Bryan Arnold

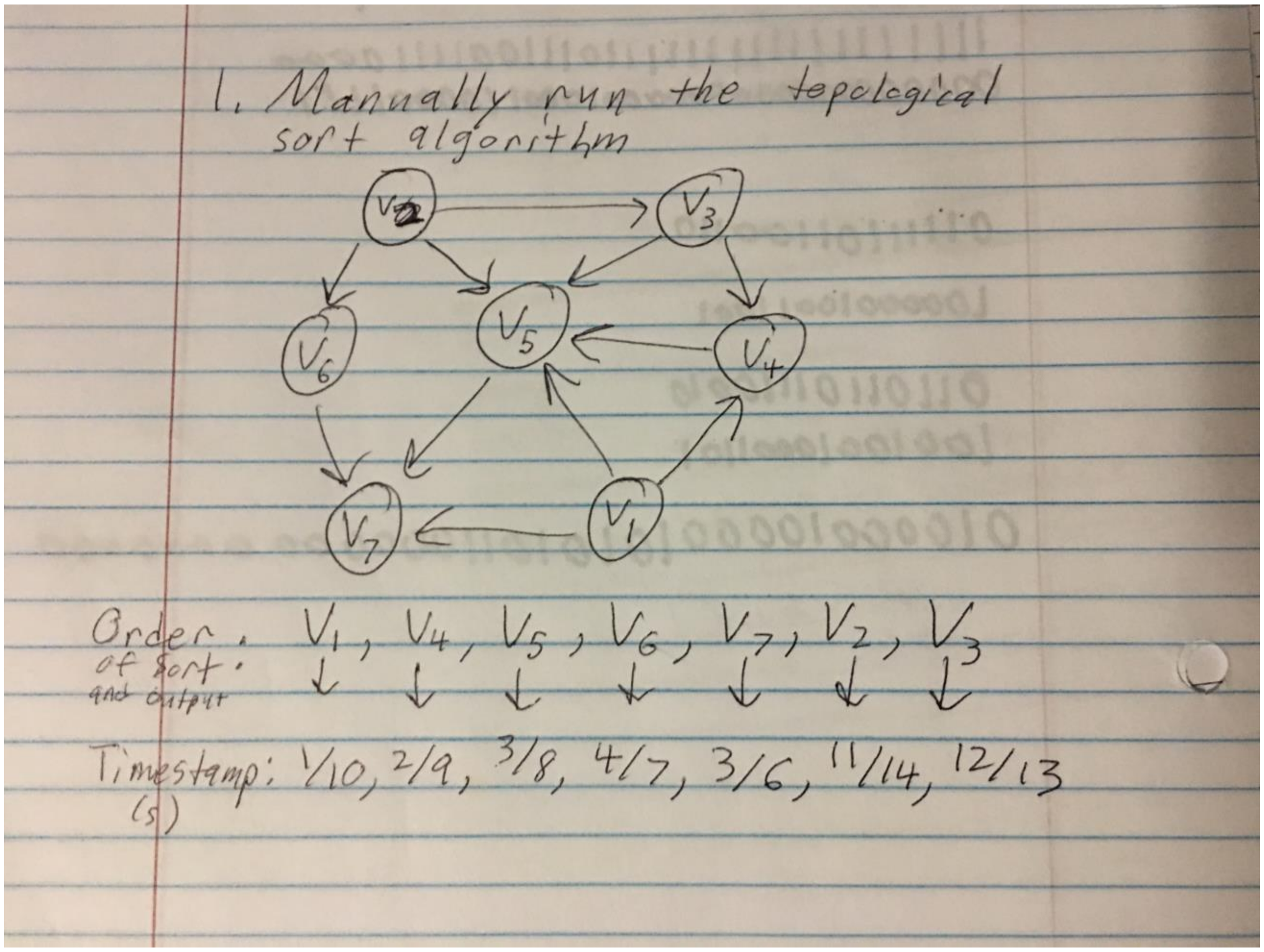
2/13/17

CSE 3500

Homework 3

1. Manually run the topological sort algorithm

The following picture is of the order of output of each node in the topological sort, with their timestamp of each output node just below the node (directed by an arrow):



1. Manually run the topological sort algorithm

Order of sort and output: V1, V4, V5, V6, V7, V2, V3

Timestamp(s): 1/10, 2/9, 3/8, 4/7, 3/6, 11/14, 12/13

2. Semi connected graph

For this problem, we must find a method in which to find out whether a directed graph that is given is semi connected or not. This can be done by finding if any of the given vertices u and v within all the vertices have paths connecting the two. First, let us construct an algorithm to show this is true.

First, assume we have a directed graph that is cyclic denoted as G, so a directed cyclic graph G. It can be assumed that this graph is semi connected in a topological sort. In this graph, G, since it is connected topologically, each vertex i, there exists an edge (v_i, v_{i+1}).

This is known to be true because the graph G is given the topological sort $v_1, v_2, \dots v_n$. If an edge (v_i, v_{i+1}) does not exists for some i, then a path of (v_{i+1}, v_i) also must not exists if no edge connects them. This is true in the case of a topological sort of a direct cyclic graph. G is not semi connected. If for every i there is an edge (v_i, v_{i+1}), then for each i, there is a j such that (iv_i *is connected to* v_{i+1}which is connected to some other node...and some node is connected to v_{j-1}which connects to v_j, and thus the graph G is semi connected.

Now that we have proof that the following given graph is semi connected, we can create an algorithm in which to find this out:

1. Find all maximal strongly connected components in the graph, a directed graph is a component graph.
2. Build the strongly connected components graph G1 = (U, E1) such that U is a set of strongly connected components. E1 = {(V1, V2) | there is v1 in V1 and v2 in V2 such that (v1, v2) is in E)}.
3. Now, do a topological sort on G1.
4. Finally, check if for every i, there is edge (V_i, V_{i+1}).

If the graph is semi connected, for a pair (v1, v2) such that there is a path between them. Let V1, V2 be their strongly connected components. There is a path from V1 to V2, and also from v1 to v2, since all nodes in V1 and V2 are strongly connected. If the algorithm yielded true, then for any two given nodes v1 and v2, they are in strongly connected components V1 and V2. There is a path from V1 to V2, thus also from v1 to v2. This ensures that a graph will be semi connected or not, since it checks whether a path exists between the two nodes, at least one or the other. This algorithm will run at O(V+E) time because it must go through all the vertices in graph G and as well as E1, also known as E. So, each check would add up to V+E times, making it O(V+E).

3. A distance problem

The first thing to do is run breadth-first search starting from the node s. Let d be the layer where you come across t. By the given assumption about the distance between the nodes s and t, distance (d) > n/2. Now, we can say that one of the layers Layer_1,… Layer_{d-1} has a single node. This is because if one of the layers does not have a single node, then the layers account for at least 2*(n/2) = n nodes. G has only n nodes, and s and t are not in these layers.

Let Layer_i be the layer containing that single node discussed previously, and denote it as node v. Now, delete v. A set S of all nodes in layers 0,…,i-1 also exists. This set cannot contain t. Any edge out of these nodes leads only to a node in Layer_i or stays in the set S, by the properties of breadth-first search. But v is the only node in Layer_i. Therefore, there cannot be any connecting path between the nodes s and t. This algorithm would run at O(m+n) time because it would have to go through all the nodes once, the nodes in the set of S, as well as all the edges connecting said nodes. Since all of them are checked once, then delete node v, then check if a path exists by checking all nodes and edges, the runtime would account for every node and edge checked, so m + n. This makes the runtime O(m+n), and also proves that there can exist some node v that can destroy all paths from node s and t.

Extra Credit

I'm not very certain on how to do this, but I believe it has something to do with using a BFS to find the path with the largest distance between the two nodes u and v. If the second vertex v is seen within the traversals of the BFS, return the nodes that this is connected to. You could then compare the distances of these paths to the other nodes, which are the only possible nodes able to be connected to node v, and see which is the largest. Next, compare it to all possible connections of paths within the graph to see which is the largest. I imagine the path between node u and v will be the largest, showing that these two nodes are the largest possible connection. You may also be able to do this with a DFS, but I'm not entirely sure how. It'd be cool to go over this question in class if possible because I'm interested in the method in which to do this!