Task 4: Grapevine (`Grapevine`)

Authored by: Teow Hua Jun, Jeffrey Lee

Prepared by: Jeffrey Lee, Leong Eu-Shaun

Introduction

Taking joints as vertices and branches as edges, the Grapevine takes the form of a weighted undirected tree graph. We will denote the distance between two vertices i and j as $d_{i,j}$.

Subtask 1

Limits: $N, Q \leq 2000$

Store the tree in adjacency-list format. We can maintain the tree by simply marking/unmarking vertices and updating edges for **soak** and **anneal** actions respectively. **Seek** queries can then be answered by running a depth-first search over the entire tree, for a time of $O(N)$ per query.

Time complexity: $O(NQ)$

Subtask 2

Limits: For all **seek** actions, $q_i = 1$

For this subtask, we root the tree at vertex 1. Starting from vertex 1, run a depth-first search to construct an arbitrary Euler Tour representation sequence of the tree, taking only the first occurence of each vertex such that every vertex appears exactly once. Note in particular that when any one vertex is picked, the subtree consisting of itself and all its descendants forms a contiguous subsequence in this Euler sequence. We can hence maintain an auxillary array S of the same length, such that wherever the i^{th} element of the Euler sequence is v_i, the i^{th} value of the array S is:

$$S_i = \begin{cases} d_{1,v_i}, & \text{if vertex } v_i \text{ has a grape} \\ d_{1,v_i} + 10^{15}, & \text{if vertex } v_i \text{ has no grapes} \end{cases}$$

With this array, **soak** actions become a point update to S at the target vertex, while **anneal** actions become a range add/subtract to the subtree of the target edge's lower vertex. **Seek** queries are then answered by finding the smallest element of the array S, i.e. the range minimum over all of S. We can perform all three types of query in $O(\log N)$ each using a lazy-propagation segment tree on S.

Time complexity: $O((N+Q)\log N)$

Subtask 3

Limits: The vine forms a complete binary tree, $A_i = \lfloor \frac{i+1}{2} \rfloor$, $B_i = i + 1$

For this subtask, we root the tree at vertex 1. The tree has a depth of $O(\log N)$, while each vertex has up to 2 children.

At each vertex, we initially store the shortest distance from that vertex to any of its marked (grape) descendants. We find that these stored values can be correctly maintained across any **soak** and **anneal** queries by starting at the target vertex, updating its stored value according to those of its immediate children, and repeating for its parent until all ancestors have also been updated.

We can then evaluate **seek** queries by starting from the query vertex q_i and trying the stored values of all of its ancestors, taking the minimum out of these trials. It is guaranteed that the shortest distance to a marked vertex will be produced this way: The shortest path between any two vertices in this graph consists of an ascending path from one vertex to their lowest common ancestor, followed by a descending path to the other vertex. Thus, each ancestor p_i covers the shortest paths from q_i to it entire subtree except in the direction of q_i itself, which is instead covered by p_i's child in that direction.

Each query traverses $O(\log N)$ ancestors in $O(1)$ time for a complexity of $O(\log N)$ each.

Time complexity: $O((N+Q)\log N)$

Subtask 4

Limits: There is at most 1 grape on the vine at any point in time.

Root the tree arbitrarily and construct an Euler Tour sequence as in Subtask 2. By creating an auxillary array with $S_i = d_{\text{root},v_i}$, we can handle **anneal** queries and also retrieve $d_{\text{root},v}$ for any one vertex v in $O(\log N)$ each.

The answer to a **seek** query is the length of the direct path between the query vertex q_i and the single marked vertex m. As described in Subtask 3, this path travels from q_i towards the root until it reaches the lowest common ancestor of q_i and m, where it then proceeds away from the root and to m. The distance between q_i and m can thus be expressed as:

$$d_{q_i,m} = d_{q_i,\text{lca}(q_i,m)} + d_{\text{lca}(q_i,m),m} = d_{\text{root},q_i} + d_{\text{root},m} - 2d_{\text{root},\text{lca}(q_i,m)}$$

We can find the lowest common ancestor of q_i and m in $O(\log N)$ via binary lifting, allowing us to evaluate **seek** queries using the above formula to yield a total $O(\log N)$ per query.

Time complexity: $O((N + Q) \log N)$

Subtask 5

Limits: All **soak** actions will occur before any **seek** or **anneal** actions. For all **anneal** actions, $w_i = 0$.

Prepare and maintain an Euler Tour sequence + binary lifting structure similarly to the previous subtask, in order to find the distance between arbitrary pairs of vertices quickly.

Construct a centroid decomposition on the tree, initially storing at each vertex the shortest distance from the vertex itself to any marked vertex in its covered subtree. We seek to keep these stored values updated across **anneal** and **soak** operations.

Suppose an **anneal** query is performed on an edge connecting vertices $a_i \longleftrightarrow b_i$, reducing the distance between them to 0. Without loss of generality, let vertex b_i be deeper in the centroid-hierachy tree than a_i. It follows that b_i must be a descendant of a_i in the hierachy tree; vertex a_i's covered subtree is bounded only at leaves or by its ancestor centroids, and thus contains b_i. Further, vertex a_i is the lowest-order centroid whose covered tree contains the edge $a_i \longleftrightarrow b_i$, and whose stored value may be affected by the **anneal** operation.

There are then two possibilities for the stored value in a_i after the **anneal**: either the closest marked vertex in a_i's covered subtree is now on a_i's side of the edge $a_i \longleftrightarrow b_i$, or is instead on b_i's side. In the former case, the closest marked vertex to a_i is the same as before the **anneal**, and no update is necessary to a_i's stored value.

It is the latter which needs to be evaluated to cover both cases. This is equivalent to finding the closest marked vertex to b_i within a_i's subtree, which can in turn be retrieved by using the stored values of every centroid on the hierachy-tree path from b_i to a_i. Remember in particular that the covered subtree of b_i extends outwards from the edge, terminating only at leaves and its centroid ancestors - which in turn cover more of a_i's subtree radiating away from the edge til their own ancestors. There are $O(\log N)$ vertices in the centroid tree path from b_i to a_i, each evaluated in $O(\log N)$ time from using the Euler Tour sequence's distances, for a total complexity of $O(\log^2 N)$ to update the lowest affected centroid a_i.

The remaining centroids on the path from a_i to the hierachal root can be updated using the same process - higher centroids on a_i's side of the edge are to retrieve stored values from their descendants on b_i's side, while higher centroids on b_i's side will retrieve from descendants on a_i's side starting from a_i itself. However, centroids after a_i can be updated in $O(\log N)$ each by keeping a running prefix minimum for each side, such that when iterating upwards from a_i each centroid need only apply its own stored value to the prefix in order to be accounted for by all its ancestors.

Soak and **seek** operations are classic on a centroid decomposition with these stored values, and can also be done in $O(\log^2 N)$ each.

Time complexity: $O(N \log N + Q \log^2 N)$

Subtask 6

Construct a centroid decomposition on the tree. At every centroid, we will store an Euler Tour sequence over the centroid's covering subtree, using the auxillary array value in Subtask 2. **Soak** queries can then be applied to the target centroid and its ancestors by performing the point update to each of their Euler Tour arrays; while **anneal** queries are applied by starting from the higher of the edge's incident centroids, and performing the range add/subtract on its and its ancestors' Euler Tour arrays. These take $O(\log N)$ per centroid, and $O(\log^2 N)$ in total.

The closest marked vertex in any one centroid's subtree can then be obtained in $O(\log N)$ via the range minimum, over which **seeks** can be evaluated in classic pattern in $O(\log^2 N)$.

Time complexity: $O(N \log N + Q \log^2 N)$