Problem 1: DPV 3.15 Computopia

The police department in the city of Computopia 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.
Note, linear time means $O(n+m)$ where $n = |V|$ and $m = |E|$.

(a) Formulate this problem graph-theoretically, and explain why it can indeed be solved in linear time.

Solution.

We will represent the city in this problem as a directed graph $G = (V, E)$. The vertices in V represent the intersections in the city, and the directed edges in E represent the streets of the city. Then, the problem is to determine whether a path from u to v exists for all $u, v \in V$, and to do so in linear time.

We can solve this problem using the `SCC` algorithm. If the entire graph G is itself a single strongly connected component, then the mayor's claim is true. The `SCC` algorithm takes linear time, as required.

(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.

The weaker claim requires that the town hall resides in a sink SCC. Why? If it lies in a sink SCC S then from the town hall we can reach every other intersection in S and from every other intersection in S we can get to the town hall. And, if S is not a sink SCC then there are edges out of it, and therefore there are intersections that can be reached from the town hall but cannot get back to the town hall.

The algorithm requires computing the SCCs and then checking if there are any edges out of the SCC containing the town hall. Hence, the total running time of this algorithm is $O(n+m)$, which is linear time.

Problem 2: DPV 4.14 Shortest paths through v_0
You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights along with a particular node $v_0 \in V$. Give an efficient algorithm for finding shortest paths between all pairs of nodes, with the one restriction that these paths must all pass through v_0.

A faster algorithm is worth more. Be sure to state/explain the running time of your algorithm. Hint: Use Dijkstras algorithm as a black-box. How many runs of Dijkstras algorithm do you need?

Solution.

For a pair of vertices u and z, a shortest path from u to z that passes through v_0 consists of a shortest path from u to v_0 and a shortest path from v_0 to z.

We do 2 runs of Dijsktra's algorithm. First, we run Dijkstra's algorithm on the graph G with start vertex $s = v_0$. This gives the shortest paths in G from v_0 to every other vertex. Let $dist_1[]$ and $prev_1[]$ denote the output from this run of Dijkstra's algorithm. Next we construct the reverse graph $G^R = (V, E^R)$ and then run Dijkstra's algorithm with start vertex $s = v_0$. This gives the shortest paths in G from every other vertex to v_0. Let $dist_2[]$ and $prev_2[]$ denote the output arrays from this run of Dijkstra's algorithm. Combining these paths gives the shortest paths between every pair of vertices with the extra restriction that we only consider paths that pass through v_0.

For a pair of vertices $u, z \in V$, let $dist[u, z] = dist_1[u] + dist_2[z]$; this is the length of the shortest path from u to z through v_0. The actual path from u to z can be reconstructed from the pair of arrays $prev_1[]$ and $prev_2[]$.

The algorithm takes linear time $O(n + m)$ to construct the reverse graph. The two runs of Dijkstra's algorithm take $O((n + m) \log n)$ time, and hence this portion of the algorithm takes $O((n + m) \log n)$. There is an extra factor of $O(n^2)$ to construct the array $dist[]$ of the shortest path length between all pairs of vertices. Hence the total running time is $O(n^2 + (n + m) \log n)$ if it is computing this array $dist[]$ with all shortest path lengths. The total running time is $O((n+m) \log n)$ if it is just implicitly computing it using the arrays $dist_1[], dist_2[], prev_1[], prev_2[]$. Both answers are fine for this homework problem.

Problem 3: Global Destination
In this problem: use the algorithms from class, such as DFS, BFS, Dijkstras, connected components, etc., as a black-box subroutine for your algorithm; see the instructions on the front page. Make sure to explain your algorithm in words.

Let $G = (V, E)$ be a directed graph given in its adjacency list representation. A vertex v is called a global destination if every other vertex has a path to v.

(a) Give an algorithm that takes as input a directed graph $G = (V, E)$ and a specific vertex s, and determines if s is a global destination. Your algorithm should have linear running time, i.e., $O(n + m) = O(|V| + |E|)$.

Solution.

Form the reverse graph $G^R = (V, E^R)$. Run `Explore` from s and check whether it can visit all other vertices in G^R. Return TRUE if s can visit all other vertices, FALSE otherwise. Both creating the reverse graph and running `Explore` take $O(n+m)$ time.

(b) Given an input graph $G = (V, E)$ determine if G has a global destination or not. The running time of your algorithm should still be $O(|V| + |E|)$. In this problem you are no longer given s, and you need to determine whether or not G contains a global destination.

Solution.

First, note that a global destination (if any exists) must be in a sink SCC S. To see why that holds, suppose that a global destination v is in a SCC S which is not a sink SCC. Thus, there are edges out of S and since the SCCs are a DAG there are SCCs (and hence vertices) reachable from S but from these SCCs we cannot reach S and thus v is not a global destination.

Now we know that if v is a global destination it must be in a sink SCC. What if there is more than one sink SCC? Suppose that there are multiple sink SCCs, say S and S'. There is no path between these SCCs S and S' since both are sink SCCs and hence have no out-going edges. Hence neither can contain a global destination.

Therefore, we know that if a global destination exists it must be in the unique sink SCC S. And if one vertex in S is a global destination then all vertices in S are in fact global destinations. Thus, we can run DFS on G^R and take the vertex with the highest post number, call it v. We know that v lies in a sink SCC of G. We now run the algorithm from part (a) to check if v is a global destination. If it is a global destination then we have found one, and if it is not then we know that there are no global destinations in G.

Alternatively, we can run the SCC algorithm on G and count the number of sink SCCs. There is a global destination iff there is exactly 1 sink SCC.

The `SCC` algorithm takes $O(n + m)$ time so the total running time of this algorithm is also $O(n + m)$.