Vivek Pandey

vpandey35@gatech.edu

Problem 1 (Computopia)

a) **Algorithm:** Let's consider a directed graph G such that G = (V, E), where V represents the vertices (all the intersections in the city), and E represents the directed edges (all the one-way streets). We can show that it is legally drivable from any intersection in the city to any other intersection if we can show that G is a single strongly connected component (SCC). In other words, all the vertices/intersections of G fall under one SCC.

In order to show this, we pass in the directed graph G = (V, E) in the form of adjacency list as inputs to the SCC algorithm and we receive as outputs the strongly connected components and the meta graph of connected components. Then, we need to check that the meta graph consists of a single vertex and no edges as there can only be a single SCC.

Runtime: Since the SCC black-box algorithm runs one DFS after another, the runtime is O(n+m), where n = |V| and m = |E|. Therefore, the algorithm runs in linear time as expected.

Correctness: We can show the correctness of this approach by analyzing the alternative scenario where there can be multiple SCCs in the graph. In such a case, by the fundamental nature of SCCs, not every intersection/vertex in G is reachable from every other vertex, otherwise there wouldn't be multiple SCCs. And hence, the only option that would satisfy the mayor's claim is if the graph only had a single strongly connected component.

b) **Algorithm:** Let's consider a directed graph G such that G = (V, E), where V represents the vertices (all the intersections in the city), and E represents the directed edges (all the one-way streets). We can show that if one leaves from the town hall intersection, then they can legally get back no matter which path they take if we can show that the townhall intersection/vertex falls in the sink SCC of G. In other words, the townhall intersection and all the intersections you can get to from the town hall intersection fall within the sink SCC of G.

In order to show this, like above, we pass in the directed graph G = (V, E) in the form of adjacency list as inputs to the SCC algorithm and we receive as outputs the strongly connected components and the meta graph of connected components. We analyze these outputs and verify that the strongly connected component where townhall intersection falls in is the sink SCC. We can do this by gathering all the vertices that are in the same SCC as the townhall, and running a DFS starting at the townhall vertex. If the DFS concludes by covering all the vertices of the townhall SCC and not visiting any other vertices outside the SCC, we can confirm that the townhall is in the sink SCC. This means, if one leaves from the townhall intersection, they can legally get back navigating the one-way streets.

Runtime: Since the SCC black-box algorithm runs one DFS after another and the verification that townhall is in sink SCC is another DFS, the runtime is O(n+m), where n = |V| and m = |E|.

Correctness: Only a sink SCC guarantees that the constituent vertices do not have an outgoing edge that leads outside the SCC. Every other SCC in the directed graph has an outgoing edge from the SCC and in such a case, there is no guarantee of the fundamental claim in this problem of being able to return to the townhall intersection.

Problem 2 (All edges in shortest paths)

Algorithm

We are given the undirected graph G = (V, E), where V are the vertices and E are the edges. Let's say we can get the positive edge weight of an edge $(a, b) \in E$ such that $\mathrm{a}, \mathrm{b} \in \mathrm{V}$ by using:

$weight(a, b)$ or $weight(b, a)$ in O(1) time.

First, we run Dijkstra's algorithm on G using vertex **s** as the source vertex. This gives us an output array, let's call it $dist_s$, which allows us to get the shortest distance from vertex **s** to any vertex $\mathrm{v} \in \mathrm{V}$ using:

$dist_s[v]$ in O(1) time

Next, we run Dijkstra's algorithm on the graph G again, but this time using vertex **t** as the source vertex. This gives us another output array, let's call it $dist_t$, which allows us to get the shortest distance from vertex **t** to any vertex $\mathrm{v} \in \mathrm{V}$ using:

$dist_t[v]$ in O(1) time

Now in order to determine the sets of edges that fall in at least one of the shortest paths from *s* to *t*, we loop through all the edges in E, and test the following condition on each edge, let's call it $(a, b) \in E$

$$min\{dist_s[a] + weight(a, b) + dist_t[b], dist_s[b] + weight(a, b) + dist_t[a]\} == dist_s[t]$$

The edges that satisfy the condition above fall in at least one of the shortest paths from s to t.

Run Time:

Dijkstra's algorithm to compute shortest distance from vertex **s** to all vertices: **O((n+m)logn)** where n = |V| and m = |E|

Dijkstra's algorithm to compute shortest distance from vertex **t** to all vertices: **O((n+m)logn)** where n = |V| and m = |E|

Iterating through all edges to check if an edge falls in the shortest path from **s** to **t:** **O(n)**

Therefore, the overall run time for my algorithm is **O((n+m)logn)**

Correctness:

An edge **e** can only fall in at least one shortest path from vertex s to t if there is a path from s to t such that it enters the edge e in one vertex, traverses and leaves from the other end towards t. Weight of such a path can be computed using minimum sum of three main sections: cost of getting from s to one vertex of edge e, weight of the edge e itself, and the cost of getting from the second vertex of edge e to vertex t. If the total weight of this path equals the weight of the shortest path from s to t, then the edge e is guaranteed to be on at least one shortest path.