COP 4516 Spring 2021 Week 7 Final Individual Contest Solution Sketches

Problem A: Spin

The input is a graph with up to 5000 vertices and 10000 edges. We must find a shortest distance (from vertex 0 to vertex n-1, where n is the number of vertices), but there is a catch! Not all routes are allowed. Any route with more than w crosswalks is not allowed for consideration. Normally, Dijkstra's algorithm would be used to find a single shortest distance in a graph, but our added restriction seems to put a damper on this plan.

Ultimately, when tracing through Dijkstra's algorithm, it becomes clear that when arriving at any vertex, **it's also necessary to know how many crosswalks** a particular path took to get there. Arriving at vertex a using 3 crosswalks is simply a different case than arriving at vertex a using 4 crosswalks. Thus, instead of storing a shortest distance for each vertex, **we must store a shortest distance for each ordered pair (v, k) where v is a vertex in the graph and k is the number of crosswalks used to arrive there.**

The effect of doing this is that Dijkstra's is run on a graph with n(w+1) number of vertices, where n is the number of vertices in the graph and w is the maximum number of crosswalks allowed. For these bounds we have 5000 x 21 = 105000 vertices. The effective # of edges is 10000 x 21 = 210000. The run time of Dijkstra's (efficient implementation) is O(ElgV). Thus, by plugging in these maximum values into this run-time, we see that the algorithm is indeed fast enough.

Thus, the key modification to the Dijkstra's code shown in class is to keep a two dimensional Boolean array for "used" vertices (or an equivalent shortest distance array), and when enqueing items into the priority queue, enqueue ordered pairs of vertex number and number of crosswalks. Alternatively, create a single integer, say, n*Crosswalks + v that stores both of these integers in a single integer. Stop looking for paths once ANY path to the class is found.

To solve the small case only, with $n \le 20$, we see that $n(w+1) \le 420$, and for this bound, Floyd Warshall's is fast enough to obtain the shortest distance between the dorm and class. After running Floyd's, take shortest entry of reaching class amongst using any number of crosswalks.

Problem B: Vaccine Distribution

This problem has a greedy solution. We always want to prioritize people who are leaving first. So, each hour, we want to provide the vaccine to the people who will leave the earliest. We simulate as follows:

Keep an array of size n+10, where index i stores the number of people who will leave i+1 hours after the beginning of the simulation.

For each hour, do the following:

1. Read in everyone who shows up and update the number of people who are waiting and will leave at each hour. (If this is hour i and the person is willing to wait j hours, then they will leave at hour i+j.)

2. Start at your current hour in the array and keep on vaccinating everyone (subtract them out of the array and add them to the vaccinated count) and move in order of hour as necessary until you've either used up all of your vaccines or have no one left to vaccinate.

This will run very fast. A simple implementation that doesn't use the fact that you just have to search for 10 hours from the current hour will run in $O(n^2)$ time. If you utilize the fact that at any given time, you just have to check up to 10 hours ahead of you, the run time improves to O(n). The key in either of these implementations is that the run-time is not dependent on the number of people vaccinated. I didn't want this to be a hard problem, which is why I created the data such that the $O(n^2)$ algorithm would easily run fast enough. (Note: two obvious ways to make the problem a bit harder is to increase n to 10^6 and increase the number of people in each hour to 10^9, the first requires the O(n) solution and the second requires the use of longs. I specifically didn't want to force students to make either of these observations.)

The reason that this problem was split up into three sub-cases is that there are solutions to smaller versions that don't pass the data with the bounds given:

For the small data, we must only vaccinate the people who show up at that hour. If more people show up than we can vaccinate, then we just vaccinate our maximum allowable amount per hour. Otherwise, we vaccinate everyone. So, just add up min(input, people) over each hour.

For the medium data, we can use a priority queue of people, where each person gets inserted into the priority queue as a separate integer, representing when that person would leave. When vaccinating, just pull people out of the priority queue, one by one. For the medium bounds, where we have at most 10000 people showing up over 100 days we get 1 million people maximum in the priority queue, which will run in time since each priority queue operation takes O(lg n) time, where n is the number of items in the priority queue.

Problem C: Volume

This problem was the easiest in the set. Simulate the process given. When hitting the up button, set the volume to the minimum of the maximum volume and the sum of the current volume and the addition for the up button. When hitting the down button, set the volume to the maximum of 0 and the current volume minus the subtraction for hitting the down button.

Problem D: Party Like It's Y2k21

To find the fewest number of button presses, use a Breadth First Search between volume levels, keeping track of how many pushes it takes to get to each possible volume level. Just like problem C, the most important part is to adjust a new volume if it gets too high or too low, using the min and max functions as needed. Then, just enqueue each new volume level, immediately storing its distance (in button presses) in the distance array. The goal with this problem was to reduce the problem obfuscation (Problem A had more of this) and just test who could implement a standard Breadth First Search.