Romanian Master of Informatics
6th Edition, Bucharest, 10th - 13th of October 2018

Traffickers – solution

Author: Andrei Arnăutu

Naive solution (15 points)

This subtask has the additional constraint that $0 \leq t_1 \leq t_2 \leq 10$. Let $MaxTime = 10$ and let $count[u][t]$ be the number of units delivered at vertex u at time t.

For every operation that adds or removes a trafficker on the path $u \to v$, we perform these steps:

- Run a depth-first traversal from u to find the path to vertex v.
- Next, it is easy to find the vertex where the trafficker will be at every moment $t \in [0, 10]$. For every such moment we update the matrix $count$. If at time t the trafficker is at vertex x, then we increment $count[x][t]$ to add a trafficker or decrement it to remove a trafficker.
- Each such operation takes $O(N + MaxTime)$ time.

For query operations:

- Run a depth-first traversal from u to find the path to vertex v.
- For every vertex u on the path and every $t \in [t_1, t_2]$ add $count[x][t]$ to the result.
- Each such operation takes $O(N \cdot MaxTime)$ time.

This approach therefore requires $O(K \cdot (N + MaxTime) + Q \cdot N \cdot MaxTime)$ time and $O(N \cdot MaxTime)$ space.

Square root decomposition (60 points)

Let d be the number of vertices on a trafficker's path $u \to v$. If a vertex x has index p on this path (where u has index 0), then note that the trafficker delivers merchandise at moments of the form $kd + p$, where $k \geq 0$ is an integer.

Let $D = 20$ be the number of possible distinct values of d.

We redefine *count* as follows: $count[u][d][p]$ is the number of units delivered at vertex u at moments of the form $kd + p$ for $k \geq 0$ and $p < d$.

Since $1 \leq d \leq D$, *count* holds $O(N \cdot D^2)$ values.

Note that for update operations (adding or removing traffickers) we can find the path $u \to v$ in $O(D)$ time by choosing an arbitrary root and precomputing the depth and parent of every vertex.

So how do we handle update operations?

- Find the path from u to v in $O(D)$.
- For every vertex on the path there is only one value to update in *count*, so we perform $O(D)$ updates. Specifically, if vertex x has index p on the path, we update $count[x][d][p]$.
- Update operations therefore take $O(D)$ time.

Let us now rephrase query operations. Let $\mathbf{Query}(u, v, t)$ denote the number of units delivered on the path from u to v at times $[0, t]$. Then the answer to a query is $\mathbf{Query}(u, v, t_2) - \mathbf{Query}(u, v, t_1 - 1)$.

How can we efficiently calculate the number of units delivered at a vertex in an interval $[0, t]$?

- Take every value $d \in [1, D]$.
- Let $a = \lfloor t/d \rfloor$ and $b = t \bmod d$ (where $\lfloor x \rfloor$ denotes the integer part of x).
- There are now a occurrences each of numbers of the form $k{\cdot}d, k{\cdot}d{+}1, \ldots, k{\cdot}d{+}(d{-}1)$ and one more occurrence for $k \cdot d, \ldots, k \cdot d + b$).
- With this information we can calculate the answer for a single vertex in $O(D^2)$ operations.

However, for query operations we evaluate $O(N)$ vertices, which leads to a time complexity of $O(N \cdot D^2)$. To improve on this, we show how to split the tree in chunks of size $\sqrt{N}$, yielding running times of $O(\sqrt{N} \cdot D^2)$ per query and $O(\sqrt{N} \cdot D)$ per update.

Layer decomposition (flawed)

Let us apply one more reduction to simplify queries. For a path $u \to v$, let $\mathbf{QueryRoot}(u, t)$ denote the number of units delivered on the path from u to the root at times $[0, t]$. Furthermore, let l be the lowest common ancestor of u and v and let p be the parent of l. Then we can rewrite

$$\begin{aligned}\mathbf{Query}(u, v, t) = \mathbf{QueryRoot}(u, t) + \mathbf{QueryRoot}(v, t) \\ - \mathbf{QueryRoot}(l, t) - \mathbf{QueryRoot}(p, t)\end{aligned}$$

Now, let H be the height of the tree. Conceptually, we decompose the tree into **layers** of height $\sqrt{H}$, in order to obtain $O(D \cdot \sqrt{H})$ query times. Practically we achieve this by making each vertex store a pointer to its lowest ancestor in the layer above it. For vertices in the top layer, the pointer is null. We can calculate these pointers in a single depth-first traversal.

We can now answer LCA queries in $O(\sqrt{H})$. We move up the tree, following layer pointers and thus skipping an entire layer at a time, while the layer pointers differ. Once the layer pointers coincide, we move up the tree one vertex at the time. Both these phases take at most $\sqrt{H}$ steps.

For counting traffickers, we want each vertex to store unit information not just for itself, but for all its ancestors in the same layer. This allows us to answer queries by adding at most $\sqrt{H}$ values, one for each layer between u and the root.

What about adding and removing traffickers at a vertex u? Obviously u is an ancestor to all its descendants in the same layer. As discussed, we need to notify these descendants so that they too can store the new value. Therefore, we need to run a depth-first traversal from u to the bottom of its layer and increment or decrement the unit counts on every vertex.

This can still lead to $O(D^2 \cdot N)$ queries when the tree is very flat.

Block decomposition

In order to guarantee $O(D^2 \cdot \sqrt{N})$ queries, we need to replace layers with a different concept. We partition the tree into $\sqrt{N}$ **blocks**, each having at most $\sqrt{N}$ vertices, such that each block is connected. This may sometimes be impossible to do, for example if the root has $n - 1$ children. To address this, we allow the addition of **supplementary vertices**.

In the case of one root with $n-1$ children, we would group every $\sqrt{N}$ children under a supplementary vertex u' and make u' a child of the root. More generally, we can partition the tree in a single depth-first traversal. Every vertex u:

1. calls its children recursively;
2. whenever the total number of vertices in subtrees seen so far exceeds $\sqrt{N}$, groups those children under a supplementary vertex and makes that vertex a child of u;
3. at the end, returns the number of descendants not distributed to a supplementary vertex (including u itself).

Supplementary vertices now partition the tree into blocks. Since each block has size at least $\sqrt{N}$ (the above algorithm can wait until there are $2\sqrt{N}$ undistributed vertices), it follows that there are at most $\sqrt{N}$ blocks. Since the structure is still a tree, it follows that there are at most $\sqrt{N}$ supplementary vertices connecting the blocks, so the size of the problem does not increase significantly. Also, in contrast with the previous solution, each vertex has at most $\sqrt{N}$ descendants in its block.

We now apply the same algorithm as above, with three minor changes:

1. Block pointers point to the nearest supplementary ancestor.
2. The LCA of two vertices can be a supplementary vertex. If that is the case, its parent is the true LCA.
3. When adding traffickers on the path from u to v, we must ignore all supplementary vertices along that path.

This solution takes $O(K \cdot \sqrt{N} \cdot D + Q \cdot \sqrt{N} \cdot D^2)$ time and $O(N \cdot D^2)$ space.

Euler tour and Fenwick trees (100 points)

We linearize the tree into an array in $O(N)$ using an **Euler tour**. Specifically:

- Choose an arbitrary root and run a depth-first traversal. When first entering a vertex u, add it to the array and store its position as $\mathit{first}[u]$. When leaving u, add it to the array once more and store its position as $\mathit{last}[u]$. For simplicity, we index the array starting at 1.
- Note that the resulting array has at most $2N$ elements. For leaves, both occurrences happen in succession and we have the option of storing just one of them.

- We can run path queries on the resulting array, from an arbitrary vertex u to the root, by querying the array prefix up to $\mathit{first}[u]$.

- Updates to a vertex u are added to the array at position $\mathit{first}[u]$ and subtracted at position $\mathit{last}[u]$.

We store D two-dimensional Fenwick trees, one for each value $d \in [1, D]$. If the length of the linearized array is S, then each Fenwick tree has size $d \times S$. Thus, if a trafficker visits vertex u at times of the form $k \cdot d + p$, we update the d-th Fenwick tree at positions $(b, \mathit{first}[u])$ and $(b, \mathit{last}[u])$. A Fenwick tree update will take $O(\log D \log N)$.

In order to answer LCA queries in O(1), there are two approaches:

1. Tarjan's offline LCA algorithm, which precomputes all the answers in $O(M + N)$.

2. A range-minimum query precomputation in $O(N \log N)$, which then permits online LCA queries in O(1).

Thus, steps required for adding/removing traffickers are:

- Find the path $u \to v$ in $O(D)$.

- For every vertex along the path, update two positions in the Fenwick tree.

- The total cost is $O(D \log D \log N)$.

Steps required for (u, v, t) queries are:

- Find the LCA of u and v in O(1).

- Answer four $\mathbf{QueryRoot}(u, t)$ queries.

- For each root query, check all D Fenwick trees.

- The total cost is $O(D \log D \log N)$.

The overall running time is $O(M + (K + Q) \cdot D \log D \log N)$ and the memory usage is $O(N \cdot D^2)$.

Heavy path decomposition (100 points)

A similar solution uses **heavy path decomposition** to split the tree into paths such that any path query traverses at most $O(\log N)$ different paths. For every path we will store D Fenwick trees in similar fashion.

Although the theoretical complexity increases to $O((K+Q) \cdot D \log D \log^2 N)$, in practice this approach behaves well and can obtain the full score.