_2, v_3)\}$, then $G = (V, E)$ has a single connected component. Removing either one of the two edges in E would increase the connected components from one to two, so the algorithm should return both of these edges in this example. The worst-case running time of the algorithm should be $O(m + n)$.

   Solution:

   We call any edge whose removal increases the number of connected components a *bridge*. We begin by observing that an edge is a bridge if and only if it does not belong in a cycle. Clearly, if an edge belongs to a cycle, then its removal does not increase the number of connected components. Also, if an edge (u, v) does not belong to a cycle, then after its removal there exist no path connecting u to v, and thus the two vertices belong to different connected components, while they belonged to the same one prior to the removal.

   Using this characterization of bridges, we can focus on finding the edges of the given graph G that belong to a cycle. If we use the DFS algorithm, any non-DFS-tree edge will clearly not be a bridge, since the reason why it was not included in the tree is that it connected to a previously visited node, i.e., it formed a cycle. However, there may be several tree edges that are not bridges as well. In particular, all the tree edges that would be part of the cycle formed, if we were to introduce one of the non-tree edges, are not bridges. For instance, if we were to introduce a back-edge (u, v) connecting u to its ancestor v in the DFS tree, then all of the tree edges on the path connecting u to v are not bridges.

   If we run a DFS algorithm, identifying the non-tree edges is easy. In order to efficiently identify which of the tree edges are bridges as well, we will use the properties of DFS described at Pages 606-610. In particular, whenever DFS visits a node, it marks that node with the corresponding visit time. A tree edge (u, v), where u is the parent, is not a bridge if there exists a non-tree edge connecting the sub-tree rooted at v to one of the nodes that were visited prior to v. Therefore, it suffices to remember, for each vertex v of the tree, what is the earliest visited vertex w that one of v's descendants is connected to. If w was discovered before v, then the edge connecting v to its child is not a bridge. A simple modification of DFS that recursively computes this information for each vertex v following the DFS traversal sequence therefore solves this problem in linear time.