PI: Shortest Path Trees and Reach in Road Networks - INF421

Pierre-Elie Belouard and Antoine Oustry

January 2017

1 Getting ready : working with network data

The first step of our project was to convert the input data to a relevant graph structure. We builded the class "*Map*" representing the road network.

- We **reindex** the vertices of the graph because the index of the input data are much larger than the number of vertices. At index i of "*correspondayArray*" we have the **real** index (the index in the input data) of the represented vertex. **We will name i the small index of the vertex**. At index i of "*coordinateArray*" we store the coordinates of this vertex.
- The edges are represented by adjacence lists. At box i, we have the list of the (**small**) indexes j of vertices, such that it exists an edge from i to j. The head of a "*List*" instance contains thus the index of the destination but also the duration of the travel.

In fact those three arrays are no simple arrays but arrays of arrays, because it enables the computer not to have to find millions of consecutive memory places. Thus the box for i is (q, r) (which stands for $i = q * M + r$, the euclidian division of i (in $[0, vertexNumbers - 1]$) by $M = 1000$). In the following part of this document, we won't speak of this coding tip anymore, to avoid confusion. We will stay on the fact that a vertex has two indexes, its (large) real index and its small one.

- To know what is the small index corresponding to a real index of a vertex was required to build the adjacence lists. Just with the "*correspondayArray*", this operation would have taken a $O(verticesNumber)$ time. Moreover, we couldn't not use an array storing at box k the new index corresponding to real index k (if it exists). This arrays would have been really to large because the real indices are really big ($\sim 10^9$). That is why we use a **binary search tree** to know in a $O(log(verticesNumber))$ time, which small index corresponds to a given real index.

2 First Task: Properties of Shortest-Path Trees

2.1 Dijkstra's algorithm

For this part, the first step was to implement Dijkstra's algorithm in an efficient way, in order to compute the shortest path tree gooing from a given edge. We used a priority queue to store the next edges to add to the shortest-path tree.

Pseudo-code : Dijkstra's algorithm

Variables : vertex $departure$, graph Map, graph SP, priority queue Q (sorted according to the second variables the label), array $OptimalTime$, array $predecessor$

Algorithm :

$SP = \{departure\}$

For all v such that $(departure, v)$ is an edge of Map :

.....$Q.add(v, duration(departure, v), departure)$.

End for

While (Q is not empty) do :

.....Let be (v, t, u) the root of Q

.....Erase the root of Q

.....If v hasn't been added to SP yet

.....Add v to SP

.....$OptimalTime[v] \leftarrow t$

.....$predecessor[v] \leftarrow u$

.....For all w such that (v, w) is an edge of Map :

..... $Q.add(w, t + duration(v, w), v)$

.....End for;

.....End if

End while

Return SP, $OptimalTime$, $predecessor$

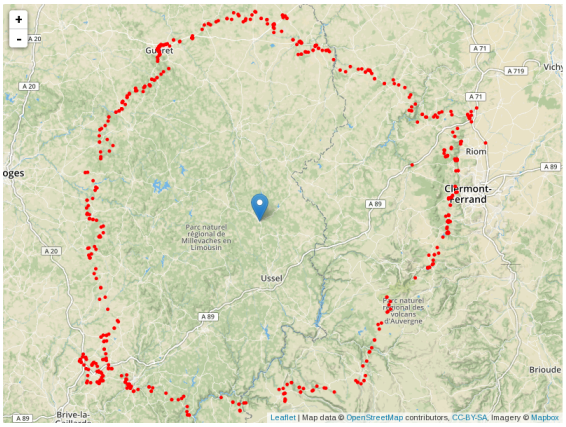
Observations :

- A vertex can be stored many times in Q. But once it has been added to SP, we don't do anything with it if we pop it from Q again. It enables not to have to update some priority weights in the queue.
- Most of time we don't need to compute the shortest-path tree of the all network but under a given travel time. So we put an upper travel time limit : the algorithm stops when it reached all vertices whose travel time from the departure vertex is under this upper bound.
- **Termination and complexity** : each edge is added in the priority queue at most once, each edge is erased from the priority queue at most once. To add one label in the priority queue costs $O(log(|E|))$. To pop one label costs $O(1)$. The global complexity is thus $O(|E|.log(|E|)) = O(|E|.log(|V|))$ (because $|E| \leq |V| * |V|$)

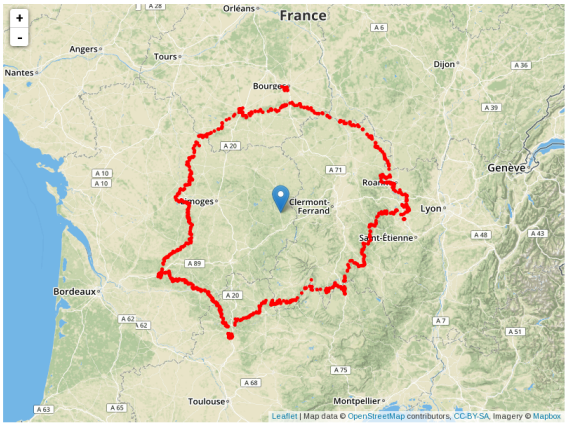
This algorithm enables to answer the questions **1.1** and **1.2** : we compute the SP and $OptimalTime$ with Dijkstra (with a time limit $t + epsilon$) and we keep the vertices whose optimal time is included in $[t - \varepsilon, t + \varepsilon]$ ($t = 1h$ or $2h$, ε the maximum weight of an edge).

Results :

- Departure : La Courtine, Lozère

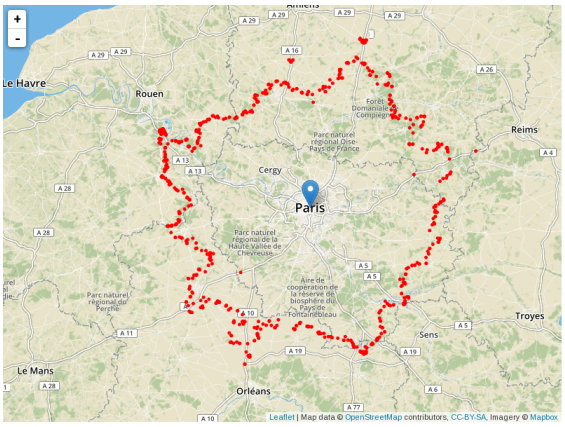


$t = 1h$
538 points

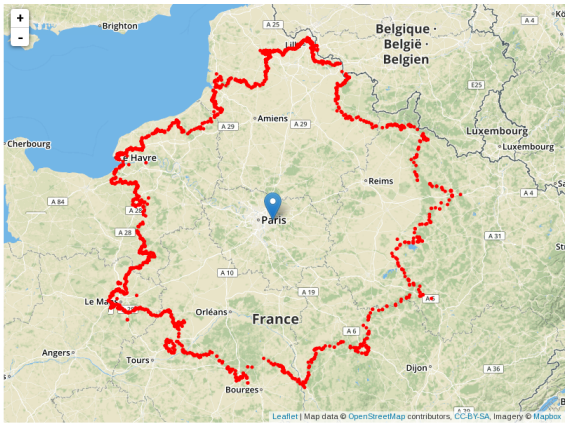


$t = 2h$
1161 points

- Departure : Notre-Dame de Paris, Paris



$t = 1h$
769 points



$t = 2h$
2350 points

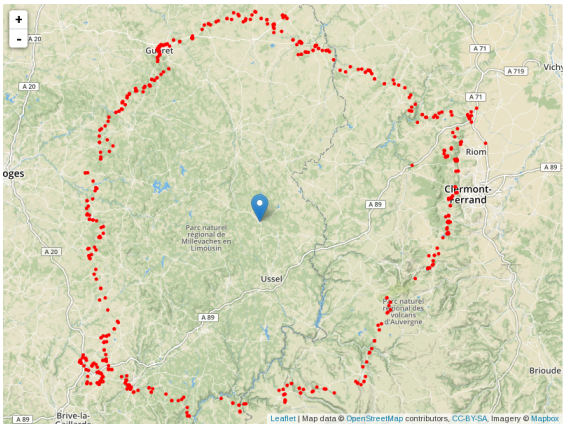
2.2 Strategic roads

1.3. The subject asks to identify the possible positions after 1h of "optimal traveling" to a point 2h away from the starting point. We used also our Dijkstra's algorithm implementation to compute the points two hours away from the starting point. Then we go back to the points 1h away from the starting point, using *predecessor* array (at each vertex, we know where we come from). In order not to go back many times on the same road, we use a "flag" array : when we go back on a vertex, we mark it as already visited. When we arrive at an already visited vertex, we stop the run and we start from an other point which is 2h away.

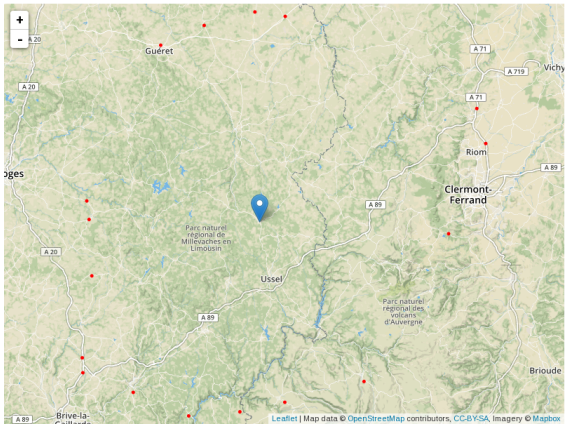
Results

We see than the number of vertices 1h away from v when you are going to a point 2h away is really smaller than the number which are just 1h away.

- Departure : La Courtine, Lozère

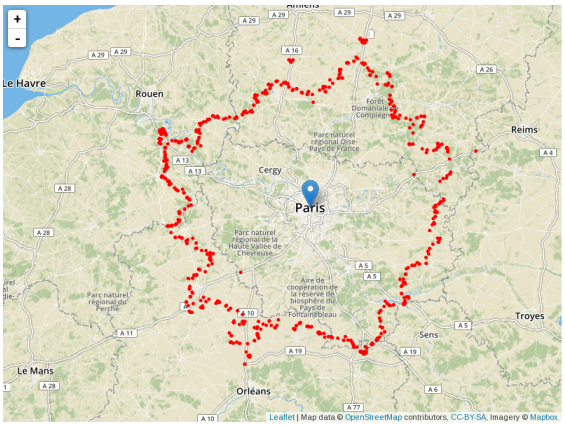


$t = 1h$
538 points

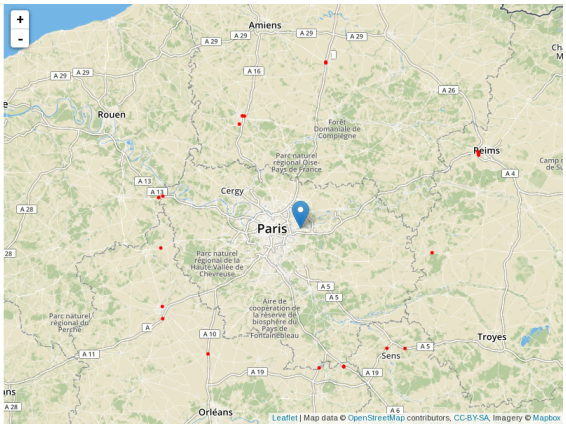


$t_1 = 1h$, $t_2 = 2h$
17 points

- Departure : Notre-Dame de Paris, Paris



$t = 1h$
769 points



$t_1 = 1h$, $t_2 = 2h$
19 points

3 Option B : More theory and algorithms

B.1. Let be v a vertex and let be s and t two vertices such that :

- Some fastest path from s to t passes through vertex v.
- The travel time from s to v following the fastest path is at least $r = reach(v)$.
- The travel time from v to t following the fastest path is at least r.

Thus $optimalDistance(s,t) \geq 2r$ and$D \geq optimalDistance(s,t)$ so $D \geq 2r$.

B.2. Let's assume that it exists a point s in S_{in} and a point t in T_{out}, such that some quickest path from s to$t$ passes through v. We have thus a pair of vertices such that :

- Some fastest path from s to$t$ passes through vertex v.
- The travel time from s to v following the fastest path is at least$1h$.
- The travel time from v to t following the fastest path is at least $1h$.

Thus $reach(v) \geq 1h$.
Let we take the converse assertion : if $reach(v) \leq 1h$, it can't exist a point s in S_{in} and a point t in T_{out}, such that some quickest path from s to$t$ passes through v.

B.3. Let's assume that the reach of v is at least 2 hours. We have thus a pair of vertices such that :

- Some fastest path from s to$t$ passes through vertex v.

- The travel time from s to v following the fastest path is at least $2h$.
- The travel time from v to t following the fastest path is at least $2h$.

Thus there is a point s' between s and v such that the travel time between s and s' is $1h$: so $s' \in S_{in}$. There is also a point t' between v and t such that the travel time betwwen v and t' is $1h$: so $t' \in T_{out}$. The fastest path from s' to t' passes through vertex v otherwise we can find a shortest path between s and t which doesn't pass through v.

B.4. For this algorithm, we use an heuristic upper bound M of the map diameter (ex : 1h for Malta, 4h for Ile-de-France, 20h for France). For $r > 0$ and a vertex v, we use the notation $S_{in}(v, r)$ for the set of points to which the travel time from v is exactly r when traveling to a destination at least $2r$ away and $T_{out}(v, r)$ for the set of points from which the travel time to v is precisely r, when following a quickest path from a starting point located at least $2r$ away from v.

Pseudo-code : algorithm for a 2-approximation of reach(v)
Variables : graph Map, int M, vertex v, int$r$
Algorithm :
$r \leftarrow M$
While $(r > 1s)$:
.....If it exists $s \in S_{in}(v, r/2)$ and $t \in T_{out}(v, r/2)$ such that some fastest path from s to$t$ passes through vertex v :
..... Return r
.....End If
.....$r \leftarrow r/2$
End While
Return $1s$

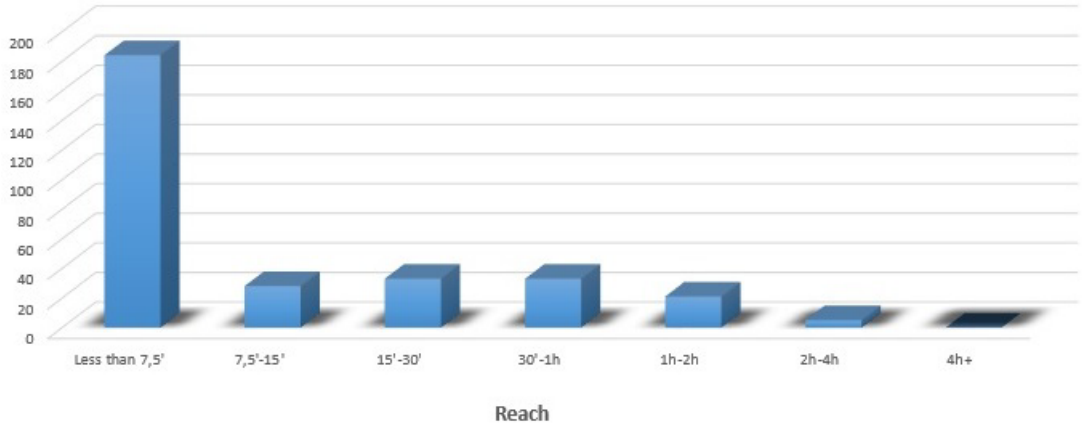
Observations :

- **Termination** : trivial. At worse case, there are $log_2(M)$ rounds in the while loop.
- **Correctness**. Loop invariant : if we enter the loop $r = x$, we know that $reach(v) \leq x$.
  - First round : $r = M \geq D \geq reach(v)$
  - For $r = x \leq M/2$. If we are here, it means that for $r = 2 * x$, there were no $s \in S_{in}(v, r/2) = S_{in}(v, x)$ and $t \in T_{out}(v, r/2) = T_{out}(v, x)$ such that that some fastest path from s to$t$ passes through vertex v. So according to B.3. (and its the converse assertion), $reach(v) \leq x$
  - When we stop at $r = x$, we know that it exists $s \in S_{in}(v, x/2)$ and $t \in T_{out}(v, x/2)$ such that that some fastest path from s to$t$ passes through vertex v. According to the converse assertion of the result of B.2, we know that $reach(v) \geq x/2$. As a conclusion, $x/2 \leq reach(v) \leq x$.

- **Complexity** : this algorithm has a worst case complexity of $log(M)|E|log(|V|)$. We see that it's smart to use the sets $S_{in}(v,r)$ and $T_{out}(v,r)$ rather than the set of the vertices which optimal time to v is r because these two sets **are really smaller** (as we saw in 1.3.). We remind that in order to compute $S_{in}(v,r)$ and $T_{out}(v,r)$, we don't compute the shortest path tree for all vertices of the graph but only for the vertices whose distance to v is less than r.

Exemple of result

Reach of 304 randomly chosen vertices of the map of France. Calculation time : 24h.



Reach	Number of vertices
$r \leq 7min30s$	184
$7min30s \leq r \leq 15min$	28
$15min \leq r \leq 30min$	33
$30min \leq r \leq 1h$	33
$1h \leq r \leq 2h$	21
$2h \leq r \leq 4h$	5
$4h \leq r$	0