'The Potion of Great Power' Solution

Topics

graph representation, sqrt decomposition, sorting, binary search

Problem author

Márton Erdős

Subtask 2

$Q, U \leq 1000$

We can store each operation, and replay them for each question. We can store all edges in a data structure, or keep neighbours in separate data structures for each node.

When answering a question, find the neighbours of both nodes, and consider the distance of each pair of neighbours, computing the minimum. This yields to a simple solution in $O(QUD+QD^2)$ time, or $O(QU\log(D)+QD^2)$ when using an associative container, like `std::set`.

Ordering Trick

Although not required for this subtask, we can improve the second part by ordering the neighbour lists by H value, and stepping through them simultaneously, using two pointers. We always step in the list in which the pointer points to the entry with the smaller H value out of the two, and consider the current pair for the minimum computation:

```
p1 = 0, p2 = 0
while (p1 < l1.length) and (p2 < l2.length)
        consider(H[l1[p1]], H[l2[p2]])
        if (H[l1[p1]] <= H[l2[p2]])
                then p1++
                else p2++
```

The correctness of this is easy to prove.

Subtask 3

$V = U$ for all queries

In this case, each question will refer to the same version (the final one). Hence, we can just apply all updates at the start once (applying updates is done in the same way as in *Subtask 2*), and then answer questions on this single version (like before). Using an efficient data structure and the ordering trick (we can actually pre-sort, before answering any questions), we can solve this subtask in $O(U\log(D) + M\log(D) + QD\log(D))$.

Subtask 4

$H[i] \in \{0, 1\}$ for all nodes i

For a node u and version V, we need to be able to tell whether u had a neighbour u' with $H[u'] = f$ for both possible values of f (0 or 1). Once we have obtained this information for both X and Y, we can easily work out the answer.

For each node u and possible f value, let us build an ordered list of 'events' of the following types:

- Node u *stopped* having any neighbours u' with $H[u'] = f$ in version V.
- Node u *started* having neighbours with $H[u'] = f$ in version V.

These lists can be built in $O(U \log(M))$ time by iterating through all updates (once), and keeping track of the number of neighbours u' with each f value for each node (the logarithmic factor comes in from keeping track of which edges are active). Once built, we can binary search for the last event (or the parity of the number of events) for both X and Y, for both $f = 0$ and $f = 1$ to get all the necessary information to answer the queries.

This yields a solution in $O(U \log(M) + Q \log(U))$ time and $O(U)$ space for this special case.

Subtask 5

$U, N \leq 10000$

$\sqrt{U}$ checkpoints

We first apply all updates in order, producing $\sqrt{U}$ checkpoints, evenly spaced, then – for each question – we simulate updates from the closest checkpoint (in the same way as we did for *Subtask 1*). An efficient implementation of this can achieve $O(U \log(M) + \sqrt{U}M + Q\sqrt{U} \log(M) + QD \log(D))$ time and $O(\sqrt{U}M)$ memory.

Save neighbour list by node, binary search by version

Another solution is to separate updates by node, and save the neighbour list of the updated node for each update (in a vector of neighbour lists for that node). Then, we can binary search for the neighbour list at version V. This can be implemented in $O(UD + QD)$ time and $O(UD)$ space.

Self-copying data structures

We can use a self-copying (also known as persistent or copy-on-write) data structure. These data structures are constructed as a (directed) tree, where each node holds some information.

When updates are applied, we copy every node that was modified, including nodes whose children are modified, thus each version will have its own root node, from which queries can be performed. There are multiple possibilities here from static binary trees (holding neighbour lists, or neighbours directly) to balanced binary search trees (e.g. treap). These will solve this subtask, but will struggle to gain full marks for the problem due to exceeding memory constraints: the static versions have a high complexity ($O(ND + U \log(ND))$ space, or worse), and the BST version has very high constants (both in space and time).

One segment tree

We need to find when each edge is present in the graph. For each edge, this is the union of contiguous intervals. In total, we have at most U intervals.

Take a segment tree of length U, with one leaf per version. Each node will contain a vector of edges, ordered by starting node (edge e is stored in the tree node for $[a, b]$, if edge is present in each version during the interval $[a, b]$ – and not entirely present in the parent's interval). This has a combined space requirement of $O(U \log(U))$. For each question, we look up edges starting from X and Y using binary search in the vector of each of the $\log U$ relevant tree nodes.

This can be implemented in $O(U \log^2(U) + Q \log^2(U) + QD \log(D))$ time and $O(U \log U)$ space. With a few tricks and an efficient implementation, this solution could possibly pass subtask 6.

Subtask 6

No additional constraints

Save neighbour list by node, reduce storage space by constant factor

Clearly, saving each version of the neighbour lists of each node will not fit in memory for these limits. However, since memory limits are very generous, we can cut this down by a reasonably small constant C: we only save the neighbour list of node u after every C'th update *that affects* u. We also save an ordered list of updates affecting each node. For each question, we only need to replay at most $2C$ updates (C for X and C for Y). Choosing $C \approx 50$ will suffice the pass every subtask. The time complexity is $O(U \log D + \frac{UD}{C} + Q(D + C \log C))$, using $O(\frac{UD}{C})$ memory.

We think this solution is interesting in the sense that it demonstrates how big-O complexity can often be misleading for real-world problems: this solution is far simpler than other solutions, some of which do not even gain 100 marks, and it passes the limits because the constants involved (normally hidden by big-O complexity classes) are far smaller.

Multiple segment trees

We can eliminate the binary search from the previous segment tree solution, by keeping a separate segment tree for each node. Naïvely, this will not fit in the memory limits. However, we can save space by only keeping a leaf for each update where the given node was affected. This yields a solution in $O(U \log(U) + Q \log(U) + QD \log(D))$ time and $O(U \log(U))$ space.