(Sol) HW04 Graph algorithms (2)

Assume that we have the proximity data as an undirected graph $G = (V, E)$ - every person is represented as a vertex and there is an edge between two vertices if and only if two people represented by these vertices spent enough time in proximity for one person to catch a virus from the other person.

1. (6 points) In epidemiology, it is crucial to identify potential virus carriers for testing. Design an efficient algorithm that can identify all potential carriers for any given person (vertex) with a positive case.
   a. (2 points) Explain your algorithm. Clear English explanation is enough.
      Answer: Anyone who is connected from the given person should be tested, so we need to find all vertices reachable from the given vertex. Any flood fill algorithm works, e.g. BFS and DFS.
   b. (2 points) Analyze the time complexity of your algorithm.
      Answer: The time complexity of the BFS is $O(|V| + |E|)$.
   c. (2 points) Explain why your algorithm is the most efficient one. (In other words, why can't we do better?)
      Answer: The time complexity of the BFS is $O(|V| + |E|)$, but starting from the given vertex, the BFS algorithm will not check vertices and edges beyond the connected component this given vertex belongs to, and every vertex and edge in the connected component has to be visited to find all reachable vertices.
2. (6 points) "Community-based infection" refers to a positive case where the source of infection is not traceable using the proximity data, i.e. there is no path from this given positive case to another known positive case. Given G, a set of known positive cases S, and a newly found positive case v, design an algorithm to determine if this new case is a community-based infection.
   a. (2 points) Explain your algorithm. Clear English explanation is enough.
      i. Run any flood-fill algorithm like BFS and check each vertex against S. If found in S, return false (not community-based infection). Otherwise, return true.

   ii. Run Prim's algorithm from v, and every time a vertex u is added to the known set, check if u is in S. If found in S, return false (not community-based infection). Otherwise, return true.

   iii. Run Kruskal's algorithm on G. Traverse the tree v belongs to and check if any vertex in the tree is in S. If found in S, return false (not community-based infection). Otherwise, return true.

b. (2 points) Analyze the time complexity of your algorithm.

   i. BFS takes $O(|V| + |E|)$, and checking the membership in S takes $O(1)$ per vertex if using HashSet to represent S. Overall, $O(|V| + |E| + |V|)$ = $O(|V| + |E|)$.

   ii. Prim's algorithm takes $O(|E| \log |V|)$, but it does not check vertices and edges beyond the connected component v belongs to. So if we say E' is the set of edges in the connected component v belongs to, and V' is the set of vertices in it, then the time complexity is $O(|E\text{'}| \log |V\text{'}| + |V|)$.

   iii. Kruskal's algorithm takes $O(|E| \log |V|)$, and traversing the tree and checking against S takes at most $O(|V|)$, so the overall time complexity is $O(|E| \log |V|)$.

c. (2 points) Analyze the space complexity of your algorithm.

   i. Representing S in HashSet takes $O(|V|)$. BFS creates a `visited` array and a queue, so also costs $O(|V|)$. Overall $O(|V|)$.

   ii. Representing S in HashSet takes $O(|V|)$. Prim's algorithms create a `visited` array and a (priority) queue, so also costs $O(|V|)$. Overall $O(|V|)$.

   iii. Representing S in HashSet takes $O(|V|)$. Kruskal's algorithm creates a parent/rank array for $O(|V|)$.

3. (8 points) When new positive cases are continuously found, it is more important to have an efficient algorithm per case. Design an algorithm that checks if a new case is a community-based infection in $O(\log |V|)$ time. The algorithm may be "stateful", i.e. has some data stored in memory over all queries. (I will explain what a stateful service is in class.)

   a. (2 points) Explain your algorithm. Clear English explanation is enough.
   Answer: First, add edges from the first vertex in S to the second, from the

second to the third, etc to G. (You will be adding $|S| - 1$ edges.) This ensures that all vertices in S belong to the same connected component. Pre-process the graph G with the parent array and the rank(height) array, so we know which connected component each vertex belongs to. For a new positive case vertex v, run `find(v)` and `find(u)`with any vertex u in S and see they are in the same connected component. If they are, return false. Otherwise, `union` the two connected components and return true.

b. (2 points) Analyze the time complexity of preprocessing.
Answer: Adding edges takes $O(|S|)$ and the preprocessing takes as long as Kruskal's algorithm, so $O(|E| \log |V| + |S|)$, and $|S|$ is dominated by $|E|$, the overall time complexity is $O(|E| \log |V|)$.

c. (2 points) Show that the time complexity of checking a new case is $O(\log |V|)$.
Answer: Checking if two vertices are in the same connected component takes two `find()` operations, so $O(\log |V|)$.

d. (2 points) Analyze the space complexity of your algorithm.
Answer: The parent and rank arrays, either merged or separate, takes $O(|V|)$.