Problem 1 [DPV] Problem 3.15 (Computopia)

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.

Answer:

Consider such a directed graph $G = (V, E)$.
Let the intersections in the city be the set of vertices V.
Let the streets of the city be the set of directed edges E.
We want to verify whether a path from u to v exists for all $u, v \in V$ in linear time.

According to the definition of strongly connected component, we know that the statement is true if and only if G is a strongly connected component.

Thus we can use the SCC algorithm learned in class to verify whether G is a strongly connected component itself. The algorithm takes a linear time of $O(|V| + |E|)$.

(b) 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.

Answer:

Form the graph G in the same way as stated in part (a). This time we construct another graph $G^R = (V, E^R)$, where $E^R = \{e(u,v) \mid e(v,u) \in E\}$. G^R is G with all edge reversed.

Let the town hall be vertex v_0 in G and G^R.

For $u, v \in V$, there is a path in G from u to v if and only if there is a path from v to u in G^R. This gives us the following algorithm to check the statement as stated by the problem:

(1) Construct G^R with all edge reversed in G : $O(|V| + |E|)$
(2) Run BFS/DFS on G with the starting node v_0. Store all the vertices that can be reached from v_0 as S : $O(|V| + |E|)$
(3) Run BFS/DFS on G^R with the starting node v_0. Store all the vertices that can be reached from v_0 as S^R : $O(|V| + |E|)$
(4) Compare the set S and S^R. Return true if $S \subseteq S^R$, and false otherwise : $O(|V|)$

The total running time for this algorithm is $O(|V| + |E|)$, which is a linear time.

Problem 2 [DPV] Problem 4.14 (shortest paths through v_0)

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

Answer:

Claim: For $u, v \in V$, a shortest path in G from u to v passing through v_0 is consisted of a shortest path from u to v_0 and a shortest path from v_0 to v.

Proof:
Let a shortest path from u to v_0 be p_1 and a shortest path from v_0 to v be p_2. Assume a shortest path from u to v passing through v_0 is p, which is consist of p_1' (path from u to v_0) and p_2' (path from v_0 to v).

If $cost(p) = cost(p_1') + cost(p_2') < cost(p_1) + cost(p_2)$, then WLOG, we can assume $cost(p_1') < cost(p_1)$, which violates the assumption that p_1 is a shortest path from u to v_0. Thus the claim is proved. $\square$

Here we construct another graph $G^R = (V, E^R)$ just as part (b) of problem 1, where $E^R = \{e(u,v) \mid e(v,u) \in E\}$. G^R is G with all edge reversed. And we know for $u \in V$, a shortest path in G from u to v_0 is a reversed shortest path from v_0 to u in G^R with the same cost. So we can construct the following algorithm with the help of Dijkstra Algorithm:

(1) Construct G^R with all edge reversed in G : $O(|V| + |E|)$

(2) Run Dijkstra Algorithm on G^R with the starting node v_0. Store all the shortest paths from v_0 to all $u \in V$ as $P_1 = \{p_1(v_0, u) \mid u \in V\}$: $O(|E| \log |V|)$ or $O((|E| + |V|) \log |V|)$

(3) Run Dijkstra Algorithm on G with the starting node v_0. Store all the shortest paths from v_0 to all $v \in V$ as $P_2 = \{p_2(v_0, v) \mid v \in V\}$: $O(|E| \log |V|)$ or $O((|E| + |V|) \log |V|)$

(4) For every pair $u, v \in V$, we get a shortest path from u to v by concatenate the reverse of $p_1(v_0, u)$ and $p_2(v_0, v)$. We store all these shortest paths as P : $O(|V|^2)$

The total running time is $O(|E| \log |V| + |V|^2)$ and it takes $O(|V|)$ time to output a shortest path for any specific pair of vertices u, v.

Problem 3 Global Destination

In this problem: use the algorithms from class, such as DFS, BFS, Dijkstra's, connected components, etc., as a black-box subroutine for your algorithm. If you attempt to modify one of these algorithms you will not receive full credit, even if it is correct. Make sure to explain your algorithm in words. (Note, this is problem 3.22 in [DPV].)

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.

Part (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., $O(n + m) = O(|V| + |E|)$.

Answer:

Run DFS or BFS on G^R (reverse graph of G) form s and check whether it can visit all other vertices in G^R. Return true if s can visit all other vertices in G^R and false otherwise. DFS or BFS takes $O(|V| + |E|)$.

Part (b). More Global Destination:

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.

Answer:

Run SCC algorithm on G and count the number of sink SCCs. G contains a global destination if and only if there exists exactly one sink SCC. The SCC algorithm takes $O(|V| + |E|)$ time.

Proof:
If the global destination exists, it must be in sink SCC according to the definition of sink SCC.
If there are more than one sink SCCs, say S_1 and S_2 are two of them. Let $s_1 \in S_1, s_2 \in S_2$, and we know that there is no path from s_1 to s_2, or from s_2 to s_1. So any vertex in sink SCC cannot be global destination, which implies there doesn't exist any global destination.

Another possible solution is first find s from a sink SCC in G by find the vertex with highest post number in G^R and check whether s is a global destination using the algorithm of part a.