CS 180 Midterm

Hanna Co

May 4, 2021

1.
(a) False, $2^n \neq \theta(3^n)$

(b) True, because if removing e diconnects the graph, than that edge is needed in order to access all nodes, thus it must be a tree edge in DFS.

(c) False. Say the DAG is comprised of two components that arent connected. In this topological sort, there would not be a path from the first to the last node.

(d) True. For example, you can remove nodes with degree 1 without disconnecting the graph.

2.

(a) We can keep a max heap with at most K elements, and a linked list that holds the rest of the elements. The max heap will contain the K smallest elements in our data structure. We can also maintain a pointer p_k to the element at the top of the max heap, which is the K-th smallest element. We also have another pointer p_min, which points to the smallest element on the linked list.

push:

We will compare the value we want to insert, n, to the item at the top of our max heap, m. If $n > m$, then we can simply insert n into our linked list, and adjust p_min if necessary. Otherwise, we use our pointer p_k and update it with the value n, and treat it as a single insert with time complexity $logK$ (for the re-heapification). We put m onto the linked list. We update our pointers p_k and p_min as necessary. This has a time complexity of $O(logK)$.

find_Kmin:

Since we have a pointer to the top element of the max heap, accessing it is simple $O(1)$. Alternatively, we can pop the value off the max heap and push it back on, for a time complexity of $2logK$, or $O(logK)$.

(b) Similarly, we can keep a max heap with at most K elements, and a min heap that holds the rest of the elements. We keep a pointer p_k to the top of the max heap, and pointer p_min to the top of the min heap.

push:

We will compare the value we want to insert, n, to the item at the top of our max heap, m. If $n > m$, then we can push n onto our min heap, which is a $logn$ operation and update p_min if needed. Otherwise, we can use our pointer p_k and update it to have value n, and treat it as a single insert, which has a time complexity of $logK$ (for re-heapification). We then push m onto our min heap, and update our pointers p_k and p_min as necessary. This has a worst case time complexity of $logK + logn$, or $O(logn)$.

pop:

We can return the value pointed to by p_k, and update p_k with the value at the top of the min heap, pointed to by p_min. There is no need for re-heapification, since we know the value at the top of the min heap will be larger than any of the values in the max heap. We then pop our min heap, with a time complexity of $logn$, and update p_min accordingly. This has a time complexity of $O(logn)$.

find_Kmin:
Since we have a pointer to the top element of the max heap, accessing it is simple $O(1)$.

3.

(a) Let's say that we end up with a matching where we have pairs (m_1, w_i), (m_2, w_j), (m_i, w_1), and (m_j, w_2). This is not a stable matching, because w_1 prefers either m_1 or m_2 to her current partner, and both m_1 and m_2 prefer w_1 to their partners. The same is true for w_2. Thus, in all stable matchings, m_1 and m_2 will be paired with w_1 and w_2.

(b) Let's say we can group the men and women, such that each group has $w_i, w_{i+1}, m_i, m_{i+1}$ (where i ranges from 1 to $\frac{n}{2} - 1$), so we end up with $\frac{n}{2}$ groups. Suppose the preference list within each group are as follows:

$$m_i : w_i > w_{i+1} > ...;$$

$$m_{i+1} : w_{i+1} > w_i > ...;$$

$$w_i : m_{i+1} > m_i > ...;$$

$$w_{1+i} : m_i > m_{i+1} > ...;$$

As we proved in part (a), with this preference list, we only have 2 possible stable matchings. So it follows that, since there are 2 possibilities for every $\frac{n}{2}$ groups, there are a total of $2^{\frac{n}{2}}$ stable matchings.

4.
The best approach is to perform 'binary search' n times.

Flip the first $\frac{n}{2}$ switches
    Change in desired door (opens/closes):
        You know that its controlled by one of these switches
        If $n = 2$:
            This must be the switch, so we mark it
            Return
        Otherwise, run this again on switches $[1, \frac{n}{2}]$
    Else:
        You know it's controlled by the second half of switches
        If $n = 2$:
            It must be the other, so we mark that
            Return
        Run this again on switches $(\frac{n}{2}, n]$

We would run this n times, decreasing n by 1 each time, so once a switch is 'marked', we don't touch it anymore.

Proof of Time Complexity:
Our 'binary search' is $O(logn)$, and since we do this for n doors, our algorithm ends up running $O(nlogn)$ times.

Proof of Correctness:
Since this is essentially binary search, we know that we will find the switch that corresponds to the desired door.

5.

(a) For each segment that you can walk without being teleported, create a node. For this example, we would have the nodes:

$(s, a_1), (a_1, b_1), (b_1, a_2), (a_2, c_1), (c_1, d_1), (d_1, d_2), (d_2, c_2), (c_2, b_2), (b_2, t)$

Create an edge between the nodes, such that if we have an edge going from u to v, v is the segment you would walk after walked u. In this case, we would have an edge going from (s, a_1) to (a_2, c_1). While a node might have multiple indegrees, with cannot have an outdegree greater than zero, because if we are only walking eastward, there is only one segment we can walk.
We start at node (s, x), where x can be any endpoint, and if we are able to find a path to node (y, t), where y is also any endpoint, then you will always reach the t. However, if we get stuck in a cycle, then this proves that we will never reach t.

(b)

Assume that the teleporters are sorted by starting points, in ascending order.

Create an array to represent the entire distance from s to t, with each entry representing a location that is a possible endpoint. We populate this array L with integers, 0 meaning there's nothing there, and 1 indicating that there is an endpoint present.
Using this array, we can create nodes representing the segments you can walk.
We can also use it to create egdes, representing teleportations.
We perform BFS to identify components and cycles, and push the cycles onto a max heap.
For each teleporter we want to add:

Identify the biggest cycle. If it exists:

We create a teleporter with an endpoint in one of the cycle's segments (nodes), and the other endpoint in one of the main component's nodes. This merges the two together, removes the cycle, and doubles the number of edges.

Else:

We create a new segment by putting an endpoint after the very last endpoint, and connecting that to the path.

Proof of Time Complexity:

Creation of the array takes n steps, as well as the creation of the nodes and edges. BFS takes $O(edges + vertices)$, but in the worst case, we have $2n + 1$ vertices and n edges, so we can simplify this to $O(n)$. Our loop for adding the new teleporters runs in $Klogn$ time, because of the max heap access. Thus we can simplify our total time complexity to O(Klogn).

Proof of Correctness:

In our DAG, the edges essentially represent teleportations, so we want to add more edges. We do this by first exploiting any cycles present, since those can expand our graph the fastest. If we still have more teleporters to add, we create our own components that much connect to the main graph.