PROGRAMMING PROJECT
INF421

Most Pleasant Itineraries

January 22, 2018

Christopher Aykroyd,
Ramon Pereira Sena

ÉCOLE POLYTECHNIQUE
UNIVERSITÉ PARIS-SACLAY

SUMMARY

Abstract

Paris is a beautiful city with great charm. Unfortunately, local Parisians complain that many paths in the city are very noisy because of tourists or vehicles. As a result, when they travel inside the city, they do not necessarily take shortest itineraries, but instead, more pleasant ones, i.e., itineraries with less noise. For the following, Let $G = (V, E)$ be a connected and undirected graph representing the map of the city. For every edge $e \in E$, there is a non-negative integer value w_e indicating the noise level at the edge e. For a pair (u, v) of vertices from V , we define a most pleasant itinerary between u and v as a u-to-v path in the graph G such that the maximum noise level w_e of all edges e along this path is minimized.

1

THEORETICAL VIEW

Consider any minimum spanning tree T of G. We will show that for any query (u, v), the answer to this query (i.e., the maximum noise level on a most pleasant itinerary between u and v) in the graph G is the same as the answer to that query in the tree T.

Lemma: If C is a cycle in G and M the maximum noise along C, then there's an edge e of $C - T$ whose noise level is M.

Proof of lemma: Since T is a tree, let us define $(e_1, e_2, ..., e_m)$ the edges of C not in T. We also define x_i and y_i vertices such that $e_i = (x_i, y_i)$. By contradiction, suppose that $\forall i \in [1, m]$, the weights $|e_i| < M$. Then by definition of M, $T \cap C \equiv (t_1, t_2, ..., t_p) \neq \emptyset$, and there exists an edge $e_q = (x_q, y_q)$ such that the path $P(x_q, y_q)$ from x_q to y_q in T contains $(t_1, t_2, ..., t_p)$: if not, $(t_1, t_2, ..., t_k, P(x_1, y_1), ..., P(x_m, y_m))$ would form a cycle in this tree (Figure 1). In this case, the graph T' produced by replacing any edge t_j, $|t_j| = M$ with e_q, in T, will produce no cycles. Since the number of edges in T' is still $|V| - 1$, T' is a spanning tree, and the total weight of T' must be smaller than T. Contradiction.

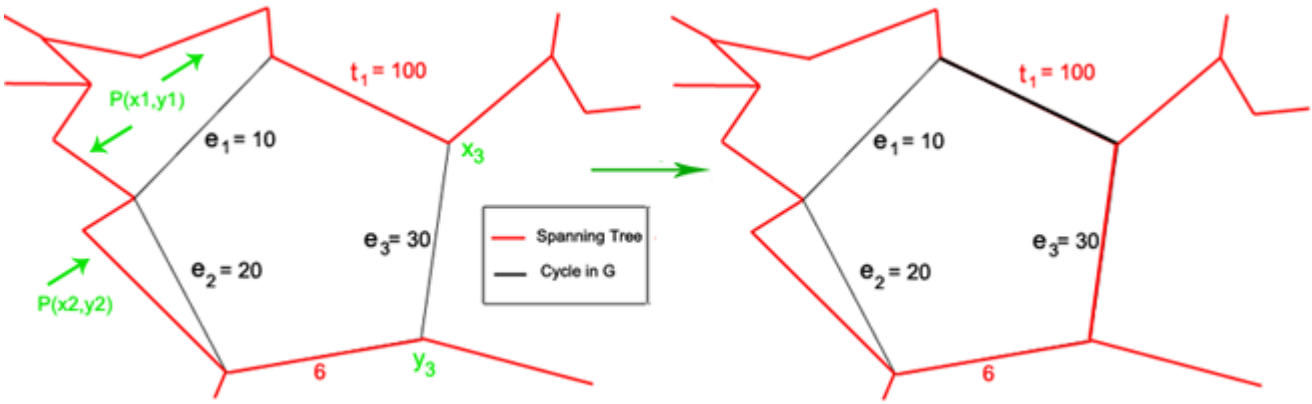

Figure 1: This is an hypothetical MST containing the maximum edge (t_1) of a cycle in G. If we want to go from x_3 to y_3, (i.e, $P(x_3, y_3)$), we will pass a cycle edge. Then, e_3 is the only edge that can replace t_1 keeping the red line as a spanning tree. Note that if e_3 were "closed" as the others, we would have a cycle.

Corollary: Consider two points u and v of G and P_{uv} the unique path from u to v in T (passing through each edge at most once). Then, P_{uv} is a *most pleasant itinerary* of G.

Proof: First of all, P_{uv} exists since T is a spanning tree. If P'_{uv} is a different path from u to v, in G, then, $P' \cup P$ is a cycle and by the lemma $\max_{e \in (P \cup P')} \omega_e \notin T$. As $P \subseteq T$, $k = \max_{e \in (P \cup P')} \omega_e \in P'$. Then, $\max_{e \in P} \omega_e \leq \max_{e \in P'} \omega_e = k$. Thus, P is a most pleasant itinerary.

2

PRACTICAL VIEW

Based on the above observation, we have written three programs to compute multiple subsequent queries in optimized time complexity.

2.1 First Approach

We begin by subdividing our problem into two steps. At the first step, through means of the *Union-Find* data structure, we apply *Kruskal*'s algorithm to compute a *Minimum Spanning Tree* once, with time complexity of $\theta(|E| * log(|E|))$. By taking advantage of the spanning tree, we can subsequently compute any query (u, u) in linear time by advancing *upwards* through the tree until reaching the *Least Common Ancestor* of u and v.

2.2 Implementation Details

To further detail our code, we summarize its implementation below. Our project was subdivided in three packages, namely, *graph*, *tree* and*default*.

- The *graph* package was the most general in that it handled general cycled graphs. The *Graph* class was used to wrap it up as collections of edges or vertices.
- The *tree* package contained two classes, *Tree.java* and *Node.java*, responsible for transforming any *Graph* object into a *MST* and its respective tree-like data structure.
- Finally, the default package contained any stray classes, such as the static *LCA* class, responsible for applying all *Lowest Common Ancestor* algorithms to the trees. The auxiliary *Query* class was used as a function argument to pass on queries and retrieve their numerical results.

2.3 Optimization Through Exponential Steps

To further improve the time complexity of our algorithm, we add a second preprocessing step before beginning to compute the queries. The main idea is to save, for each node u in T, all ancestors distant

[1]We define the LCA as the deepest vertex that has both u and v as descendants.

of 2^n, as well as the noise between u and this ancestor. The algorithm was implemented using Depth-first-Search in the tree, going through each node exactly once, computing the powers-of-two and noise arrays each time. Whenever we reach a leaf (i.e., the last node of a branch), we use a pre-saved array representing the full branch (i.e., the path between root and leaf) to calculate, for each node in the branch, the desired arrays of ancestors 2^n and their noise in log complexity. By using the principles of *dynamic programming* we in this way achieved a total complexity bounded by $O(n.log(n))$.

The noise calculation of the powers-of-two arrays is based on the following recurrence formula. Let $u[.]$ be an array representing a full branch in the *Minimum Spanning Tree*, and $w(a, b)$ the weight between two neighboring elements. Then, the noise between u_i and an ancestor u_{i-2^j} distant 2^j is given by:

$$Noise(u_i, u_{i-2^j}) = \begin{cases} max(Noise(u_i, u_{i-2^{j-1}}), Noise(u_{i-2^{j-1}}, u_{i-2^j})), \text{if j ¿ 0} \\ w(u[i], u[i-1]), \text{if j} = 0 \end{cases}$$

We are assured that if the powers-of-two array has already been computed for a certain node, all nodes with smaller depth also have had their arrays computed. In this way, we can compute, for growing j, the noise between all the powers-of-two ancestors.

Calculating the queries in logarithmic time resumes to finding the *Lowest Common Ancestors* in steps of size 2^n while simultaneously keeping track of the maximum noise levels. To this end, our algorithm uses the powers-of-two arrays to, given a query (u, v), such that $depth(u) > depth(v)$:

1. Find w, such that $depth(w) = depth(v)$, using exponential steps 2^n with n decreasing each step.
2. Knowing that the pair of nodes (w, v) possess the same depth, recursively jump to the furthest pair of 2^n ancestors (most distant) **that are still on different branches of the tree**. This way, our n-sized steps will gradually reduce until our final pair of nodes are direct children of the *Lowest Common Ancestor*.

2.4 Tarjan's Algorithm

Our final approach utilizes a well known algorithm, *Tarjan's Algorithm*, to further reduce time complexity. A peculiarity of Tarjan's algorithm is that it requires that all queries (u, v) to be specified in advance.

Tarjan's algorithm makes strong use of the *Union-Find* data-structure. Using path compression, the *union* and *find* operations can be run in amortized constant time.

To quickly access the queries involving a particular node, we used an array of linked-lists of queries (u, v), which was preprocessed with each position in the array representing the id of the node (similar to a HashTable). In this way, at the end of each call of *TarjanLCA(u)* we could simply shuffle through the list of queries relating to u.

Since Tarjan's algorithm goes through each node exactly once, we could compute the LCA's of all queries at once with complexity of roughly $O(n+q)$, being n the number of nodes and q the number of queries. As the value of each query is saved during the execution it can later be accessed in constant time.

2.5 Summary of Running Time

As expected, the running time of the first algorithm was bigger than the two others. Tarjan's algorithm seemed to have similar durations to the optimized exponential algorithms. We should note, however, that the times here disclosed include the generation of the MST through means of the Kruskal algorithm (which was in general many orders of magnitude higher than other preprocessing phases). As a rule, all algorithms satisfactorily run in a time shorter than 7 seconds.

```
Input               itineraries_v1       itineraries_v2     itineraries_v3
itineraries.0.in... 14 milliseconds  | 5 milliseconds   | 6 milliseconds
itineraries.1.in... 9 milliseconds   | 52 milliseconds  | 36 milliseconds
itineraries.2.in... 1514 milliseconds| 1196 milliseconds| 1305 milliseconds
itineraries.3.in... 5027 milliseconds| 3713 milliseconds| 4151 milliseconds
itineraries.4.in... 5505 milliseconds| 4447 milliseconds| 4320 milliseconds
itineraries.5.in... 2158 milliseconds| 1573 milliseconds| 1367 millisecond
itineraries.6.in... 6531 milliseconds| 4257 milliseconds| 4356 milliseconds
itineraries.7.in... 7084 milliseconds| 4735 milliseconds| 4763 milliseconds
itineraries.8.in... 7004 milliseconds| 5459 milliseconds| 5107 milliseconds
itineraries.9.in... 6078 milliseconds| 4005 milliseconds| 4776 milliseconds
```