Sample Solution for Problem Set 8

Data Structures and Algorithms, Fall 2020

January 4, 2021

Contents

1 Problem 1

Consider having a list for each potential in degree that may occur. We will also make a pointer from each vertex to the list that contains it. The initial construction of this can be done in time $O(|V|+|E|)$ because it only requires computing the in degree of each vertex, which can be done in time $O(|V| + |E|)$. Once we have constructed this sequence of lists, we repeatedly extract any element from the list corresponding to having in degree zero. We spit this out as the next element in the topological sort. Then, for each of the children c of this extracted vertex, we remove it from the list that contains it and insert it into the list of in degree one less. Since a deletion and an insertion in a doubly linked list can be done in constant time, and we only have to do this for each child of each vertex, it only has to be done $|E|$ many times. Since at each step, we are outputting some element of in degree zero with respect to all the vertices that hadn't yet been output, we have successfully output a topological sort, and the total runtime is just $O(|E| + |V|)$. We also know that we can always have that there is some element to extract from the list of in degree 0, because otherwise we would have a cycle somewhere in the graph. To see this, just pick any vertex and traverse edges backwards. You can keep doing this indefinitely because no vertex has in degree zero. However, there are only finitely many vertices, so at some point you would need to find a repeat, which would mean that you have a cycle.

If the graph was not acyclic to begin with, then we will have the problem of having an empty list of vertices of in degree zero at some point. That is, if the vertices left lie on a cycle, then none of them will have in degree zero.

2 Problem 2

(a) Given the procedure given in the section, we can compute the set of vertices in each of the strongly connected components. For each vertex, we will give it an entry SCC, so that v.SCC denotes the strongly connected component (vertex in the component graph) that v belongs to. Then, for each edge (u, v) in the original graph, we add an edge from u.SCC to v.SCC if one does not already exist. This whole process only takes a time of $O(|V| + |E|)$. This is because the procedure from this section only takes that much time. Then, from that point, we just need a constant amount of work checking the existence of an edge in the component graph, and adding one if need be.

(b) Professor Bacon's suggestion doesn't work out. As an example, suppose that our graph is on the three vertices $\{1, 2, 3\}$ and consists of the edges $(2, 1), (2, 3), (3, 2)$. Then, we should end up with $\{2, 3\}$ and $\{1\}$ as our SCC's. However, a possible DFS starting at 2 could explore 3 before 1, this would mean that the finish time of 3 is lower than of 1 and 2. This means that when we first perform the DFS starting at 3. However, a DFS starting at 3 will be able to reach all other vertices. This means that the algorithm would return that the entire graph is a single SCC, even though this is clearly not the case since there is neither a path from 1 to 2 of from 1 to 3.

3 Problem 3

(a) Use the algorithm in Problem 1 to perform a topological sorting on graph G and create $P[1...n]$ where $n = |V|$ and $P[1...n]$ are n nodes in V in the topololical order ($P[1]$ has in-degree 0 and $P[n]$ has out-degree 0). Initialize $cost[u]$ as p_u for any $u \in V$, then do the following loop:

- For $i = n$ to $i = 1$ do
  - For each node v where $(P[i], v) \in E$, let $cost[P[i]] = \min(cost[P[i]], cost[v])$.

(b) First create the component graph of G as $G' = (V', E')$. For each point u in G', suppose u corresponds to the nodes set $S_u \subseteq G$ as a components. Initialize $p_u = \min_{v \in S_u} p_v$. Then we get an instance $G', \{p_u\}_{u \in V'}$ which can be the input for **(a)**. Use algorithm in **(a)** on $G', \{p_u\}_{u \in V'}$ to get an array $cost[u]$ for $u \in V'$. For any $v \in V$, let $cost[v] = cost[u]$ where $v \in S_u$.

4 Problem 4

Suppose G is not an empty graph. First judge whether G is connected, if not, G cannot be "sort-of-connected". create the component graph of G as $G' = (V', E')$. For each point u in G', suppose u corresponds to the nodes set $S_u \subseteq G$ as a components. Calculate the topological order of points in G' at get $P[1...n]$. If there exists an edge from $P[i]$ to $P[i+1]$ for any $1 \leq i \leq n-1$, then G is sort-of-connected. Otherwise not. To prove the correctness of our algorithm, we prove the following lemma.

Lemma 1. *G is "sort-of-connected" iff. G' is "sort-of-connected".*

Proof. If G is "sort-of-connected", for every two points $u, v \in V'$, take arbitrary $u' \in S_u, v' \in S_v$. There exists a path from u' to v' or v' to u' in V, in which case there is a path from u to v or v to u in G'. Thus, G' is "sort-of-connected".

If G' is "sort-of-connected", for every two points $u, v \in V$, if $u, v \in S_w$ for some $w \in V'$, then there must be a path from u to v since they are in the same components. Otherwise, suppose $u \in S_{u'}, v \in S_{v'}$ for $u', v' \in V'$, there must be a path from u' to v' or v' to u', in which case there is a path from u to v or v to u. Thus, G is "sort-of-connected". □

Lemma 2. *P is defined above. If $P[i]$ has an edge to $P[i+1]$ for any $1 \leq i \leq n-1$, then G' is "sort-of-connected", otherwise not.*

Proof. If $P[i]$ has an edge to $P[i+1]$ for any $1 \leq i \leq n-1$, then any two nodes $P[i]$ and $P[j]$ with $i < j$ has a path from i to j: $P[i], P[i+1], ..., P[j]$. Thus, G' is "sort-of-connected".

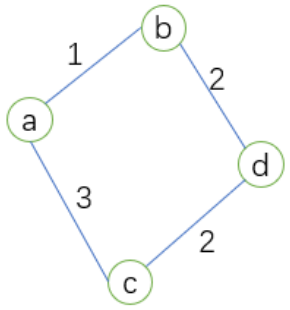
If G' is "sort-of-connected". Define an order on V' as: $u > v$ iff. u has a path to v. Since the graph is acyclic and "sort-of-connected", one can verify that the order is a total order. The topology order of a total order is $P[1...n]$. If $P[i]$ to $P[i+1]$ do not have edge, then there exists $P[j]$ where $P[j] < P[i]$ and $P[i+1] < P[j]$, which is impossible in any case $j < i$ or $j > i+1$. □

5 Problem 5

(a)

Disagree: counterexample is as follow.

- $S = a, b, V - S = c, d$, $A = \{(a, b), (c, d)\}$.
- $e(c, d)$ is a safe edge while it is not a light edge.



(b)

Disagree: counterexample is the same as (a).

- If partition V into $\{a, c\}$ and $\{b, d\}$, then (a, c) will be in the spanning tree.

6 Problem 6

(a)

If we use **Kruscal** to find the M.S.T. of the graph, each selected edge is unique, then the M.S.T. is unique.

(b)

Assume T' is not the M.S.T. of G'. There exists another spanning-tree T'_2,

$$weight(T'_2) < weight(T')$$

. Then $(T - T') \cup T'_2$ is a spanning-tree of G and

$$weight\left((T - T') \cup T'_2\right) < weight(T)$$

, which means T is not a M.S.T. of G.

7 Problem 7

(a)

If not, we assume that T' is the second-best minimum spanning tree which minimize $|E(T') - E(T)|$. By assumption, $|E(T') - E(T)| \geq 2$. Let e be the edge with minimum weight among $T - T'$, we have the following fact:

- There exists exactly one cycle C in $T' \cup e$ which contains e.
- There exists $e \neq e' \in C$ with $w(e') \geq w(e)$.

That leads to contradiction once one spots the fact that $T'' = T' - e' + e$ has smaller weight than T' and $|E(T'') - E(T)| = |E(T') - E(T)| - 1 \geq 1$.

(b)

That can be done via simple dfs. To be more precise, given vertex u, we may calculate $dist(u, v)$ for all $v \in V$ in $O(n)$ via dfs.

(c)

Run Kruskal algorithm, dfs described in (b), and enumerate all possible pair $(u, v) \notin T$.