Recitation Problems - Com S 311

Week of Apr 1^{st} - Apr 6^{th}

1. You are helping a group of ethnographers analyze some oral history data. The ethnographers interviewed members of a village and learned about the birth and death a set of n people: $P_1, P_2, \ldots, P_n$. The information is categorized as follows:

   - For some i and j, person P_i died before person P_j was born
   - For some i and j, the life spans of P_i and P_j overlapped at least partially.

   As these are word of mouth information, it becomes important to validate the data. You are tasked with writing an algorithm which takes as input the learned information (about relative ordering of birth and death of n people) and verify its validity.

   Construct a graph as follows: For all i, construct a vertex b_i to represent birth of P_i and construct another vertex d_i to represent death of P_i. Add directed edge from b_i to d_i. The edge captures the fact that for any person the birth occurs before death.

   Next, based on the information gathered if P_i died before P_j is born, then add a directed edge from d_i to b_j.

   Finally, if P_i and P_j have overlapping lifespan, then add a directed edge from b_i to d_j and another directed edge from b_j to d_i. These edges capture the fact that these persons have overlapping lifespan, i.e., P_i was born before P_j's death and similarly, P_j was born before P_i's death.

   Now, we have a graph that represents the temporal ordering of the birth and death events of all people. For the information to be valid, the graph must be a DAG (directed acyclic graph). DFS explore the entire graph and look for backedges. If no backedge exists, then the graph is a DAG and the information is valid; otherwise the information is invalid.

   Overall runtime is $O(V + E)$, where V is two times the number of persons (for each person we have two vertices) and the number of edges is bounded by the sum of the number of people and the number of information pairs.

2. Consider a directed graph $G = (V, E)$ that does not have self-loop, i.e., $(v, v) \notin E$ for all $v \in V$, a universal sink is a vertex in V such that all other vertices have an edge to the sink and the sink has no outgoing edges. Write an algorithm that takes as input adjacency matrix representation of a graph and verifies the existence of a universal sink in $O(V)$ time.

   Observation: For ease of arguments, consider that the vertices are numbered from 1 to n where n is the total number of vertices. A sink does not have any outgoing edges. Therefore, the row in the adjacency matrix corresponding to the sink must be all 0's. Similarly, all other vertices have an edge to the sink. Therefore, the column corresponding to the sink must be all 1's except the row for the sink. That is, if i is a sink

   $$\forall j \leq n : adj[i][j] = 0 \text{ and } \forall j \leq n.(j \neq i \;\Rightarrow\; adj[j][i] = 1)$$

   Strategy: We start with the first row and first column of the adjacency matrix. If $adj[1][1] = 1$ then vertex 1 cannot be universal sink because it has an outgoing edge. In that case, we move to examine the next vertex (i.e., next row) $adj[2][1]$. On the other hand, if $adj[1][1] = 0$, then vertex 1 may be an universal sink, and we proceed to examine the next column $adj[1][2]$.

   In general, if $adj[i][j] = 1$ then we proceed to $adj[i + 1][j]$ and if $adj[i][j] = 0$ then we proceed to $adj[i][j + 1]$. We proceed with the process until either row or column is equal to n. On termination, the vertex corresponding to the row **is a candidate** for being a sink. This can be justified as follows.

Consider that when the above process terminates the row is x and column is y. All vertices $< x$ are not sinks because they have an outgoing edge (note the process increments the row value when we see an outgoing edge).

Finally, we need to check the candidate vertex is indeed a sink or not. This can be verified by just iterating over the row and the column corresponding to the candidate.

Algorithm 1 (given adjancency matrix adj of $G = (V, E)$ and number of vertices n)

```
1: i = 1
2: j = 1
3:
4: while i ≤ n and j ≤ n do
5:     if adj[i][j] == 1 then
6:         i++
7:     else
8:         j++
9:
10: // check for candidate i being the sink
11: for k = 1 to n do
12:     if adj[i][k] != 0 then
13:         return false
14:
15: for k = 1 to n do
16:     if k != i and adj[k][i] != 1 then
17:         return false
18:
19: return true
```

Each loop terminates within $O(n)$ steps.

Consider solving the same problem when the graph is represented using adjacency list and discuss the runtime of your solution.