CISC 380 **Instructor: Sarah Miracle**

Homework 5

Student: Solutions

1. A *vertex cover* of a graph $G = (V, E)$ is a subset of the vertices $S \subset V$ that includes at least one endpoint of every edge in E. Give a linear-time algorithm for the following task.

   Input: An undirected tree $T = (V, E)$.
   Output: The size of the smallest vertex cover of T.

Solution: Root the tree arbitrarily at some node r.

1. Let $VCover(n)$ be minimum cost of a vertex cover of the subtree descending from node n. For a leaf, we set $VCover(n) = 0$.

2. At any node n, we have the choice of including the vertex n, or not including the vertex n. If we include n, we cover all the edges between n and its children. The remaining subtrees descending from n's children still need to be covered, in child case $VCover(n) = 1 + \sum_{x \in n.children} VCover(x)$.

   If we do not include n, then we MUST include every child of n in the vertex cover. This leaves all the subtrees descending from n's gradchildren uncovered. In this case, $VCover(n) = |n.children| + \sum_{x \in n.children} \sum_{y \in x.children} VCover(y)$. We choose the smaller of the two.

   We use the convention that we can tack on attributes to each node of the tree; for each node, we allow *n.cover* to be an editable attribute. You also have copied the tree and stored this information in your copy, created an array (or hash table) with $|V|$ elements in which to store all this information, etc. This time, we'll use memoization instead of an iterative approach since it's cumbersome to iterate upwards from the leaves of a tree to the root.

3.
```
Global Tree $T$
function Init(node $n$)        ▷ Initializing takes $O(|V|)$ time $n.cover \leftarrow NULL$
    for $c \in n.children$ do
        $Init(c)$
function VCover(node $n$)
    if $n.cover = NULL$ then $n.cover \leftarrow \min(1+\sum_{x \in n.children} VCover(x), |n.children|+$
$\sum_{x \in n.children} \sum_{y \in x.children} VCover(y)$
    return n.cover
return $Init(T.root)$.
return $VCOVER(T.root)$.
```

4. Outside the summations, we do $O(1)$ work per node.

   Inside the summations, at each node n, we do an amount of work proportional to the number of all n's children and all n's grandchildren. This could be large for some nodes and small for others. To count this a different way, think about it backwards.

   Any node n is only ever considered inside a summation by n's parent or n's grandparent. This means $VCover(n)$ is only ever called inside a summation $O(1)$ times per node. Thus, the total work done by all the summations over every call of $VCover$ is $O(|V|)$.

   This giver us a total running time of $O(|V|)$. (Recall that in a tree, $|V| = |E| + 1$).

2. To get in shape, you have decided to start running to work. You want a route that goes entirely uphill and then entirely downhill, so that you can work up a sweat going uphill and then get a nice breeze at the end of your run as you run faster downhill. Your run will start at home and end at work and you are given as input a map detailing the roads with n intersections and m one-way road segments, each road segment is marked uphill or downhill. You can assume home is intersection 1 and work is intersection n, and the input is given by two sets: E_U representing the uphill edges and E_D representing the downhill edges, where each set is given in adjacency list representation.

   Give an algorithm that determines if there is a route (regardless of the length) which starts at home, only goes uphill, and then only goes downhill, and finishes at work.

   You should give a clear description of your algorithm in words, and you should explain its running time. Faster (in O notation) and correct is worth more credit.

   Here is an example with 5 vertices, where vertex 1 is home and vertex 5 is work. The input are the following arrays of linked lists:

   $E_U = 1 \to 2; 3 \to 4, 3 \to 5.$
   $E_D = 1 \to 3; 2 \to 3, 2 \to 4; 4 \to 5.$
   There is one uphill-downhill way to go from 1 to 5: $1 \to 2 \to 4 \to 5$.

Solution: We will present two algorithms for this problem. The first is easier to think of, but the second is somewhat more clever, and the ideas in the second algorithm will be useful for you in the future.

First run a DFS starting at home using only uphill edges, and mark all vertices reached. These are the vertices we can reach using only uphill edges. Then, run a DFS on the reverse graph starting at work - these are the vertices we can descend down to get to work. If we ever come across a marked vertex, return "True", else return "False". This requires two runs of DFS/reachability, which takes linear time.

Alternatively, we construct a NEW graph G' as follows. For each vertex $v \in V(G)$, make two copies, v_{up}, v_{down}. For every uphill edge in G, connect the corresponding vertices labeled *up*, and for every down hill edge, connect the corresponding vertices labeled *down*. Finally, put a directed arrow from v_{up} to $vdown$ to represent a transition from going uphill to downhill. Now, we run normal DFS to check if $home_{up}$ can reach $work_{down}$. Constructing the graph copy takes $O(V)$ work to copy the vertices, and $O(E)$ work to transplace edges in G to G'. Running a DFS on the new graph takes $O((2V) + (V + E)) = O(V + E)$ time.

IF we want to be rigorous, we need to show that G' has a path from $home_{up}$ to $work_{down}$ IF AND ONLY IF G has a valid uphill/downhill path from home to work. That way, we know that if there is a valid path in G, we will find it AND we know that we won't find paths that aren't actually valid in G. In this case, this statement is fairly clear from the construction of G', and a proof could be omitted, but in future problems of this type, this step will be necessary and less obvious.

Say G' has a path from $home_{up}$ to $work_{down}$. Such a path would have to "transition" fromm uphill to downhill vertices, since the only edges connecting uphill to downhill edges are transition edges by construction. Before this transition, this path only travelled between "up" vertices, which means by construction, the corresponding edges in G were all uphill.

Similarly after the transition, all the corresponding edges in G were downhill. Thus all the non-transition edges correspond to a a valid uphill/downhill path in G.

Now, say there is a uphill/downhill path in G, then the corresponding uphill path in G' exists among the up vertices, the transition vertex exists, and the corresponding downhill path in G' exists among the downhill vertices, connecting $home_{up}$ and $work_{down}$.

3. The police department in the city of St. Thomasville has made all streets one-way. The mayor contends that there is still a way to drive legally from any intersection in the city to any other intersection, but the opposition is not convinced. A computer program is needed to determine whether the mayor is right. However, the city elections are coming up soon, and there is just enough time to run a *linear-time* algorithm.

   (a) Formulate this problem graph-theoretically (this should involve clearly stating what the vertices and edges of your graph are) and explain why it can indeed be solved in linear time.

   Solution: Treat each intersection as a vertex, and each street as an directed edge between vertices. By asking if we can move from any intersection to any other intersection, we are asking if this graph is strongly connected.

   The SCC algorithm given in class is linear time, and we can tell that the mayor is right if there is a single SCC, and wrong if there is more than one.

   (b) Suppose it now turns out that the mayor's original claim is false. She next claims something weaker: if you start driving from town hall, navigating one-way streets, then no matter where you reach, there is always a way to drive legally back to the town hall. Formulate this weaker property as a graph-theoretic problem, and carefully show how it too can be checked in linear time.

   Solution: In our graph representation, the mayor is claiming that everything reachble from the town hall also has a path back to it. Consider the SCC meta graph of our graph. If the node containing the town hall had any outgoing edges, then the endpoint of that node would NOT have any path back to the town hall (as the meta graph is acyclic). If that node had NO outgoing edges, then the only set reachable from the town hall are vertices in its own SCC, which by definition always have a path back to the town hall. Thus, the property we are looking for is equivalent to the SCC containing the town hall being a sink node. First, run the SCC algorithm. This allows us to determine which vertices are in the same SCC as the town hall. To determine if this SCC is a sink SCC we just need to check if there are any edges adjacent to the vertices in this SCC that go to vertices outside the SCC. Since we've already compute the SCC numbers for each vertex this can be done in linear time, giving an overall linear time algorithm. $O(n + m)$ to run the SCC algorithm and then $O(n + m)$ to determine if the town hall SCC is a sink SCC.

4. **(PRACTICE)** Given a directed graph $G = (V, E)$, we want to create another directed graph $G' = (V, E')$ such that: G' has the same strongly connected components as G. G' has the same component graph as G. ¥E is as small as possible (i.e. minimize $|E'|$). Describe a fast

algorithm to compute G'. Prove that your algorithm is correct and justify its running time. Faster (in O notation) and correct is worth more credit.
Hint: Recall that a *connected* graph with $n-1$ edges is a tree. Note that two vertices in a tree cannot be in the same SCC - because there is a unique path between any two vertices in a tree and that path can only go in at most one direction (it might not be a directed path at all).

Solution: We observe that the way to make n vertices strongly connected using the fewest edges is to make a cycle with n edges. From the hint, we know that to make it connected we need strictly more than $n-1$ edges. So the fewest edges needed to make n vertices strongly connected is n. Thus for each SCC, we would minimize the number of edges needed to retain those vertices as an SCC by replacing any edges present with a cycle.

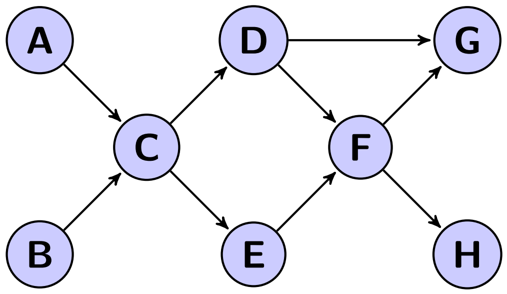
Between SCC's in the metagraph MG, we still need to retain that the SCC can reach all the SCC's that it could in MG. Each edge in MG between components represents potentially many edges in G between those components, but we can minimize this by just using one edge between any two vertices in those SCC's.

All edges in G are either within an SCC or between two different SCC's, so we've covered all edges in G in the above two cases.

Thus our final algorithm is: run the SCC algorithm to obtain the metagraph MG of the SCC's of G. Copy the vertices of G. For each SCC vertex in MG, connect the corresponding vertices of G with a cycle. For each edge between two SCCs in MG, add a single edge between any two vertices in the corresponding sets of vertices in G.

The SCC algorithm takes $O(V+E)$ time, and copying the vertices of G is $O(V)$ time. In the next step, each SCC of MG is considered once, and we do work on order the size of the SCC. Added up over all the SCC's in MG, this is $O(V)$ work. Finally, we do work for every edge in MG, which is at most the number of edge in G, which is E. Thus this runs in total time $O(V+E)$.

5. **(PRACTICE)** Run the DFS-based topological ordering algorithm on the following graph. Whenever you have a choice of vertices to explore, always pick the one that is alphabetically first.



(a) Indicate the pre and post numbers of the nodes.

Solution: In order of pre number, (the order of visitation), we have:

- A: 1, 14
- C: 2, 13

- D: 3, 10
- F: 4, 9
- G: 5, 6
- H: 7, 8
- E: 11, 12
- B: 15, 16

(b) What are the sources and sinks of the graph?

Solution: The sources are A, B, the sinks are G, H.

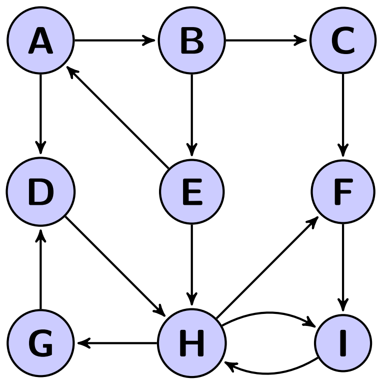
(c) What topological ordering is found by the algorithm?

Solution: In order of decreasing post number, B, A, C, E, D, F, H, G.

(d) How many topological orderings does this graph have? List them all.

Solution: There are 8 orderings - we can choose which of A, B, D, E, and G, H to place in front of the other, but everything else is fixed.

6. **(PRACTICE)** Run the strongly connected components 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.

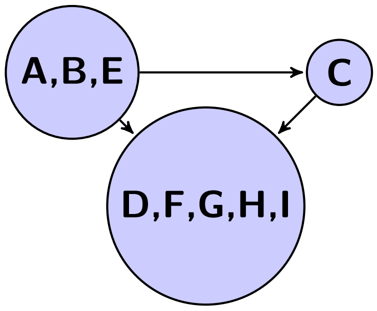


(a) In what order are the strongly connected components (SCCs) found?

Solution: We "find" SCCs when we finish exploring the reachable nodes at each location, in order of decreasing post number in the reverse graph. If you do this, you will first complete the component D, F, G, H, I, then C, and finally A, B, E.

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

Solution: A, B, E is the source SCC, D, F, G, H, I is the sink.



Solution:

(c) What is the minimum number of edges you must add to this graph to make it strongly connected?

Solution: I could add one edge from D to A say, and thus make a cycle in the meta graph. This would collapse that cycle into a single strongly connected component.

7. **(PRACTICE)** Give an efficient algorithm that takes as input a directed graph G and determines whether there is a vertex from which all other vertices are reachable. Prove your algorithm is correct and compute its running time. Faster (in O notation) and correct is worth more credit. (Note: The input graph G may not be acyclic.)

Solution: We'll start with the algorithm, and then we'll explain why it's correct.

Run the SCC algorithm from class in linear time. We note that the question is equivalent to asking "Is there an SCC from which all other SCC's are reachable," since every vertex belongs to some SCC, and all vertices inside an SCC can reach all other vertices in that SCC. Check each component in the meta graph MG to see how many sources there are (no incoming edges) in linear time in the size of the meta graph, which is at most as large as the original graph. Return TRUE if there is only one source in MG, and FALSE if there is more than one.

It's not obvious why there is only one source if and only an SCC (that source) reaches everything, so we give a short proof of correctness. First, consider the case that there are at least two source SCCs; then one source can't reach a different source, as there are no incoming edges to a source. Thus the property doesn't hold.

Now consider the case where there is only one source SCC in MG, call it X. Consider deleting all the SCC's reachable from X from the meta graph to obtain new meta graph MG'. The elements we deleted remaining had no incoming edges to anything in MG', since otherwise those elements would be reachable from X and would have been deleted. Thus any source in MG' was also a source in MG. We claimed that MG had only one source, and it was deleted in MG'. Since every non-empty DAG has some source vertex, this means that MG' must be empty, which means that everything in MG was reachable from the source.

An alternate algorithm could do two passes of a DFS, one to obtain all the post numbers, and then a second DFS from the element with highest post number, which is a "necessary candidate" for an element that reaches everything. You would need to prove that if some element reached every vertex, then so does the element with highest post number to ensure correctness - this ensures that the element with highest post number is all that you need to check.