CSE 101 FALL 2019 Homework 2 Due: Thursday, 10/10 at 11:59pm

1. Run the SCC algorithm on the following directed graph G. When doing DFS on G^R: whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.

   $A : D, E$
   $B : D, F, J$
   $C : A$
   $D : E$
   $E : C$
   $F : H, I$
   $G : F, H$
   $H : D$
   $I : G$
   $J : B, F$

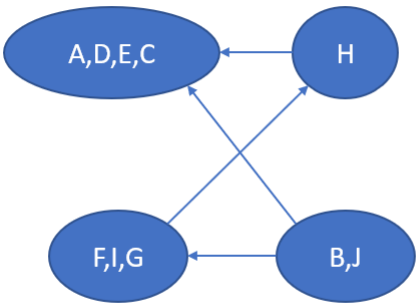
   (a) (5 points) In what order are the strongly connected components (SCCs) found?

   Solution: $A, D, E, C, H, F, I, G, B, J$

   (b) (5 points) Which are source SCCs and which are sink SCCs?

   Solution: $\{B, J\}$ is the only source, $\{A, D, C, E\}$ is the only sink.

   (c) (5 points) Draw the "metagraph" (each meta-node is an SCC of G)



2. Say you are given the dimensions for n matrices, $M_1, ..., M_n$ where M_i is a $r_i \times c_i$ matrix, and two integer dimensions R, C . You want to decide whether any product of the M_i's (possibly with repetitions, and not necessarily including all matrices) has dimensions $R \times C$. (Remember that we can only multiply a $r_1 \times c_1$ matrix with a $r_2 \times c_2$ matrix if $c_1 = r_2$. Your algorithm doesn't have to actually multiply the matrices, just decide whether such a matrix can be the result of a product of matrices in the list.

   (7 points for reasonably efficient correct algorithm (with correctness proof), 5 points for correct time analysis, and 3 points for efficiency of your algorithm.

   This problem also has a programming assignment element to it. It's worth two points. This is a trial run so it won't be graded on correctness this time.)

   Solution:

   $O(n^2)$ **Algorithm Description:** Create a graph G with a vertex for each matrix. Create a directed edge from matrix i to matrix j if $r_i = c_j$. Then create two new vertices c and r. Create a directed edge from c to all matrices that have C columns and create a directed edge to r from all matrices with R columns. Then run **explore** from vertex c. If r is visited then return TRUE. Otherwise return FALSE.

   justification: ($\leftarrow$) Suppose there is a sequence of multiplications using the matrices $M_1, \ldots, M_n$ such that the product has dimensions $C \times R$. Then since each valid sequence of multiplications

of these matrices correspond to a path in G, this particular multiplication will correspond to a path from a matrix i with C columns to a matrix j with R rows. Then there will be an edge from c to i and an edge from j to r, so along with these two extra edges and the path, there is a path from c to r and the algorithm will return TRUE.
($\rightarrow$) Suppose the algorithm returns TRUE, then there is a path from c to r in G. Vertex c points to a vertex corresponding to a matrix with C columns and vertex r is pointed to from a vertex corresponding to a matrix with R rows. Since paths in G correspond to valid sequences of matrix products, this particular path corresponds to a sequence of matrix products that has dimensions $C \times R$.
Runtime analysis: In the worst case, each matrix is square and can multiply all of the other matrices. In this case, for each pair of vertices, there will be directed edges pointing in both directions (Also, there will be a self loop on every square matrix.) Since there are n matrices, there are n^2 edges. Therefore in the worst case the runtime of this algorithm is $O(n^2)$

$O(n)$ **Algorithm Description:** Create a graph G with an edge for each matrix and a vertex for each possible dimension. Create a directed edge from i to j if there exists a matrix with dimesions (i, j).
If $C \neq R$ then run *explore* from C. If R is visited then return TRUE, otherwise return FALSE.
If $C = R$, then check all matrices to see if there is a matrix with dimensions (C, C). If so, then return TRUE, otherwise, run the SCC algorithm on the matrix. If vertex C belongs to an SCC by itself then return FALSE, otherwise return TRUE.
justification:
Case 1: Suppose $C \neq R$.
($\leftarrow$) Suppose there is a sequence of matrices $(m_1, \ldots, m_k)$ such that their product after multiplying in this order has dimensions $C \times R$. Then since there is an edge from m_i to m_{i+1}, this corresponds to a path from C to R. Therefore the algorithm will output TRUE.
($\rightarrow$) Suppose the algorithm returns TRUE, then there is a path from C to R in G, i.e., $(C, x_1, \ldots, x_k, R)$ then there exists matrices in the input with dimensions $(C, x_1), (x_k, R)$, and (x_i, x_{i+1}). Therefore, these matrices when multiplied together will yield a product with dimensions (C, R).
Case 2: Suppose $C = R$.
($\leftarrow$) **Case 1:** There is a matrix with dimensions (C, C) in which case the algorithm will output TRUE.
Case 2: Suppose there is a sequence of matrices $(m_1, \ldots, m_k)$ such that their product after multiplying in this order has dimensions $C \times C$ with $k > 1$. Then since there is an edge from m_i to m_{i+1}, this corresponds to a path from C to C with more than one edge. Therefore C is in an SCC with at least one other vertex and the algorithm will output TRUE. ($\rightarrow$) Suppose the algorithm output TRUE. Then either there is a matrix with dimensions (C, C) which is a valid product with dimensions (C, C) or there is not a matrix with dimensions (C, C) but the SCC containing C has more than one vertex. This means there is a path $(C, x_1, \ldots, x_k, C)$ with more than one vertex which means there exist matrices with dimensions $(C, x_1), (x_k, C)$, and (x_i, x_{i+1}). Therefore, these matrices when multiplied together will yield a product with dimensions (C, C).
Runtime analysis: In order to construct this matrix efficiently, you can loop through the list of matrices and for each matrix, if it has dimensions (x, y), add y to the adjacency list of x. (if x hasn't been added to the adjacency list then create a new list. You can use hash maps to keep track of which numbers have been added to the list so far.)
This will take $O(n)$ because you are doing a constant operation per matrix.
Then your graph will have n edges and at most $2n$ vertices. Therefore *explore* and *SCC* algorithms each will have runtime $O(2n + n)$.
So, total runtime is $O(n)$.

3. Suppose you are on a road trip driving from point A to point B. There are several scenic routes you would like to drive through but they are time consuming so you only want to drive through exactly one. Design a reasonably efficient algorithm that, given a *directed* graph $G = (V, E)$, a subset $S \subseteq E$, a starting vertex A and an ending vertex B, will *determine* if there exists a path from A to B that goes through *exactly one* edge in S.

   (7 points for reasonably efficient correct algorithm (with correctness proof), 5 points for correct time analysis, and 3 points for efficiency of your algorithm.

   Solution:

   Algorithm Description: Create a graph G' by removing all of the edges in S. Run explore on G' starting at s and call the array `visited_s`. Create G'^R (the reverse of G'). Run explore on G'^R starting at t and call the array `visited_t`. Loop through the edges in S. If there is an edge $(u, v) \in S$ such that `visited_s`(u) = TRUE and `visited_t`(v) = TRUE then return TRUE. Otherwise, return FALSE.

   Justification of Correctness:

   ($\rightarrow$) Suppose that the algorithm returns TRUE. Then we found an edge $(u, v) \in S$ such that there is a path from s to u and a path from v to t. This describes a path from s to t using exactly one edge from S.

   ($\leftarrow$) Suppose that there is a path from s to t in G that uses exactly one edge from S. Call this edge (u, v). Then there must be a path from s to u in G' so `visited_s`(u) = TRUE. Then there must be a path from v to t in G' (which means there is a path from t to v in G'^R) so `visited_t`(v) is TRUE. So, then when the algorithm loops through the edges, it will find (u, v) and return TRUE.

   Runtime Analysis: Creating G' is linear time because it just requires creating a new graph with fewer edges than G. Reversing a graph also takes linear time. Running explore on G' and G'^R each take linear time. Then looping through the edges takes linear time. Therefore, since there are a constant number of linear time procedures, the entire algorithm is linear time.

4. Design a reasonably efficient algorithm that takes a *directed* graph $G = (V, E)$ and *determines* if there exists a path in G that goes through each vertex *at least* once.

   (7 points for reasonably efficient correct algorithm (with correctness proof), 5 points for correct time analysis, and 3 points for efficiency of your algorithm.

   Solution:

   Algorithm Description: Run the SCC algorithm on G. (The SCCs will be found in reverse topological ordering. Suppose that we found k SCCs.) For each $i = 1...k-1$, check to see if there is an edge that goes from a vertex in the $i+1$ SCC to the i SCC. If there is such an edge for eache $i = 1...k-1$ then return TRUE. Otherwise return FALSE.

   Justification of Correctness:

   ($\rightarrow$) Suppose that the algorithm returns TRUE. Then there is an edge between each consecutive pair of SCCs. We can build a path through all of the vertices at least once in the following way: Start with any vertex in the SCC with the highest **cc** number (Suppose this number is k.) Create a path that goes through all the vertices in the k SCC and continue the path into the $k-1$ SCC. Then have the path visit all vertices in the $k-1$ SCC and continue the path into the $k-2$ SCC. Continue the path in this way until you have gone through all vertices at least once.

   ($\leftarrow$) Suppose the algorithm returns FALSE. Then there exists some pair of consecutive SCCs ($i+1$ and i) such that there is no edge that goes from any vertex in the $i+1$ SCC to any vertex in the i SCC. Suppose by contradiction that there is a path that goes through every vertex at least once. Then this path must visit all the vertices in the $i+1$ SCC *before* the path visits the vertices in the i SCC. If the $i+1$ SCC is a sink then there is no way to get to the i SCC and the path can't exist. If the $i+1$ SCC is not a sink, then it must have an

edge going to another SCC (Let's say the j SCC.) Then $j \neq i$ which means that $j < i$ which means that, since the graph is a DAG of SCCs ordered by decreasing cc number, that there is no way to get back to i from j. Therefore the path cannot exist.

Runtime Analysis: Running the SCC algorithm will take linear time. In order to check if all consecutive pair SCCs are connected by an edge, one quick way to do this is to create an array of Booleans: $check[1, \ldots, k-1]$ that is initialized to FALSE. Then loop through each edge in G and check the **cc** number of its endpoints. If you ever find an edge (u, v) such that $component(u) = i + 1$ and $component(v) = i$ then set $check(i) =$ TRUE. Then after looping through all edges, check to see if the *check* array is all TRUE. This will also take linear time. So the whole algorithm is linear time. $O(|V| + |E|)$.