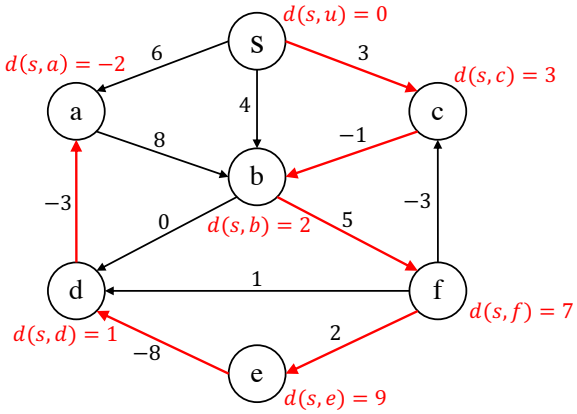
28 (100 PTS.) Changing the Weights

Let $G = (V, E)$ be a directed graph with edge lengths that can be negative. Let $\ell(e)$ denote the length of edge $e \in E$ and assume it is an integer, $\ell(e) \in \mathbb{Z}$. Assume you have found shortest path tree T rooted at a source node s that contains all the nodes in V. You also have the distance values $d(s, u)$ for each $u \in V$ in an array (thus, you can access the distance from s to u in $O(1)$ time). Note that the existence of T implies that G does not have a negative length cycle.

For example, consider the graph below, T is shown using red edges and the minimum distance $d(s, u)$ is shown next to each node.



28.A. (20 PTS.) Let $e = (p, q)$ be an edge of G that is *not* in T. Given e, show how to compute in $O(1)$ time the smallest integer amount by which we can decrease $\ell(e)$ before T is not a valid shortest path tree in G. Briefly justify the correctness of your solution. For example, in the graph above it is enough to decrease (b, d) by -2 for T to become invalid.

Solution:

Changing the length of e can only change the length of paths including e. Since a shortest path tree must contain a shortest path to each vertex, changing the length of e can change the shortest path tree only if it changes the shortest path to some vertex to a path using e. If e is in the shortest path to some vertex, it must be in a shortest path to q (since if there was a shorter path to q not using e than any path to q using e, every path using e could be made shorter by using this alternate path to q).

The shortest path to q which uses e is clearly the shortest path to p followed by the edge e, so decreasing $\ell(e)$ will cause T to become invalid if and only if it causes $d(s, p) + \ell(e)$ to become less than $d(s, q)$, that is if we decrease $\ell(e)$ by at least $d(s, p) + \ell(e) - d(s, q) + 1$

Since we assumed that we can access $d(s, v)$ in constant time for any v, computing $(d(s, p) + \ell(e) - d(s, q) + 1)$ takes $O(1)$ steps.

28.B. (80 PTS.) Let $e = (p, q)$ be an edge in the tree T. Given e, show how to compute in $O(m+n)$ time the smallest integer amount by which we can increase $\ell(e)$ such that T is no longer a valid shortest path tree. Your algorithm should output ∞ if no amount of increase will

change the shortest path tree. Briefly justify the correctness of your solution. For example, in the graph above it is enough to increase (d, a) by 9 for T to become invalid.

Solution:

Increasing the length of e only makes paths using e worse, so it cannot change the shortest path to any vertex whose shortest path does not currently use e. Thus, if increasing the length of e causes T to become invalid, it must be because it causes a different path to be shorter than the one in T for reaching q or some descendant of q in T, that is some vertex in the subtree T' rooted at q.

For any vertex v in the subtree of T', when we increase the length of e by k, we increase the length of the path to v in T by k as well. Thus, increasing the length of e by k will cause T to be invalid if and only if some such v can be reached by a path of length less than $d(s, v) + k$ which does not use e.

Let $d'(s, v)$ be the length of the shortest path from s to v which does not use the edge e (or ∞ if no such path exists). Then the maximum amount that we can increase the weight of e by without causing T to become invalid is $k = \min_{v \in T'} d'(s, v) - d(s, v)$.

Fix some vertex v in T' such that $d'(s, v) - d(s, v) = k$, the minimum value, and the number of edges in the shortest path from s to v with the fewest edges is less than or equal to the number of edges in the shortest path from s to t with the fewest edges for any other $t \in T'$ with $d'(s, t) - d(s, t) = k$. Let (u, v) be the last edge in a shortest path from s to v with the minimal number of edges.

We prove that u must be in $T \setminus T'$. If u was in T', then we have that $d'(s, v) = d'(s, u) + \ell(u, v)$ and $d(s, v) \leq d(s, u) + \ell(u, v)$ (since the shortest path to v using edge e may or may not go through u), and thus, $d'(s, u) = d'(s, v) - \ell(u, v)$ and $d(s, u) \geq d(s, v) - \ell(u, v)$, and so $d'(s, u) - d(s, u) \leq d'(s, v) - d(s, v) = k$. Thus, if u was in T', then it would also have the minimum difference between the length of the shortest path to it not using e and the length of the shortest path using e, but the number of edges in the path to u with the fewest would be less than the number of edges in the shortest path to v with the fewest edges, since the path to u is a subpath of the path to v. This would contradict our assumption above, and thus u cannot be in T'.

Since $u \in T \setminus T'$, the shortest path to u in the original graph already didn't use e, and increasing the weight of e cannot possibly change this. Thus, we find that $d'(s, v) = d(s, u) + \ell(u, v)$. We can compute this value without knowing u by simply taking the minimum of this value across all valid choices of u, since we can create a path to v by taking the shortest path to some other vertex with an edge to v and then following that edge. Thus $d'(s, v) = \min_{u \in T \setminus T'} d(s, u) + \ell(u, v)$.

As noted above, the maximum amount that we can increase $\ell(e)$ by without rendering T invalid is $\min_{v \in T'} d'(s, v) - d(s, v)$ which we now know is equal to $\min_{v \in T'}(\min_{u \in T \setminus T'} d(s, u) + \ell(u, v)) - d(s, v)$. The minimum integer amount that we can increase by to render T invalid is obviously just this value plus one.

Putting all of the above observations together, we obtain the following algorithm:

```
FindCostIncrease(G, T, e):
T' ← DFS(T, q)
k = ∞
For (u, v) ∈ E \ {e}
   if u ∉ T' ∧ v ∈ T'
      k = min (k, d(s, u) + ℓ((u, v)) − d(s, v))
return k + 1
```

Finding T' takes $O(n)$ time since we are running DFS on a tree. The remainder of the algorithm is iterating over the edge list of G and doing constant work for each edge, so it runs in $O(m)$ time, and thus our total runtime is $O(m+n)$.

Rubric: Out of 100 points:

- Part A 20 pts:
  - 10 pts Correct algorithms
  - 10 pts Brief explanation
- Part B 80 pts:
  - 50 pts Correct algorithm
  - 20 pts Brief correctness justification
  - 10 pts Runtime analysis

29 (100 PTS.) 5G Cellular Deployments II

We have seen or will see in midterm 2 a question on 5G small cellular deployments. In that question, our goal was to minimize the cost of the deployment without taking into account whether it ensures coverage to all customers. In this question, we will try to ensure coverage without caring much for the cost. (Note that you do not need to have solved the midterm to solve this question nor does solving this question helps you on the midterm.)

Suppose there are n customers living on Green street which is a perfectly straight street. The ith customer lives at distance x_i meters from the beginning of the street (i.e., you are given n numbers: $0 \leq x_1 < x_2 < \cdots < x_n$). `GlobalCell` is planning to connect all of these customers together small cell 5G base stations. A base station, which can be placed anywhere along Green street, can serve all the customers in distance r from it.

The input is $x_1, x_2, \ldots, x_n, r$. Describe a greedy efficient algorithm, as fast as possible, that computes the *smallest* number of base stations that can serve all the n customers. Namely, every one of the n customers must be in distance $\leq r$ from some base station that your algorithm decided to build.

As with all greedy algorithms, you must always prove the correctness and running time of your algorithm.

Solution:

We take the following greedy approach. First, break the points into groups by scanning the locations in increasing order. Starting at x_1, let x_{i_1} be the first location that is in distance $> 2r$ from x_1. Continue scanning, and let x_{i_2} to be the first location that is in distance larger than $2r$ from x_{i_1}. In the jth iteration, set $x_{i_{j+1}}$ be the first location that is in distance larger than $2r$ from x_{i_j}. Let x_{i_k} be the last such location marked.

This breaks the location into groups

$$
\begin{aligned}
I_1 &= \{x_1, \ldots, x_{i_1-1}\} \\
I_2 &= \{x_{i_1}, \ldots, x_{i_2-1}\} \\
&\ldots \\
I_k &= \left\{x_{i_{k-1}}, \ldots, x_{i_k-1}\right\} \\
I_{k+1} &= \{x_{i_k}, \ldots, x_n\}\,.
\end{aligned}
$$

For each interval I_j, we place a base station at the location $m_j = (\text{left}(I_j) + \text{right}(I_j))/2$, where left and right are the two extreme locations in the interval I_j.

Running time. Since the numbers are presorted, the running time is $O(n)$.

Correctness. We will simply sketch the idea and outline of the proof which is by induction and contradiction. We, however, expect students to write a formal induction proof.

Let $o_1 < o_2 < \cdots < o_u$ be the optimal solution, and assume for the sake of contradiction that $u < k+1$. We can safely assume that a client is assigned to its nearest base station, which implies that the optimal solution breaks the clients into groups $O_1, O_2, \ldots, O_u$, that are sorted from left to right. If $o_1 < m_1$, then we can set $o_1 = m_1$ – this would not make the solution any worse. Similarly, if $o_1 > m_1$, then we can set $o_1 = m_1$ (again), since O_1 can not contain any client from I_2. We now repeat this argument, inductively, moving the locations of the optimal solution to the greedy algorithm locations. Clearly, if $u < k+1$, then the clients in I_{k+1} can not be served by the new solution (which has the same coverage as the optimal solution), which is a contradiction.

Rubric:

- 50 pts for correct algorithms
- 40 pts for correct proof
- 10 pts for correct runtime analysis

30 (100 PTS.) Minimum Spanning Tree

30.A. Consider the following "local-search" algorithm for MST. It starts with an arbitrary spanning tree T of G. Suppose $e = (u, v)$ is an edge in G that is not in T. It checks if it can add e to T and remove an edge e' on the unique path $p_T(u, v)$ from u to v in T such that tree $T' = T - e' + e$ is cheaper than T. If T' is cheaper then it replaces T by T' and repeats. Assuming all edge weights are integers one can see that the algorithm will terminate with a "local-optimum" T which means it cannot be improved further by these single-edge "swaps". Assuming all edge weghts are distinct prove that a local-optimum tree is an MST. Note that you are not concerned with the running time here.

Solution:

We need to prove that the local optimum tree is also the (unique, since weights are different) MST. Let MST be tree T and let tree T' be any local optimum tree. As shown in class, for a graph with distinct edge weights, the set of all safe edges forms the unique MST T. We will prove that T' is the MST by contradiction.

Assume, T and T' differ by at least one edge. Let $e = (u, v)$ be an edge present in T but not in T'. Consider the path $p_{T'}(u, v)$. Since T' is a local optimum tree, we have the property that every edge e' in $p_{T'}(u, v)$ has a lower edge weight than e (i.e. $w(e') < w(e)$), since otherwise, we can replace e' by e to get a lower weight tree.

Now, we prove that e cannot be a safe edge. Consider any partition of the vertices into S and $V \backslash S$, such that e rosses the partition (i.e. $u \in S$ and $v \in V \backslash S$). Then, there exists some other edge $e'' \in p_{T'}(u, v)$ with one end in S and other in $V \backslash S$. (Note that $p_{T'}(u, v) \cup e$ forms a cycle and since e crosses the partition, some other edge should also cross the partition). But as seen before, $w(e'') < w(e)$. Thus, e'' crosses the partition and has a lower weight. This is implies that e is not the minimum weight edge across any partition.

Thus, e is not a safe edge. But then, e cannot be present in the MST T. Contradiction.

30.B. We saw in lecture that Borouvka's algorithm for MST can be implemented in $O(m \log n)$ time where m is the number of edges and n is the number of nodes. We also saw that Prim's algorithm can be implemented in $O(m + n \log n)$ time. Obtain an algorithm for MST with running time $O(m \log \log n)$ by running Borouvka's algorithm for some number of steps and then switching to Prim's algorithm. This algorithm is better than either of the algorithms when $m = \Theta(n)$. Formalize the algorithm, specify the parameters, argue carefully about the implementation and analyze the running time details. No proof of correctness required but your algorithm should be clear.

Solution:

We first check that the graph is connected by running DFS. If not, we return false. Otherwise, we can run the Boruvka's Algorithm for $\log \log n$ iterations, shrink each connected component in the forest formed, to a single vertex, and then run Prim's Algorithm on the resulting graph.
By shrinking the connected component, we mean creating a new graph, where for each connected component, we have one vertex. We add an edge from one connected component c_1 to connected component c_2, if there exists an edge between a vertex in c_1 to a vertex in c_2. The weight of the edge between c_1 and c_2 would be the minimum weight of all edges from some vertex in c_1 to some vertex in c_2.

Algorithm Steps:

- Run DFS to ensure graph is connected. If not, return "No MST".
- Run Boruvka's Algorithm for $\log \log n$ iterations. Let the partial forest formed by T.
- Run DFS on T to find all connected components and create the connected component graph as described above. Let the new graph formed be $G' = (V', E')$
- Run Prim's Algorithm on G' with same weights for edges as in G. Let the tree formed be T'.

- Return $T \cup T'$.

Runtime Analysis:

Boruvka's Algorithm runs for $\log \log n$ iterations, each iteration taking $O(m)$ time. Thus, it runs in $O(m \log \log n)$ time.

Finding connected components and shrinking to a single vertex can be done using a single DFS, and takes $O(m + n)$ time.

In every iteration, in the worst case, Boruvka's algorithm halves the number of connected components. Hence, after $\log \log n$ iterations, the number of connected components remaining would be $\frac{n}{2^{\log \log n}} = \frac{n}{\log n}$. This would be the number of vertices in the graph we run Prim on. The number of edges would still be $O(m)$. Prim's algorithm would now take $O\left(m + \frac{n}{\log n} \log\left(\frac{n}{\log n}\right)\right) = O\left(m + \frac{n}{\log n} \log(n)\right) = O(m + n)$ time.

Thus, the total time taken would be $O(m \log \log n)$ as required, since in a connected graph, we must have $m \geq n - 1$

Rubric:

1. Part A 50 points
   - 50 points for proving the local optimum is the MST.
   - -50 points for proving every MST is a local optimum tree.
   - -10 to -20 points for minor mistakes in syntax etc.
2. Part B 50 points
   - 20 points for correct number of Boruvka's algorithm loops followed by Prim's Algorithm
   - 10 points for using DFS to compute the connected components and shrinking the vertices appropriately.
   - 20 points for runtime analysis.
   - Do not deduct points for not checking for connected graph.