PROGRAMMING PROJECT (ASSIGNMENT 5) - INF432

January 2023

Artur César Araújo Alves
Ksenia Khmelnikova

SOMMAIRE

1 OVERVIEW OF THE PROBLEM

We have a connected and undirected graph $G = (V, E)$. For every edge $e \in E$, there is a non-negative integer value indicating the noise level at the edge e. For a pair (u, v) of vertices, a most pleasant itinerary between u and v is a u-to-v path in the graph G such that the maximum noise level of all edges along this path is minimized.

We have a set of queries, each containing a pair of vertices (u, v) from V. The task is to compute the maximum noise level on a most pleasant itinerary between u and v for each query. Below we discuss different ways to solve this problem and also provide **a link to our github repository** containing C++ implementations to all coding tasks.

2 TASK 1

Let $M = (V, E')$ be a minimum spanning tree built from the original graph $G = (V, E)$. We show that the answer for a query (u, v) must be the same for both graphs.

By contradiction, assume that this assertion is false, i.e., there is an alternative path P_{alt} in G that connects u and v and has a smaller maximum noise level. This is only possible if the maximum noise edge in the path from u to v in the MST, call it e, is strictly greater than all of the edges in P_{alt}.

Now we claim that it is possible to construct a spanning tree with less total weight than M by replacing e with another edge e' from the alternative path.

To prove this claim, we proceed as follows :

1. $(V, E' \setminus \{e\})$ has exactly 2 connected components, one of them containing u and the other one v.
2. P_{alt} must contain at least one edge connecting vertices from different components, name it e'. Otherwise, all vertices in the alternative path would be in the same component (u and v in particular).
3. $M' = (V, \{e'\} \cup E' \setminus \{e\})$ is a connected graph by construction and since its edge-vertex balance remains the same as M we now it also follows the equation $|Vertices| = |Edges| + 1$. These both elements characterize a tree and since it contains all vertices of G, M' is also a spanning tree.

4. By assumption $c_{e'} < c_e$, M'. We conclude that M' has a lower total weight than M.

Finally, since M is a minimum spanning tree, this is clearly a contradiction and it concludes the proof.

3
TASK 2

As discussed in the first task, we can operate entirely on an MST generated from the original graph and still get the correct answers and we did so with `Kruskal`'s algorithm in all of the tasks from now on.

To write our program **itineraries_v1** that naively answers each query (u, v) we run a BFS starting at u where we keep track of the current maximum noise level seen throughout the path. Once v is reached, we return the corresponding max noise level.

The correctness of this approach rests on the fact that there is only one path between u and v (we run it in a tree) which is indeed found in the BFS traversal and it's complexity is linear for each query since we just perform an ordinary BFS.

4
TASK 3

In this task we write the program **itineraries_v2** that reduce the query time to $O(\log n)$. Thus, we solved the two subproblems, and the complexity of every problem is $O(\log n)$ time. First of all, we found the LCA of two vertices u and v in the tree T. And after we computed the maximum noise level on the path between a vertex u and LCA for two vertices u and v and the maximum noise level on the path between a vertex v and LCA the tree T. In the end we chose the maximum of these two values.

During the preprocessing we make BFS and after memorize for every vertex u the 2^i-th ancestor and the maximum level noise between u and his ancestor 2^i-th. The complexity of this algorithm is $O(\log n)$.

In conclusion, the preprocessing time of the following algorithm is $O(n \log n)$ and afterwards the time to answer each query is $O(\log n)$ as we'd like to make.

```
//search the 2^i-th ancestor and the max noise
//between this anestor and the initial vertex
//up[][] is the array of the ancestors
//noise[][] is the array where we memorize the maximum noise
void search_ancestor_and_maxnoise(int n)
{
    up[0][0] = 0;
    for(int j = 1; j < (int) log(n) - 1; j++)
    {
        for(int i = 0; i < n; i++)
        {
            up[j][i] = up[j - 1][up[j - 1][i]];
            noise[j][i] = max(noise[j - 1][i], noise[j - 1][up[j - 1][i]]);
        }
    }
}
```

Figure 1 – Preproccesing - the goal is to reduce the query time to $O(\log n)$

5 TASK 4

In this task, we were proposed to use Tarjan's algorithm to improve the time complexity of the second program, by writing **itineraries_v3**. Tarjan's original LCA algorithm, as stated in the project description, is built on the disjoint-set data structure and is able to compute the smallest common ancestor of each pair (u, v) while performing a DFS traversal.

5.0.1 • Algorithm Design

In order to use Tarjan's LCA to compute the actual (u, v) queries, i.e., the maximum noise in the $u \to v$ path, we need the following :

1. Tree is rooted in some vertex (1 by default)
2. Array $par[v]$: stores the parent of v ($par[1] = 1$)
3. Array $noise[v]$: initially, store the value of the edge connecting v to $par[v]$ ($noise[1] = 0$)
4. DSU function $find(v)$ that implements path compression to also update $len[v]$ to the maximum edge value in the path between v and the head of its set
5. Set $LCA^{-1}[z]$: stores a list of pairs (u, v) belonging the query list such that $z = LCA(u, v)$

Note that the original algorithm calculates $z = LCA(u,v)$ for a given pair of vertices (u,v) just after post traversing one of them, say u. Now observe that at this moment, the new $find$ function would also calculate the maximum noise between u and z which we can store in $noise[u]$. To finally calculate the query (u,v) we just have to run $find$ in the second vertex v of the pair right after post traversing z : it gives us similarly $noise[v]$ as the maximum noise in the $v \to z$ path. As a consequence, the answer for each query would be just $max(noise[u], noise[v])$ if $u \neq z$ or $noise[v]$ otherwise.

Algorithm 1 Modified Tarjan's LCA algorithm

```
1 : function TarjanMaxNoise($u$)
2 :     MakeSet($u$)
3 :     for each child $v$ of $u$ linked by an edge of noise $w$ do
4 :         TarjanMaxNoise($v$)
5 :         $v$.parent $\leftarrow u$
6 :         $v$.noise $\leftarrow w$
7 :     end for
8 :     $u$.visited $\leftarrow true$
9 :     for each $v$ such that $(u,v) \in P$ do
10 :        if $v$.visited then
11 :            the LCA of $(u,v) \leftarrow$ Find($v$).parent
12 :            $LCA^{-1}$[LCA of $(u,v)$].insert($u,v$)
13 :        end if
14 :    end for
15 :    for each $(x,y)$ in set $LCA^{-1}[u]$ do
16 :        Find($x$)
17 :        if $y$ and $u$ are the same then
18 :            the answer to query $(u,v) \leftarrow x$.noise
19 :        else
20 :            the answer to query $(u,v) \leftarrow max(x$.noise$, y$.noise$)$
21 :        end if
22 :    end for
23 : end function
```

5.0.2 • Algorithm Complexity

The efficiency of **itineraries_v3** critically depends on the performance of `Find`. Without this function, one could easily see that it has a linear complexity since the algorithm would do no better than an ordinary DFS.

The total number of iterations that `Find` does is limited by the sum of the distances between the LCAs and the associated vertices present in P. We show now that `Find` never "travels" the same path twice :

Suppose we have two vertices u, v, each belonging to a different pair in P that is associated to the same LCA z, and suppose additionally that v is in the path $u \to z$. Observe that in this situation u is necessarily post traversed before v and as consequence whenever we call $\texttt{Find}(u)$ it will also do the job of $\texttt{Find}(v)$ and by recurrence will link both vertices to z with the correct maximum noise in each vertex's path (path compression). The next call of $\texttt{Find}(v)$ will then demand only one iteration.

As a consequence, the addition of this modified $Find$ function adds up to $O(m + l)$ complexity, where m and l are respectively the number of edges and queries. Therefore, the total complexity of **itineraries_v3** is linear which also means constant complexity for each query.

6 RUNTIME PERFORMANCES

For each given input and program version, we calculate the corresponding running time in the same machine with bash's built in command "time". Bellow we present the results by taking the mean of 10 executions values for each combination of program and input lasting at most $30s$ (∞ otherwise) :

№	**itineraries_v1(s)**	**itineraries_v2(s)**	**itineraries_v3(s)**
0	0.09	0.0187	0.0375
1	0.010	0.022	0.038
2	∞	0.8989	0.6716
3	∞	3.2693	2.6003
4	∞	3.268	2.8675
5	∞	1.6166	1.1877
6	∞	4.2358	2.9515
7	∞	3.6364	3.0026
8	∞	3.7991	3.1381
9	∞	3.6378	2.9502

As we can see, the results for the largest inputs are in agreement with the theoretical complexities calculated for each algorithm version, where the last one was effectively the fastest.

It is important to note that all three programs include the step where the MST is extracted from the graph with Kruskal's algorithm, which lower bounds the overall complexity of all algorithms to $m \log(n)$ even though that of Tarjan's LCA alone is linear. In other words, without this step we would have seen a significantly better performance of **itinerary_v3**.