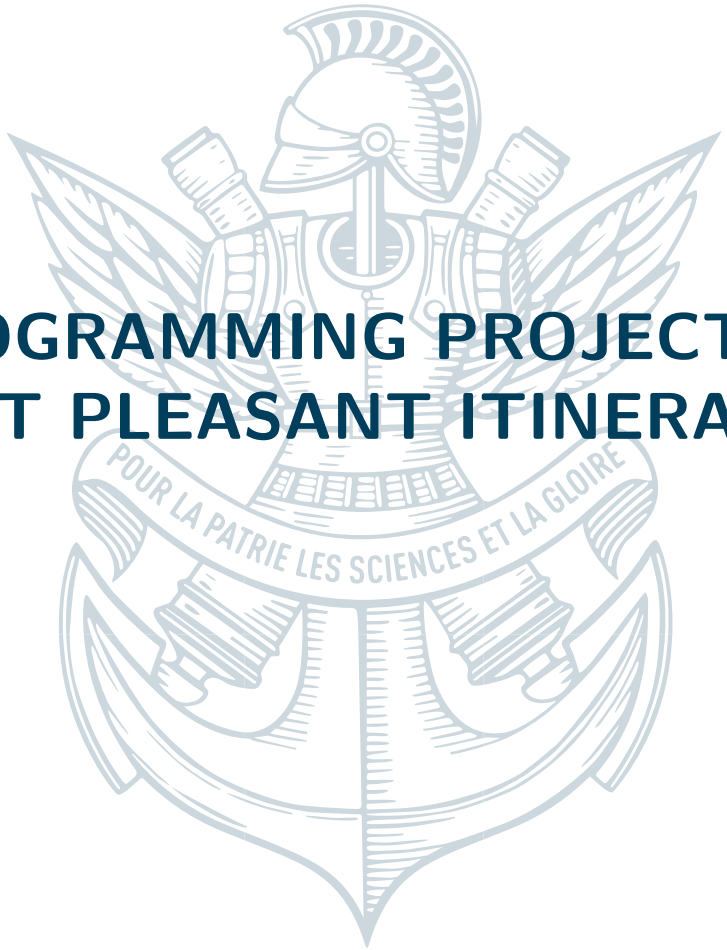

PROGRAMMING PROJECT 5 - MOST PLEASANT ITINERARIES

1er février 2023

Ziad Eddine OUMZIL
João Pedro SEDEU GODOI

ÉCOLE POLYTECHNIQUE

IP PARIS

Please click here to access the GitHub of the project

SUMMARY

1 TASK 1

Please click here to access the GitHub of the project

First, we are going to define some important elements for the problem.

We consider a connected and undirected graph $G = (V, E)$, $T = (V, E_T)$ any minimum spanning tree of G and $u, v \in V$ two distinct vertices. Let $MST(u, v)$ be the unique path from u to v in the tree T

$$u \xrightarrow{f_1} a_1 \xrightarrow{f_2} a_2 \xrightarrow{f_3} a_3 \xrightarrow{f_4} ... \xrightarrow{f_{m-1}} a_{m-1} \xrightarrow{f_m} v$$

Where $a_1, a_2, ..., a_{m-1}$ are the vertices of G, $f_1, f_2, ..., f_m$ are the edges and $|f_1|, |f_2|, ..., |f_m|$ are their weights, respectively.
We suppose by absurd that this path is a most pleasant itinerary between u and v.
So let consider any most pleasant itineraries between u and v.

$$u \xrightarrow{e_1} u_1 \xrightarrow{e_2} u_2 \xrightarrow{e_3} u_3 \xrightarrow{e_3} ... \xrightarrow{e_2} u_{n-1} \xrightarrow{e_n} v$$

Where $u_1, u_2, ..., u_{n-1}$ are all are distinct vertices of G, $e_1, e_2, ..., e_n$ are the edges of G and $|e_1|, |e_2|, ..., |e_n|$ are their noisy respectively.

By assumption (absurd hypothesis), we have : $\max_{k\in\{1,...,m\}} |f_k| > \max_{k\in\{1,...,n\}} |e_k|$
let $j \in \{1, \dots, m\}$ such that $|f_j| = \max_{k\in\{1,...,m\}} |f_k|$. Therefore $\forall k \in \{1, \dots, n\}$ $|f_j| > |e_k|$ (1).

We are going to look at the graph $T' = (V, E_T \bigcup \{e_1, e_2, ..., e_m\})$. (We just added the edges e_k to the tree T).
T' is a subgraph of G and a connected graph, since its subgraph $T = (V, E_T)$ is a minimum spanning tree.
T' contains two different paths from u to v, hence it contains a cycle.

$$u \xrightarrow{f_1} a_1 \xrightarrow{f_2} a_2 \xrightarrow{f_3} a_3 \xrightarrow{f_4} ... \xrightarrow{f_{m-1}} a_{m-1} \xrightarrow{f_m} v \xrightarrow{e_n} u_{n-1} ... \xrightarrow{e_2} u_1 \xrightarrow{e_1} u \quad \text{(cycle 1)}$$

We want to remove all cycles by deleting some specific edges of T' in order to get a new tree that contradict the fact that T is an MST.
We start by removing the edge f_j from the (cycle 1), and then : while the new T' still contains a cycle, this cycle will contain at least one edge e_k that was not initially in the tree T. We remove it.
Let T" be the tree that we get after this algorithm. By construction, the only edge of the initially T that has been removed, was f_j. Which means that f_j was swapped with an edge e_p that wasn't in T, hence $E_{T''} = E_T \bigcup \{e_p\} \setminus \{f_j\}$. By (1), we have $|f_k| > |e_p|$

$$w(T'') = \sum_{edge \in E_{T'}} |edge| = w(T) - |f_k| + |e_p| < w(T)$$

.
Therefore, T'' is a spanning tree with a smaller weight, which contradict the assumption that T is an MST.

2 TASK 2

We compute an MST of the graph using the **Kruskal**'s algorithm, which compute an MST in a time complexity of $nlog(n)$. To answer a query (u, v), we start in the vertex u, and we do a breadth first search. We make sure that in our BFS, we save the weights in our path while doing our search. As we visit each vertex at most one time, the complexity of the algorithm is $O(n)$.
For the code, please click here.

3 TASK 3

For the code, please click here.

The algorithm proposed in the **TASK 3** is based on the binary Lifting method. The three first functions in the code are initializing :

1. the *depth* list.
2. the *parent* array : which is an array such that for each vertex u and $0 \leq i < log(n)$, $parent[i][u-1]$ is the ancestor of u that is at distance $h = 2^i$ from u
3. the *noise* array : which is an array such that for each vertex u and $0 \leq i < log(n)$, $noise[i][u-1]$ is the maximum noise on the path between u and $parent[i][u-1]$.

The complexity of initializing the *parent* **and** *noise* **array is in :** $O(nlog(n))$ (because of the two for-loop)

After the initialization, the function $itineraries - v2$ takes a query (u, v), and starts by defining a variable m, which will contain the result. If u and v are not in the same depth, the deepest vertex between them will be moved up until both become in the same level, which is in done in $O(log(n))$ (First loop).
Note, that in our way, we always update m. The second loop, will find the LCA of u and v, but this time both of the vertex will be moved up, the condition $self.parent[i][u-1] != self.parent[i][v-1]$ and the new assignments, will make sure that for each step i in the loop, the distance of u and v to their LCA is less than 2^i. As the loop, is a decreasing loop, in the end we have that ; the distance of u and v to their LCA is less than $2^0 = 1$. Hence, we got the LCA. The loop does this work in $O(log(n))$. **the answer to a single query can be done in** $O(log(n))$ Bellow we show two functions : the first one does the initialization, and the second one, computes the maximum noise,

```
#
class Task3:
    ...
    def fill_in(self):
        for i in range(1,self.log):
            for v in range(self.n):
                if self.parent[i-1][v-1] !=-1:
                    self.parent[i][v-1] = self.parent[i-1][self.parent[i-1][v-1]-1]
                    self.noise[i][v-1] = max(self.noise[i-1][v-1],self.noise[i-1][self
                        .parent[i-1][v-1]-1])


    def itineraries_v2(self,u,v):
        m=0
        if self.depth[u-1] < self.depth[v-1]:
            u, v = v, u
        for i in range(self.log):
            if (self.depth[u-1] - self.depth[v-1]) >> i & 1:
                u,m = self.parent[i][u-1], max(m, self.noise[i][u-1])
        if u == v:
            return m
        for i in range(self.log - 1, -1, -1):
            if self.parent[i][u-1] != self.parent[i][v-1]:
                u,v,m = self.parent[i][u-1],self.parent[i][v-1],max(m,self.noise[i][u-1],
                    self.noise[i][v-1])
        # the Lowest common ancestor is given by parent[0][u-1]
        return max(m,self.noise[0][u-1],self.noise[0][v-1])
```

4
TASK 4

For the code, please click here.

We introduce our arguments and variables :

1. *parent* : the list of parent in our disjoint-set method.
2. *noise* : stores the noise
3. array $queries[u-1]$: a list that stores couples (v, i) such that (u, v) is the i^{th} query.
4. l : the number of queries
5. array LCA : $LCA[u-1]$ is a list of couples (x, y) such that the common ancestor of x and y is u

In this task, we were proposed to use Tarjan's algorithm to implement the third version of *itineraries*.
We start by defining our function $find$: You can see that while doing the find, we update the noise. So at the end, $noise[v-1]$ will contain the maximum noise to its class representative.

Algorithm 1 Find Method

```
1:  function FIND(parent, noise, v)
2:      if parent[v − 1] == v then
3:          return v
4:      else
5:          p ← parent[v − 1]
6:          r ← FIND(parent, noise, p)
7:          if p ≠ r then
8:              noise[v − 1] ← max(noise[p − 1], noise[v − 1])
9:          end if
10:         parent[v − 1] ← r
11:         return r
12:     end if
13: end function
```

In the Tarjans' algorithm, the processing time is linear because we visit each vertex exactly on time in the first loop, but while visiting we answer to the queries (The second loop and the third one). Notice that the FIND algorithm has a constant complexity in average, because of the compression in the recursion. So the time complexity, to answer **all the queries**, is in $O(n + numberofqueries)$. If the numbers of queries is large ($n = O(numberofqueries)$), then the time to answer each query is constant in average.

Algorithm 2 Tarjan's method to calculate the maximum noise

```
1: function TarjanLCA(u)
2:     parent[v − 1] ← v
3:     for each child v of u in the tree do do
4:         noise[v − 1] ← w = the weight of (u,v)
5:         TarjanLCA(v)
6:         parent[v − 1] ← u
7:     end for
8:     u.visited ← True
9:     for each query (u,v)  do
10:        if v.visited then
11:            lca = the lowest common ancestor of (u,v)  ← find(v)
12:            LCA[lca − 1].append((u, v))
13:        end if
14:    end for
15:    for each (x, y) in LCA[u − 1]  do
16:        find(x)
17:        if y == u then
18:            then answer to the query (x,y) ← noise[x − 1]
19:        else
20:            then answer to the query (x,y) ← max(noise[x − 1], noise[y − 1])
21:        end if
22:    end for
23: end function
```

5
PERFORMANCE IN TESTS

Bellow, the performance timetable of each itineraries' function on each test :
We notice that the time performance of $itineraries - v3$ is the best, while $itineraries - v1$ can't answer to the input 2-9 as the complexity is in $O(n \times numberofqueries)$

Tests	vertices	queries	v1	v2	v3
input 0	10	5	0.00036	$9,9659E-05$	$5,26905E-05$
input 1	20	140	0.00870	0,000657082	0,000352621
input 2	100000	100000	>120 sec	2,994207621	1,517175198
input 3	100000	500000	>120 sec	12,13604617	3,3590312
input 4	100000	500000	>120 sec	12,07247472	3,907314062
input 5	200000	100000	>120 sec	5,710206747	2,238359213
input 6	200000	500000	>120 sec	14,5190928	4,645722151
input 7	200000	500000	>120 sec	14,77447104	4,720973492
input 8	200000	500000	>120 sec	14,35256815	4,639676809
input 9	200000	500000	>120 sec	15,15524077	4,724029541