COL 702

Homework I

Due on Sept. 4, 2021

In all cases, prove (succintly) why your algorithm is correct. Do NOT show any examples. Needlessly long arguments will fetch negative marks.

1. Let G be a directed graph. Let $C_1, C_2, \ldots, C_k$ denote the strongly connected components of G (each C_i is a subset of vertices). We now construct another directed graph H as follows: there are k vertices in H, denoted $v_1, \ldots, v_k$. There is a directed edge (v_i, v_j) in H if and only if there is an edge in G from a vertex in C_i to a vertex in C_j. Prove that H is a DAG. Give a linear time algorithm to construct H.

   Answer: Suppose H has a cycle $v_1, \ldots, v_k$, where v_i corresponds to the SCC (strongly connected component) C_i. By definition of H, there are edges $e_1, \ldots, e_k$, where $e_i = (u_i, v_i)$, with $u_i \in C_i, v_i \in C_{i+1}$ ($C_{i+1} = C_1$ if $i = k$). Since each of the C_i is SCC, there is a path from v_i to u_{i+1}, call this path P_i.

   We now argue that there is a path from v_1 to u_1. Indeed, consider the path obtained by concatenating $P_1, e_1, P_2, e_2, \ldots, P_k$. This yields a path from v_1 to u_1. Since there is also a path from u_1 to v_1 (the edge e_1), this implies that u_1 and v_1 should be in the same SCC. But this is a contradiction because $u_1 \in C_1, v_1 \in C_2$.

   In class, we saw Kosaraju's algorithm to compute the SCC's in linear time. In other words, we can find an array $S[]$ in linear time such that $S[v]$ is i if $v \in C_i$. Using this we first construct a graph H' as follows: we have one vertex v_i for each SCC C_i. Now we look at every edge $e = (a, b)$ in the graph G: if $S[a] = i, S[b] = j$ and $i \neq j$, we add an edge between v_i and v_j in H'. This is more or less what we want, but the problem is that H' may have parallel edges, whereas H should have at most one edge between any pair of vertices.

   To finally fix this, we show that we can take an graph H' which has parallel edges, and remove duplicate edges in linear time (in size of H'). Indeed, initialize an array $B[]$ to all 0. Further initialize a graph H to contain the same set of vertices as in H' and no edges. Now, for each vertex $v \in H'$, we do the following. For every incoming edge (u, v) in H' (note that there can be multiple copies of such an edge), we set $B[u]$ to 1. Now, we go through the incoming edges into v in H' once more: when we see an edge (u, v) and $B[u] = 1$, we set $B[u]$ to 0, and add the edge (u, v) to H. Note that whenever we look at the first copy of such an edge, we will add it to H' and subsequently another such copy will not be added because $B[u]$ will become 0. The time taken at a vertex v is in-degree of v, and the array B goes back to being all 0 entries when we finish this step for v. Thus, the conversion from H' to H takes linear time.

2. Call a directed graph G to be *weakly* connected if for every pair of vertices u, v , either there is a directed path from u to v or a directed path from v to u. Give a linear time algorithm to check if a directed graph is weakly connected.

Answer: Let $C_1, \ldots, C_k$ be the SCCs and let H be the graph mentioned in the previous question. Let v_i be the vertex for C_i in the graph H – let $v_1, \ldots, v_n$ be a topogical sort ordering of the vertices in H. Now we claim that G is a weakly connected if and only if H has edges of the form (v_i, v_{i+1}) for $i = 1, \ldots, k-1$. Clearly all of this can be done in linear time.

It remains to prove the claim above. First assume that H contains edges (v_i, v_{i+1}) for $i = 1, \ldots, k-1$. Consider two vertices u, v, where $u \in C_i, v \in C_j$. If $i \leq j$, then it follows that there is a path from u to v. Hence G is weakly connected.

Conversely, suppose H does not contain the edge (v_i, v_{i+1}). Let $u \in C_i, v \in C_{i+1}$. If there is a path from u to v, at least one edge in this path will have to cross from some component C_j to a component C_h, where $h < j$. But then violates the property of topological sort ordering of the vertices $v_1, \ldots, v_k$. Similarly, there cannot be a path from v to u in G. Thus, H is not weakly connected.

3. Describe an efficient algorithm to find the second minimum shortest path between vertices u and v in a weighted graph without negative weights. The second minimum weight path must differ from the shortest path by at least one edge and may have the same weight as the shortest path.

   Answer: We first run Dijkstra and computer shortest path from s to every vertex v – store the shortest path length in an array $D[v]$. Let pred(v) be the predecessor of v (on the shortest path from s to v) – again, this can be computed while running Dijkstra's algorithm. Let $s = v_1, v_2, \ldots, v_n$ be the order in which the vertices are visited by Dijkstra's algorithm (so pred(v) appears before v in this ordering).

   We will compute the second shortest path length $D'[v]$ for all v in this order as well. $D'[s]$ is ∞. Now suppose we have calculated $D'[v]$ for $v_1, \ldots, v_{i-1}$ and now want to compute $D'[v_i]$. Now two cases arise: (i) the predecessor of v_i in the second shortest path remains pred(v_i), or (ii) it is some other vertex $w \neq$ pred(v_i).

   In the first case, $D'[v_i]$ is equal to $D'[u] + w_{(u,v)}$, where u denotes pred(v_i) (observe that we have already computed $D'[u]$). In the second case, $D'[v_i] = D[w] + l_{(w,v)}$. Thus, $D'[v_i]$ is set to the minimum of these two quantities. The running time of this algorithm: first we run Dijktra's algorithm, and after that computing $D'[v_i]$ takes $O(degree(v_i))$ time. Therefore, the overall running time is dominated by the the time to run Dijkstra's algorithm.

4. Given a directed acyclic graph that has maximal path length k, design an efficient algorithm that partitions the vertices into $k+1$ sets such that there is no path between any pair of vertices in a set.

   Answer: Let $D[v]$ denote the maximum path length of a path ending at v. We can compute this as follows: let $v_1, \ldots, v_n$ be a topological sort ordering of the vertices. Clearly, $D[v_1] = 0$ (in fact, initialize it to 0 for all vertices). Now, we will compute this quantity in this order. When we come to v_i, the longest path ending at v_i must have the second last vertex as some v_j such that $(v_{,j}, v_i)$ is an edge. It follows that $j < i$,

and so, we already know $D[v_j]$. Therefore, we can set

$$D[v_i] = \max_{v_j:(v_j,v_i)\in E} D[v_j] + w_{(v_j,v_i)}.$$

It is easy to show by induction that this computes the length (i.e., number of edges) in the longest path ending at each vertex. So $D[v]$ lies in the range $\{0, 1, \ldots, k\}$. Also the time to compute $D[v_i]$ is proportional to the degree of v_i, and so, the overall time take to compute $D[v]$ for all the vertices is $O(n + m)$.

Let S_i be the set of vertices v for which $D[v] = i$. We claim that there cannot be a path between two vertices belonging to the same set S_i. Suppose not. Suppose $u, v \in S_i$ for some i and there is a path P from u to v. Composing this path with a path of length i ending at u (because $u \in S_i$), we get a walk of length $i + |P| > i$ ending at v. Since G does not have a cycle, each walk must be a path, and so there is a path of length larger than i ending at v. But this contradicts the fact that $D[v] = i$. Thus, the $k + 1$ desired sets are $S_0, S_1, \ldots, S_k$.

5. Suppose you are given an undirected graph G and a tree T on the same set of vertices. We would like to know whether it is possible to have an adjacency list representation of G (note that there are mutliple options here as the adjacency list can arrange the neighbours of a vertex in any order) such that running Breadth First Search on G with this adjacency list will result in T being the BFS Tree. Give an efficient algorithm to solve this problem (note that it is not enough for T to have the shortest path property, why?).

   Answer: We first check that T is a shortest path tree. This can be done easily in linear time as follows: run BFS on G and assign a level to each vertex. Run BFS on T and assign a level to each vertex. Now, check that the levels of a vertex in the two runs of BFS are the same. This check can be done in linear time, and clearly, if T does not satisfy this check, then T cannot be a BFS tree. So assume T is a shortest path tree. Let $l(v)$ be the level of v. Since it is a shortest path tree, it follows that for any edge (u, v), $l(u), l(v)$ differ by at most 1.

   Now we construct a directed graph H on the set of vertices V (same as the vertex set of G). An edge (u, v) in this graph would mean that BFS should visit u before v. We perform the following steps: for every vertex v, say at level l, let $u_1, \ldots, u_k$ be the neighbours of v at level $l - 1$. Let u_1 be the parent of v in T. Then we add edges $(u_1, u_2), \ldots, (u_1, u_k)$ to H. Having done this for all the vertices, we add some more edges to H as follows: for every edge $(u, v) \in H$, let u', v' be the parents of u and v in T. If $u' \neq v'$, we add the edge (u', v') also to H. We keep on doing this as long as we can add new edges to H. An efficient way of implementing this would be: initially add all edges in H after the first step above (of scanning every vertex) to a queue Q. This takes $O(n + m)$ time. Then we start dequeuing from Q, and whenever we dequeue an edge, we look at the parents of the two end-points, and add an edge between them as above, and insert it in the Q (if such an edge does not exist already) – we can use adjacency matrix for this, and so, the overall running time is $O(n^2)$. Thus, the overall

running time to construct H is $O(n^2)$. Also observe that if (u, v) is an edge in H, the u and v have the same level.

If H has a cycle, we declare T is not a BFS tree, otherwise we declare it is a BFS tree. Note that H has at most n^2 edges, and so the entire procedure runs in $O(n^2)$ time. We now prove correctness of the algorithm. First of all, we show that if we add an edge (u, v) to H, then a BFS traversal resulting in T must visit u before v. Suppose we add an edge (u_1, u_i) as explained above: if BFS visits u_i before u_1, then v will a child of either u_i or some other vertex visited before u_i. In either case, u_1 will not be the parent of v. Thus, u_1 must be visited before u_i. Further, it is also easy to check that if u, v are two vertices at level l where u is visited before v, and u', v' are the parents of u and v in T respectively, then u' must also be visited before v'. Thus all edges added to H are valid edges, i.e., if (u, w) is an edge in H, then any BFS traversal resulting in T must visit u before w. Hence if H has a cycle, T cannot be a valid BFS tree.

It remains to show that if H does not have a cycle then we can have T as a BFS tree. We first run topological sort on H and arrange the vertices in topological order, say $v_1, v_2, \ldots, v_n$. Now we arrange each adjacency list according to this order and run BFS. Let T' be the resulting BFS tree (we have to show that $T' = T$). We now prove that if (u, v) is an edge in H then BFS traversal will visit u before v. We prove this by the induction of the level of (u, v) (recall that both u and v are at the same level). This holds trivially for level 0 since there are no edges at this level. So assume this is true for level $l - 1$, and let $e = (u, v)$ be an edge in H, where both u, v are at level l. Let u', v' be the parents of u, v in T'. Two cases arise: (i) $u' = v'$: let w denote this common parent. When we visit w during BFS, we will consider u before v (since u appears before v in the topological sort, (ii) $u' \neq v'$: the graph H also has the edge (u', v') and so u' is visited before v'. Since u is a child of u' and v is a child of v' in T', we visit u, when we dequeue u' and visit v when we dequeue v'. So u is also visited before v (since u' is dequeued before v'). This completes the proof of this assertion.

Now we argue that $T' = T$. Suppose not. Then there is a vertex v which has parent u in T and parent u' in T' such that $u \neq u'$. We would have added the edge (u, u') to H and so by the assertion above, BFS (for constructing T') would visit u before u'. But then u' cannot be the parent of v in T'. This completes the proof.

6. Solve the above problem, but replace "BFS" by "DFS" everywhere.

   Answer: Let T be this tree. We check that all edges of G which are not in T are back edges, i.e., go from an ancestor to a descendant. To check if u is an ancestor of v, we can run DFS on T only, and use the discovery and finish times.

   Since we know that any DFS tree has this property. So, if T does not satisfy this property, it cannot be a DFS tree. It remains to show that if T satisfied this property, then it is a DFS tree. Let B be the set of back edges. So all the edges in G are either in T or B. For each vertex we arrange the edges incident to it in an order where the edges in T appear before those in B.

   Now we run DFS on G. We claim that we will get exactly T as the DFS tree. Suppose not. When we run DFS on G, let $e = (u, v)$ be the first edge in the DFS tree which is

not there in T (suppose u is visited before v). Since all edges not in T are back-edges with respect to T, v is an ancestor of u in T. Further, the path from the root to u in the DFS of G must be same as the path from root to u in T (because e is the first edge where DFS on G differs from T). Therefore, v must be an ancestor of u in the DFS tree of G, and so, we cannot take the edge e. This shows that running DFS on G will yield T.

Thus, we have a linear time algorithm for checking whether T is a DFS tree.