25 (100 PTS.) COVID-19 II

Since last week's homework, the number of corona virus cases has increased from 370,000 to more than 800,000; more than double within one week. At this rate, tens of millions will contract the virus before the end of the semester. Please stay at home and avoid social contact.

As we saw in last week's homework, graphs can be useful in analyzing and tracking the spread of viruses. Suppose you are given a graph G with n vertices and m edges and a parameter k. Each vertex represents an individual and each edge represents a social connection that can lead to a virus transmission. For every individual/vertex $v \in \mathsf{V}(\mathsf{G})$, you also given a value $C(v)$ which is "$+$" if the individual tested positive for the corona virus, and "$-$" otherwise.

An individual is ***at risk***, if he or she already tested positive for the virus or there are at least k other distinct individuals who tested positive for the virus and can reach the individual, i.e., there are paths in the graph from at least k vertices that tested positive to the vertex of interest (here, think about k as being a small number compared to n).

25.A. (50 PTS.) For the case that G is a DAG, describe an algorithm, as fast as possible, that finds all the ***at risk*** individuals. What is the running time of your algorithm? Argue briefly that your algorithm is correct.

Solution:

We do a topological sorting of G, and renumber the vertices $1, \ldots, n$, such that if there is an edge (i, j) in G then $i < j$. This takes $O(n + m)$ time.

Now, we compute for every vertex v (starting at 1 and going up to n) a set of at most k individuals who tested positive that can reach it.

We set initially $I(v) = \emptyset$. We add v itself to $I(v)$ if v is infected. Then, we go over all incoming edge $(z, v) \in \mathsf{E}(\mathsf{G})$ and add $I(z)$ to $I(v)$ by doing $I(v) \leftarrow I(v) \cup I(z)$. If the set $I(v)$ exceeds size k, we remove elements from it till it is of size exactly k, and we then immediately move on to the next vertex.

To perform this efficiently, we maintain the sets $I(v)$ as sorted arrays. As such, a union operation would takes $O(k)$ time using merge operation of two arrays of size at most k, and since every edge would require at most $O(k)$ work.

Finally, we scan all the vertices, and mark all the vertices for which their set $I(v)$ has size exactly k (i.e., there are $\geq k$ different infected vertices that can reach v) or are marked infected as vulnerable.

We get that the total running time is $O(nk + mk) = O(n + mk)$.

25.B. (50 PTS.) For the case that G is a general directed graph, describe an algorithm, as fast as possible, that finds all the ***at risk*** individuals. What is the running time of your algorithm? Argue briefly that your algorithm is correct.

Solution:

Assume the vertices of G are numbered 1 to n (in an arbitrary numbering). Compute the SCCs of G in $O(n + m)$ time. For every SCC U, compute the set $I(U)$ of the infected individuals in its SCC. As before, we shrink such a set of it contains more than k elements, till it contains exactly k elements. We need this sets to be listed in sorted order, but it is easy to verify that by careful implementation this can be done in linear time $O(n)$ – indeed, store for every vertex the index of the SCC that it belongs to, and scan the vertices from 1 to n, adding the infected nodes to the relevant list for the SCC that they belong to (again, not adding more than k elements to such a list).

Now, we run the algorithm from (A) on the meta-graph $\mathsf{G}^{\mathrm{SCC}}$, where every vertex U, has the set/list $I(U)$ precomputed (each such list has at most k elements). It follows that the same running time holds, and we know for each SCC if its vertices vulnerable or not. Propagating this information back to the original nodes takes $O(n)$ time.

Overall, the running time of the algorithm is $O(nk + mk) = O(n + mk)$.

Rubric: Getting an extra log factor because of sorting should cost only a few points (-10 pts).

Note that a simple solution to both parts would be to reverse all the edges of the graph, find the reach of each vertex v using graph search and determine if there are k vertices that tested positive and can be reached from v. However, such a solution is $O(n(n+m))$ and is worth 0 points.

26 (100 PTS.) **Taking the Bus**

You decide to take the bus to Walmart from home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Champaign-Urbana. It is extremely cold so you'd still like to spend as little time waiting at bus stops as possible. Unfortunately, there isn't a single bus that visits both Walmart and your home; you must transfer between buses at least once.

Describe and analyze an algorithm to determine a sequence of bus rides from your home to Walmart, that minimizes the total time you spend waiting at bus stops. You can assume that there are b different bus lines, and each bus stops n times per day. Assume that the buses run exactly on schedule, that you have an accurate watch, and that it is too cold to even contemplate walking between bus stops.

Solution:

Using Dijkstra

Assume that our input consists of the following information:

- For each bus, an array listing the n locations and times that bus stops, in chronological order. (These lists are published in a book available on every bus.)
- For each bus stop location, an array listing which buses stop at that location and when, in chronological order. (These lists are posted at every bus stop.)

We need at least one of these sets of arrays to solve the problem *at all*. Given only one of the two sets of lists, we can construct the other set of arrays in $O(nb \log nb)$ time essentially by brute force. The extra logarithmic factor comes from either sorting the lists chronologically, or using a priority queue to simultaneously walk through the union of the given lists in chronological order.

We build a directed graph G with nb vertices, each labeled by a bus stop and a time when some bus visits that stop, plus two additional nodes: a start node s representing you home, and a target node t representing Walmart. G has four types of edges:

- For every bus, G contains a sequence of $n-1$ *ride* edges joining its visits in chronological order. Every ride edge has weight 0.
- For every stop, G contains a sequence of directed *wait* edges joining its visits by buses in chronological order. The weight of a wait edge is the time between the two visits represented by its endpoints.
- G has *start* edges from s to *every* node representing the bus stop closest to home; these edges have weight 0.
- Finally, G has *target* edges from *every* node representing the bus stop closest to Walmart to t; these edges also have weight 0.

There are exactly $b(n-1)$ ride edges. In addition, every node has at most one outgoing wait edge, at most one incoming start edge, and at most one outgoing target edge. Thus, the total number of edges is at most $3nb = O(nb)$.

Every ride from home to Walmart is represented by a path from s to t in G. The length of this path is equal to the total time spent outside at bus stops. Thus, we want the ***shortest path*** in G form s to t.

We can compute this path by calling ***Dijkstra's algorithm***. The distance value $dist(t)$ tells us the time spent outside during the best trip home; the predecessor pointers allow us to reconstruct the actual ride path. Because there are no negative-weight edges, Dijkstra's algorithm runs in $O(E \log V) = \boldsymbol{O(nb \log nb)}$ ***time***.

Solution:

Another solution using DAG dynamic programming

We build the same directed graph G as in the previous solution, but we use a different algorithm to compute the shortest path from s to t. Specifically, G is a directed *acyclic* graph, because every edge leads from an earlier time to a later time. Thus, we can compute the shortest path in G from s to t using dynamic programming, as follows.

First, topologically sort the graph in $O(V+E) = O(nb)$ time. Then for any vertex v, let $dist(v)$ denote the true shortest-path distance from s to v. This function obeys the following recurrence:

$$dist(v) = \begin{cases} 0 & \text{if } v = s \\ \min_{u \to v} \big(dist(u) + w(u \to v)\big) & \text{otherwise} \end{cases}$$

This is a proper recurrence *because* G is acyclic! Moreover, if v is a source but not s, the recurrence correctly gives us $dist(v) = \min \varnothing = \infty$, so we do not need a separate explicit base case.

We can memoize this recurrence by storing each value $dist(v)$ at the corresponding vertex v. Since each distance value depends only on values later in topological order, we can compute all distances by considering the vertices in reverse topological order, altogether in $O(V+E) = \boldsymbol{O(nb)}$ ***time***.

However, unless we are given both the schedule for every bus *and* the schedule for every bus stop in the input, actually building the graph requires an additional $\boldsymbol{O(nb \log nb)}$ ***time***, which dominates the running time of the overall algorithm.

Rubric:

$O(nb \log nb)$ time via Dijkstra: 100 points: graph-reduction rubric.

$O(nb)$ time via dynamic programming: 100 points = 50 for reduction to shortest path (graph reduction rubric) + 50 from dynamic programming rubric (for shortest path in dag). But -10 for ignoring preprocessing time, unless the solution *explicitly* assumes input structures that would make sorting unnecessary.

These are not the only correct solutions.

27 (100 PTS.) **Stay stable**

We are given a directed graph with n vertices and m edges ($m \geq n$), where each edge e has a weight $w(e)$ (you can assume that no two edges have the same weight). For a cycle C with edge sequence $e_1 e_2 \cdots e_\ell e_1$, define the *fluctuation* of C to be

$$f(C) = |w(e_1) - w(e_2)| + |w(e_2) - w(e_3)| + \cdots + |w(e_\ell) - w(e_1)|.$$

27.A. (10 PTS.) Show that any cycle with the minimum fluctuation cannot have repeated vertices or edges, i.e., it must be a simple cycle.

Solution:

Let C be a cycle with the minimum $f(C)$. If there is more than one such optimal cycle, pick the one with the smallest number of edges. Let $e_1, e_2, \ldots, e_\ell, e_1$ be its edge sequence. Suppose that a vertex or edge is repeated. Then the end vertex of e_i is equal to the start vertex of e_j for some $j > i+2$. By the triangle inequality,

$$|w(e_i) - w(e_{i+1})| + \cdots + |w(e_{j-1}) - w(e_j)| \ > \ |w(e_i) - w(e_j)|.$$

Thus

$$\begin{aligned} f(C) &= |w(e_1) - w(e_2)| + \cdots + |w(e_{i-1}) - w(e_i)| + \\ &\quad\ |w(e_i) - w(e_{i+1})| + \cdots + |w(e_{j-1}) - w(e_j)| + \\ &\quad\ |w(e_j) - w(e_{j+1})| + \cdots + |w(e_\ell) - w(e_1)| \\ &> |w(e_1) - w(e_2)| + \cdots + |w(e_{i-1}) - w(e_i)| + \\ &\quad\ |w(e_i) - w(e_j)| + \\ &\quad\ |w(e_j) - w(e_{j+1})| + \cdots + |w(e_\ell) - w(e_1)|. \end{aligned}$$

So, the cycle $e_1, \ldots, e_i, e_j, e_{j+1}, \ldots, e_\ell, e_1$ has smaller fluctuation and is thus also optimal but has fewer edges: a contradiction.

27.B. (90 PTS.) Describe a polynomial-time algorithm, as fast as possible, to find the cycle with the minimum fluctuation.

Solution:

Let $G = (V, E)$ be the original directed graph. Construct a new directed graph G':

- For each edge $e \in E$, make e a vertex in G'.
- For each pair of edges (u, v) and (v, y) in E, create an edge from (u, v) to (v, y) with weight $|w(u, v) - w(v, y)|$.

The new graph $G' = (V', E')$ has m vertices, and at most $O(mn)$ edges (since for each edge $(u, v) \in E$, there are at most n choices for y).

A cycle with minimum fluctuaton in G corresponds to a cycle with minimum weight in G'.

We can find a shortest-weight cycle in G' by finding the shortest path between any two pair of nodes i.e. all-pairs shortest paths. We can do this by repeatedly runnning Dijkstra from every vertex in the grah. A shortest cycle corresponds to a shortest path from u to v, followed by an edge (v, u), so it suffices to minimize the shortest path distance from u to v plus the weight of (v, u). The computation takes $O(|V'|^2 \log |V'| + |V'||E'|)$ time by repeatedly running Dijkstra's algorithm $|V'|$ times from every start vertex. Since $|V'| = O(m)$ and $|E'| = O(mn)$, the algorithm takes $O(m^2 n)$ time.