Homework 3

Due date: 1/27/2021 right before class

Problem 1

In this problem, we consider the notion of "semi-connectivity". A directed graph is semi-connected if, for all pairs of $u, v \in V$, there is a path from u to v or from v to u.

Consider a case where G is a DAG. Design an $O(|E|)$ time algorithm to test whether G is semi-connected.

0.1 Solution 1

A DAG is semi connected in a topological sort, for each i, there there is an edge (v_i, v_{i+1})

Proof: Given a DAG with topological sort $v_1, v_2, ..., v_n$: If there is no edge (v_i, v_{i+1}) for some i, then there is also no path (v_{i+1}, v_i) (because it's a topological sort of a DAG), and the graph is not semi connected.

If for every i there is an edge (v_i, v_{i+1}), then for each $i, j (i < j)$ there is a path $vi -> vi+1 -> ... -> vj-1 -> vj$, and the graph is semi connected.

Complexity will be O(E) since we are running a topological sort while keeping track of 2 node connectivity.

Problem 2

For directed graph we define the notion of "strongly connected components" (SCCs). A directed graph is strongly-connected if, for all pairs of $u, v \in V$, there is a directed path from u to v and from v to u. A strongly connected component is then a subgraph $G' = (V', E')$ where $V' \subseteq V, E' \subseteq E$, which is strongly connected within the original graph. For 2 nodes v, w the directed relation RSC of "being strongly connected" (meaning that there is a path back and forth in G from v to w and vice sersa, is an equivalence relation, i.e. it satisfies:

1. $v \ RSC \ v$ (reflexive),
2. $v \ RSC \ w \ \Rightarrow \ w \ RSC \ v$ (symmetric),
3. $v \ RSC \ w \ \wedge \ w \ RSC \ v \ \Rightarrow v \ RSC \ w$ (transitive).

An equivalence relation induces partition on the set. Consequently the notion of "componenet" applies.
We will go over SCCs in the discussion, as well as talk about an algorithm for finding strongly connected components called Kosaraju's algorithm).

Design an algorithm (using your solution for problem 1 and strongly connected components) to test semi-connectivity of a directed graph in $O(|E|)$ time. Note that you can assume functions are given to conduct topological sort and to decompose a graph into SCCs (Kosaraju's algorithm). (hint: Argue that "the graph of strongly connected component" (think of what it is!) is an acyclic graph, and then apply problem 1).

0.2 Solution 2

1. Build the SCC graph $G' = (U, E')$ such that U is a set of SCCs. $E' = \{(V_1, V_2) \mid \textit{there is } v_1 \textit{ in } V_1 \textit{ and } v_2 \textit{ in } V_2 \textit{ such that } (v_1, v_2) \textit{ is in } E\}$
2. Do topological sort on G'. Check if for every i, there is edge V_i, V_{i+1}.

Correctness proof: If the graph is semi-connected, for a pair (v_1, v_2), such that there is a path $v_1 -> ... -> v_2$ - Let V_1, V_2 be their SCCs. There is a path from V_1 to V_2, and thus also from v_1 to v_2, since all nodes in V_1 and V_2 are strongly connected.

If the algorithm yielded true, then for any two given nodes v_1, v_2 - we know they are in SCC V_1 and V_2. There is a path from V_1 to V_2 (without loss of generality), and thus also from v_1 to v_2.

Time complexity will include running Kosaraju's algorithm O(E) and topological sort on the resulting graph O(E) for a total of O(2E).

Problem 3

Given an Eulerian graph (directed or undirected, your choice)

a write a recursive algorithm to produce the trace of the Eulerian path in the graph. The trace is a not necessarily simple cycle of nodes that will be visited such that we visit every edge in the graph exactly once, and end in the same node from which we started.

b Write your algorithm as an iterative algorithm. Analyse the complexity of this version. There exists a $O(|E|)$ iterative implementation of the recursion.

0.3 Solution 3

An undirected graph has a eulerian path if and only if it is connected and all vertices except 2 have even degree. One of those 2 vertices that have an odd degree must be the start vertex, and the other one must be the end vertex.

```
Start vertex:
    case 1: an odd vertex (if there are odd vertices).
    case 2: If there are zero odd vertices, we start from any vertex.

For current vertex $u$:
    traverse all adjacent vertices of $u$ to find an edge to the next vertex $v$,

    case 1: if there is only one adjacent vertex $v$ , we immediately consider it.
    case 2: If there are more than one adjacent vertices, we consider an adjacent $v$ only if edge u-v is not a bridge.

    add u-v to the path, remove u-v from the edge list. choose v to be the next vertex to be processed.
```

Listing 1: Recursive algo

```

count number of vertices reachable from u, denoted count1. (Use DFS to count the reachable vertices.)

remove edge u-v and again count number of reachable vertices from u, denoted count2.

If count1 > count2:
    u-v is a bridge
else:
    u-v is not a bridge
```

Listing 2: Helper function: isBridge

Eulerian Path in directed graph A directed graph has an eulerian path if and only if it is connected and each vertex except 2 have the same in-degree as out-degree, and one of those 2 vertices has out-degree with one greater than in-degree (this is the start vertex), and the other vertex has in-degree with one greater than out-degree (this is the end vertex)

The idea is to keep following unused edges and removing them until we get stuck. Once we get stuck, we backtrack to the nearest vertex in our current path that has unused edges, and we repeat the process until all the edges have been used.

```
Choose any starting vertex v
follow a trail of edges from that vertex until returning to v. The tour formed in this way is a cycle, but may not cover all the vertices and edges of the initial graph.

As long as there exists a vertex u that belongs to the current tour, but that has adjacent edges not part of the tour, start another trail from u, following unused edges until returning to u, and join the tour formed in this way to the previous tour.
```

Listing 3: Recursive algo directed

```
stack St;
put start vertex in St;
until St is empty
  let V be the value at the top of St;
  if degree(V) = 0, then
    add V to the answer;
    remove V from the top of St;
  otherwise
    find any edge coming out of V;
    remove it from the graph;
    put the second end of this edge in St;
```

Listing 4: Iteratuive algo

Time complexity here involves visiting all edges exactly once. Removing visited edge from the graph (semantically) and pushing it to a stack. Overall O(E).

Problem 4

Given an undirected graph $G = (V, E)$ consider the directed relation RC between *edges* (!) in the graph: Edge e_i RC e_j if there exist a simple cycle in G that contains both e_i and e_j.
Argue that RC induces a partition of the set of edges.
Give an idea (english) of how will you go about outputting (edge sets) the components of this partition.

0.4 Solution 4

If a relation is reflexive, symmetric and transitive, then it is an equivalent relation. Each equivalence relation provides a partition of the underlying set into disjoint equivalent classes.

Think of grouping these equivalence sets as nodes, and create super-nodes. In this representation each partition will constitute a node on the graph of super-nodes.