CSE Template

Rohan Mukherjee

April 14, 2024

Problem 1.

We shall find the algorithm and prove it via induction. We first prove the following lemmata:

Lemma 1. *Suppose that G is a graph where each vertex has degree $\leq k$. Then G can be colored with $k+1$ colors.*

Proof. We use a greedy approach. First color any vertex with the first color. Then, for all of that vertex's neighbors, color them a color that was not the first. Repeat the process. Since each vertex has degree $\leq k$, it's neighbors can use up at most k colors, so we can always find a color that it's neighbors have not used to color it. This completes the proof. $\square$

Lemma 2. *For any connected graph G with $n \geq 1$ vertices which has a vertex with degree $\leq c-1$, we can color G with $c = \max_{v \in G} \deg v$ colors.*

Proof. Clearly the claim is true for the connected graph on a single vertex since there is no vertex with degree < 0. Suppose the claim is true for all connected graphs with $n \geq 1$ vertices with a vertex of degree $\leq \max_v \deg v$, and let G be a connected graph with n vertices with a vertex v of degree $\leq c-1$. Let $G' = G \setminus \{v\}$ be the resulting graph after removing v, and let $G_1, \ldots, G_m$ be G''s connected components. We know that each connected component has a vertex with degree $\leq c-1$, since we removed 1 edge from a vertex in each connected component. If it was the case that every vertex in a connected component had equal degree, that degree is $\leq c-1$ and hence we can color the entire graph with c colors by the first lemma. Otherwise, either the maximum degree on a connected component is c, and we have found a vertex of degree $\leq c-1$, or the maximum vertex is $\leq c-1$, and by the hypothesis that the vertex's do not have equal degree we can find a vertex with smaller degree than that. In any case, we can apply the inductive hypothesis to color each connected component with $\max_{v \in G_i} \deg v \leq c$ colors. Since v has degree $\leq c-1$, and we have access to c colors, we can color v something different than any of it's neighbors, which completes the proof. $\square$

Our algorithm, entirely inspired by the above proof, is as follows. First find $c = \max \deg v$. If $c < k$, the algorithm is from lemma 1, which is just coloring each edge a different color then it's neighbors. This algorithm just has to check the color of each of it's neighbors, which can be done in $O(n+m)$ time (each edge is looked at precisely twice). If $c = k$, we apply the structure of the proof from lemma 2. First we find a vertex v of degree $< c$. Then we remove v from our graph (which can be accomplished in $O(n+m)$ time), and use BFS to find the connected components of the new graph. We then call the algorithm inductively. The running time calculation goes as follows: $T(n) = \sum_{G_i \subset G\setminus\{v\}} T(|G_i|) + O(n+m)$. Since in the worst case we only get 1 connected component, and $\sum |G_i| = n-1$, we see that $T(n) \leq T(n-1) + O(n+m)$, which tells us that $T(n) = O(n^2 + nm)$ (in the best case we always split into ≥ 2 connected components of equal size, in which case I think we get $O((n+m)\log n)$). This completes the proof of the algorithm.

Problem 2.

Our algorithm is fairly simple. First, we reverse every edge. This can be accomplished in $O(n+m)$ time by simply switching the start and end vertex of each edge. After this, we run a DFS on the vertices in increasing order, and only call a DFS on a vertex if it hasn't been discovered yet. Given that vertex v was discovered on a DFS call of vertex w, we set the minimum vertex reachable by w to vertex v (including that v is the minimum vertex reachable by v). This algorithm runs in $O(n+m)$ time since we visit each vertex exactly once, and each edge at most once, and just do $O(1)$ operations in between each edge since we are only updating the minimum vertex reachable by a vertex.

We make the following observations:

(1) If w is reachable by v in the reversed graph, and w is not yet discovered, then w was not reachable by any smaller vertex than v in the reversed graph, so v is the smallest vertex it can reach.

(2) We see that we do NOT call the DFS algorithm on vertices that are already discovered. What if, say, a vertex v was discovered previously, and w's smallest index was v? Since we never call DFS on v, doesn't w have the wrong vertex as it's minimum? Let u be the vertex that discovered v. Since v can discover w, putting these paths together shows that u can discover w, a contradiction, since w's minimum index should've been v.

So the above observations show that if v is the vertex that discovers w in it's connected component, v is the smallest vertex reachable by v by the minimality of v (Since we call DFS on the vertices in increasing order). This completes the proof of correctness.

Problem 3.

We first prove the following lemma:

Lemma 3. *A connected undirected graph $G = (V, E)$ has an orientation of edges where there is no source iff G contains a cycle.*

Proof. We first prove the foward direction. Suppose that G does not contain a cycle. Then G would be connected and acyclic and hence be a tree. Clearly in a tree with 1 vertex, every orientation has a source since there is a vertex with degree 0 which we can take as the source. Suppose that every tree with $n \geq 0$ vertices has a source in any orientation of the edges, and let G be a tree with $n + 1$ vertices with a fixed orientation on it's edges. Then G has a leaf v, and having degree 1 must have it's only edge be an incoming edge. Letting $G' = G \setminus \{v\}$, G' has a source by induction. The only way that source could no longer be a source in G is if the one edge we removed was actually pointing to that source, which is clearly not the case since that edge points towards v. Thus G has a source in any orientation of it's edges, completing the forward direction.

Suppose now that G contains a cycle, and orient the cycle in the obvious way:

$$\begin{tikzcd}
\bullet \arrow[r] & \cdots \arrow[r] & \bullet \arrow[r] & \bullet \arrow[lll, bend left]
\end{tikzcd}$$

Since the graph is connected, there is a path from any other vertex to a vertex in this cycle. Orient any edge in this path to all be incoming. In the case where two paths overlap, we do not have to change the orientation of the overlap because they are already pointing away from the cycle. For example,

$$\begin{tikzcd}
 & & & & & v_1 \\
\bullet \arrow[r] & \cdots \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] \arrow[lll, bend left] & \bullet \arrow[r] \arrow[ru] & v_2
\end{tikzcd}$$

Once we have covered all the vertices in this way, orient any other edge arbitrarily (such as pointing it towards the lower number). Every vertex not in the cycle has in-degree at least 1 by hypothesis, and every vertex in the cycle has in-degree at least 1 by construction, so we are done. □

Our algorithm will now proceed as follows. First we find a cycle $v_1, \ldots, v_m$, via the following procedure:

First find a vertex with degree ≥ 1. If none exist, output "Impossible". This can be completed in $O(n)$ time, by checking the size of each entry of the adjacency list.

Since that vertex has degree at least 1, we can find a cycle by running a BFS from that vertex. If we ever reach a vertex that is already discovered, we have found a cycle. This can be done in $O(n + m)$ time (then we can backtrack to find the least common ancenstor for the true cycle). If we never find an already discovered vertex, the graph is acyclic, and equal to its BFS tree, so we can output "Impossible".

Then we remove the edges in this cycle (which can be done in $O(n + m)$ time), and run a BFS on each vertex in the cycle to see all the other vertices it can reach (we will see all other vertices can be found in this way). Since those edges do not use cycle edges, we can orient them as all outgoing (from the starting vertex in the cycle). This will run in $O(n \cdot (n+m))$ time in the worst case. Our above proof will then result in the correctness of the algorithm, once we can verify that removing cycle edges will preserve at least 1 path between any vertex outside the cycle and one inside it. This step is very lossy and I expect it can be improved by a factor of $O(n)$.

Suppose by contradiction that removing the cycle edges yields a graph where a vertex w not in the cycle is not connected to any vertex $v_1, \dots, v_m$. Adding the cycle edges back, we get a path from w to one of the vertices v_i. If there is only one cycle vertex left we are done, otherwise we can throw out all the other vertices including and after v_i and we still have a path from w to a cycle vertex. Repeating this process, noticing that it cannot continue forever, we have a path where we see precisely one v_i from the cycle, which means we did not use a cycle edge.

Problem 4.

We define our algorithm as follows:

First, sort the prices in increasing order $p_0 \le \cdots \le p_{n-1}$. Then define the array d by $d_i =$ the number of stocks bought with price p_i. Now, while $\sum_{i=0}^{n-1} d_i p_i \le B$, with j as the smallest index with $d_j < c_j$, if $\sum_{i=0}^{n-1} d_i p_i + p_j \le B$, replace $d_j \leftarrow d_j + 1$, else return d.

We make the following observations:

(1) The solution returned by greedy is always feasible, since it will only ever buy a new stock if it's under budget.

(2) The algorithm will only increase j if it cannot by any more stocks of price p_j, when either $d_j = c_j$ or if the budget is exceeded.

(3) If e is the array of stocks bought by the optimal solution with the prices in the above order, $e_0 \le d_0$. If $e_0 > d_0$, then when the greedy algorithm still had $j = 0$ it would've chose to buy another stock of the smallest price, since that would still be feasible since the optimal had it, a contradiction.

Suppose that $c \neq d$ (the optimal is not the greedy). If somehow $e_i \le d_i$ for all i, then by assumption they are not equal, so $e_i < d_i$ for some i, meaning we could buy one more stock of price p_i, and noting that after this change, $\sum_{i=0}^{n-1} e_i p_i \le \sum_{i=0}^{n-1} d_i p_i \le B$ since $e_i \le d_i$, and greedy is feasible by observation (1), a contradiction since we could add more stocks to the optimal. Thus we can find the smallest k with $e_k > d_k$. By observation (2), $k \ge 1$. By construction, $e_i \le d_i$ for each $i \le k$. If $e_i = d_i$ for each $i \le k$, when the greedy algorithm was at $j = k$, the greedy algorithm would've bought another stock of price p_k since the optimal was allowed to (meaning the solution would still be feasible), a contradiction since then the algorithm would've moved on early. Since $k \ge 1$, there is some $i < k$ with $e_i < d_i$. Then we can delete a purchased stock from e_k and add it to e_i, maintaining the same amount of purchased stocks. Repeating this process until $e = d$ shows that greedy is optimal.

The above algorithm runs in $O(n \log n)$ time, since it has to sort the prices, and it can find the maximum number of stocks to buy of price p_i in $O(1)$ time by a simple calculation, namely $d_i = \min\left(c_i, \left\lfloor \frac{B - \sum_{j=0}^{i-1} d_j p_j}{p_i} \right\rfloor\right)$ (We have assumed that we would store the value $\sum_{j=0}^{i-1} d_j p_j$ in a variable named total and update it each loop iteration). Thus the rest of the algorithm runs in time $O(n)$, which shows our algorithm is polynomial time.