EGOI 2025 Editorial - Wind Turbines

Task authors: Chur Zhe Yaw, Shi Wei Tia

July 16, 2025

Introduction

In this problem, we are given an undirected weighted graph and have to answer queries that ask for the minimum cost of connecting each node to an arbitrary node in the range $[l_i, r_i]$.

Subtask 1 $M = N - 1$ and $v_i = u_i + 1$

In this subtask, the graph is a path. For a query $[l_i, r_i]$ the optimal solution is to add all edges $(0, 1), (1, 2) \dots (l_i - 1, l_i)$ and $(r_i, r_{i+1}), \dots (N - 2, N - 1)$. To compute this cost efficiently for each query, we can precompute it using prefix sums. The runtime is $\mathcal{O}(N + Q)$

Subtask 2 $N, M, Q \leq 2,000$ and $\sum(r_i - \ell_i + 1) \leq 2,000$

In this subtask, the constraints are small enough for running a minimum spanning tree algorithm like Kruskal or Prim for each query. To account for the fact that some nodes are connected to the shore for free, we have to add a path of edges with weight 0 connecting the nodes in $[l_i, r_i]$ before running the MST (Minimum Spanning Tree) algorithm. This solution runs in time $\mathcal{O}(Q \cdot M \log M + \sum(r_i - \ell_i + 1))$

Subtask 3 $r_i = \ell_i + 1$

In this subtask, there are exactly two nodes connected to the shore for each query. That means, compared to the MST, one edge can be removed for each query. The edge that should be removed is the heaviest edge on the path from l_i to r_i. To efficiently find this edge for every query, we can use binary lifting:

First, root the tree arbitrarily. Then for every node v, and for every path length p that is a power of two, compute the heaviest edge that you encounter on the path from node v if you go p steps upwards. Also, save the node that you get to if you go p steps upwards from v. Both values can be computed in time $\mathcal{O}(N\ log\ N)$ using dynamic programming.

Then, to answer a query (l_i, r_i), we can compute the heaviest edges on the path from l_i to the lowest common ancestor (LCA) and from r_i to the LCA: we can "walk up" to the LCA in the same way as done in the standard binary lifting LCA algorithm and take the max of the heaviest edges found on the way. The runtime is $\mathcal{O}(N\ log\ N + Q\ log\ N)$

Subtask 4 $1 \leq c_i \leq 2$

This subtask can be solved using similar observations as the 100pt solution (see below), but with significant simplifications. In particular, from the size of the interval $[\ell, r]$ we know how many connections need to be payed for in total. Hence, we only need to find how many connections of cost 2 need to be used. This can also be calculated by figuring out how many connected components there are in the graph consisting of only the $c = 1$ edges for each query.

Subtask 5 $\sum(r_i - \ell_i + 1) \leq 400,000$

Suppose $q_i = [\ell_i, r_i]$ is one of the queries. The cost of this query is the cost of the MST of the original graph minus the cost of some edges. In particular, exactly the edges in the MST that joined two sets which both had a turbine from $[\ell_i, r_i]$ are the ones in the MST that we don't have to pay for. To solve the query, we just have to find the cost of those edges.

We use Kruskal's algorithm to construct MST of the input graph. We can slightly extend the union-find / disjoint-set-union data structure used by Kruskal's algorithm to compute the MST. Before we do so, we read in all the queries, and store at each node a list of queries it is part of. Since $\sum(r_i - \ell_i + 1) \leq 400,000$, we don't store too many values. When merging two sets, we can also merge the list of query indices to store at the union set. We do so by going over all the query indicies stored at the smaller set and: (1) check if they are also stored in the larger set (if so, we can record this information and update the cost saving for that query) and (2) add them to the list in the larger set. In total, this takes $O(N \log N)$ time, since whenever a vertex is processed as part of a smaller set, it will after the union operation belong to a set at least twice the size.

Alternative Solution. One can also process the each query in time $O((r_i - \ell_i + 1) \log n)$ time, using *Virtual Trees* (see e.g., `https://codeforces.com/blog/entry/140066`).

Subtask 6 $\ell_i = 0$

There are multiple ways of getting this subtask:

- Doing a simplified version of the 100pt solution (see below), where one can skip the fenwick tree since all queries start at $\ell = 0$.
- Solving offline dynamic MST, where you add 0 edges from i to $i + 1$ one at a time. See e.g, `https://codeforces.com/blog/entry/105192` for how to solve the offline dynamic MST problem.
- Using Heavy Light Decomposition, one can compute the change between the query $[0, r]$ and query $[0, r + 1]$ by figuring out which edge should no longer be used in the MST, and removing it.

We leave the details up to the reader to figure out :)

Full Solution

For full points on this problem, we need a few ideas and data structures.

Preprocessing. Let us first run Kruskal's MST algorithm on the input graph, before processing any queries.

While doing so, we can construct the following *Kruskal tree* with $2n - 1$ nodes, consisting of:

- n leaf nodes, each representing a city,
- $n - 1$ internal nodes, each representing a power line used in the original MST.

To construct this tree:

1. Sort all power lines in non-decreasing cost.
2. Process each power line in order:
    - If u_i and v_i are already connected, skip this power line.

- Otherwise, create a new internal node y and make it the parent of the roots of u_i and v_i.
- Then, set the root of u_i and v_i to y.

Observe that any power line not in the original MST will never be built in any scenario; hence we can safely ignore those power lines.

Now, let y_i be the node in the tree corresponding to power line i. It will be built if and only if y_i has at least one direct child such that all leaves in that child's subtree do not have a power plant.

In other words, power line i will **not** be built if and only if both direct children of y_i have at least one power plant in their respective subtrees.

We will also preprosses this Kruskal tree so that we can answer LCA-queries in $O(\log n)$ or $O(1)$ time.

Processing Queries. We process queries in increasing order of r_i (sweep-line). We call a (leaf) node in the Kruskal tree *active* if it has a smaller index than the current r_i we are processing.

Consider an internal node u in the Kruskal tree, let k_u be the maximum index of any active leaf in the subtree rooted at u (this is at most r_i). Also, say a and b are the direct children of u. Then the power line corresponding to u will **not** be built if and only if both $k_a \geq l_i$ and $k_b \geq l_i$. Indeed, this happens only if both the subtree rooted at a and the one rooted at b have some windtubrine in the range $[l_i, r_i]$.

We now note that:

$$k_u = \max(k_{u.0}, k_{u.1}).$$

Thus, we need to compute the sum of c of power lines satisfying the above condition efficiently. The critical observation is that this sum is equal to the following:

- the sum of $w_{u.\text{par}} - w_u$ over all u such that $k_u \geq l_i$

Here, u can be an internal node or a leaf node. $u.\text{par}$ denotes the parent of node u (if node u does not have a parent, define $w_{u.\text{par}} = 0$). If u is an internal node, w_u is the cost of the power line corresponding to u. If u is a leaf node, define $w_u = 0$.

For this, we will maintain a Fenwick Tree (or a Segment Tree) indexed by k_u to efficiently compute this sum. In particular, we will maintain, for each ℓ, the sum of $w_{u.\text{par}} - w_u$ over all u with $k_u \geq \ell$.

We can maintain the values of k_u efficiently while performing the sweep-line. When activating a leaf x, the k-values for all nodes from that leaf to the root will be set equal to the index of the newly activated leaf. Along this leaf-to-root path, there were several "chains" that share the same k-value.

Define a *chain* as a path as a maximal path from some internal node u to the leaf k_u. We say the topmost vertex on this chain is the chain's root.

When activating a leaf node x, (currently the largest activated node), this will create a new chain from the root of the tree all the way to x, and override any previous chains on this path, making them shorter.

We can observe that throughout the sweep-line on r, the total number of times chains will change roots is $O(N \log N)$. This is since, at node u with two children subtrees of size s_1 and s_2, the chain passing through u can be change direction at most $\min(s_1, s_2)$ times. The worst-case is in a balanced binary tree, where the chain at the root changes direction $N/2$ times; the chains through a node at one level below the root changes direction $N/4$ times and so on.

By the above observation we can afford to process each chain-root-change one by one (total $O(N \log N)$), and every time it happens update the affected indices in the Fenwick tree in $O(\log N)$ per update.

These chains can be maintained by simply storing the k_u values for all chain-roots, and then use LCA-queries to traverse the new root-to-leaf path downwards after an activation. Indeed, say we activate node x, then we can find the last overlapping vertex on the intersection of the root-to-x path and the chain starting at u by querying the LCA of x and k_u. (Alternatively, a Heavy-Light Decomposition or other tree data structures can help traverse these paths). We can then keep going on one of the childs of this LCA and process the next chain, while updating the Fenwick tree and chain-roots.

Time complexity: $O(M \log M)$ to sort the edge, $O(Q \log N)$ for the Fenwick tree queries and $\mathcal{O}(N \log^2 N)$ activating the leafs and updating the Fenwick tree.