Report - Shortest Path Trees and Reach in Road Networks

Grégoire Roussel – Lara Koehler

Structure of the code

To handle the subject, we first wrote classes to structure the data:

- Vertex.java

For each vertex are associated its GPS coordinates (*lng* et *lat*), its identification (*id*), its predecessor (*pred*) and the distance between this edge and the predecessor (*dist*)
This class also contains a constructor, and the functions **toString**, **compareTo** (according to the value of dist), **geoScan** (List<Vertex> *lv*, double *lat*, double *lng*, double *e*) (which lists the vertex situated at distance *e* to the searched point, whose coordinates are *lng* and *lat*, from an original list of vertex *lv*) and **geoPrint**(to save the coordinates for plotting).

- Edge.java

For each edge are associated the id of the start and end vertex (*start* and *end*), its length *length*, and the start and end Vertex (*startVertex* and *endVertex*)
It contains two constructers (with identifications or with Vertex themselves), and the functions **toString**, and **Vertexdistanceexacte** (which takes an argument *t* in [0,1] and creates a new vertex situated on the considered edge, at a distance *t***length* from its start)

- Carte.java

It creates an HashMap of vertex and a List of edges. From the given file, it then fills them.
The variable *PROGRESS* enables to follow the evolution of the calculations of the following functions.
It contains the following functions
. **computeDijkstra**(long *startVertexId*) which gives to each Vertex of the map the shortest distance to startVertex)
. **computeDijkstraIsochrone**(long *startVertexId*, int *distanceLimit*) which creates a list of all the points which shortest distance to *startVertex* is exactly *distanceLimit*)
. **computeDijkstraIsochrone**(long *startVertexId*, int *distanceDestination*, int *distanceLimit*) which creates the list of all the points which shortest distance to *startVertexId* is *distanceDestination*), and then determines where the traveller who reached those points was after an exactly *distanceLimit* time of travel.
. **getIntermediate**(Vertex *dest*, int *distance*) which is used by the previous function to determine the Vertex where the traveller were after a distance time of travel, before arriving at *dest*.
. **shortestPathTo**(long endVertexId) draws the shortest path from a start point of an already calculated Dijkstra to the destination endVertexId.
. **purge** () removes the previous Dijkstra calculation

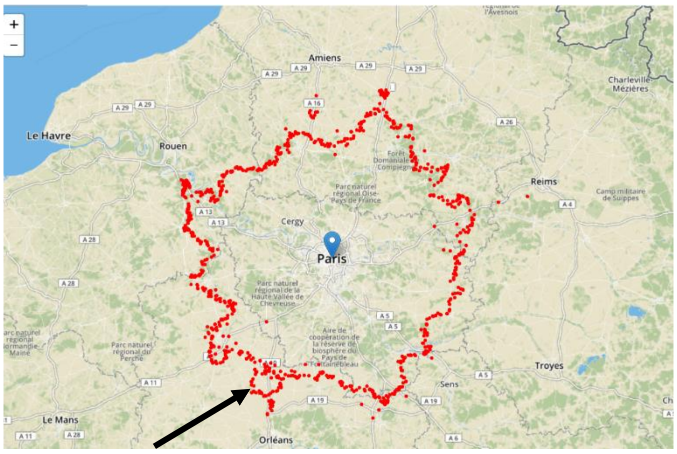
- Networks.java

This class is then used to launch the simulations
. **generatePerimeter**(Carte *map*, long *idStart*, int *length*, String *filename*) draws the map of the points at a distance *length* from *idStart* (from the map *map*), in a new file *filename*.
. **generatePerimeter**(Carte *map*, long *idStart*, int *length*, int *lengthDestination*, String *filename*) does the same but only draws the points at a distance *length* which are crossed to reach a destination at *lengthDestination* from the starting point.
. **multiPerimeter**(Carte *map*, int *length*, int *n*, String *filenameSeed*) does the same as the first **generatePerimeter** function, for n random points on the *map*, and thus creates *n* files.
. **multiPerimeter**(Carte *map*, int length, int *lDest*, int *n*, String *filenameSeed*) does the same as the second **generatePerimeter** function, for n random points on the *map*, and thus creates *n* files.

. **isochrones** (String *mapName*, long *idStart*, List<Integer> *distances*, String *filenameSeed*) does the same as the first **generatePerimeter** function, for a list distances of *length*.
. and the class **main** to run all this fonctions

Properties of shortest path trees

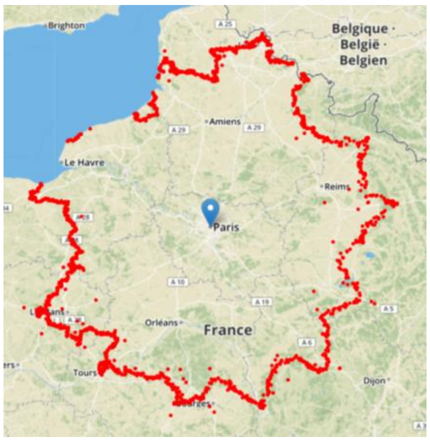
(1.1) After having used geoScan, from the class Vertex, to find a Vertex situated in the center of Paris, we ran our founction **generatePermiter** with this starting point and the distance one hour (3 600 000 ms)



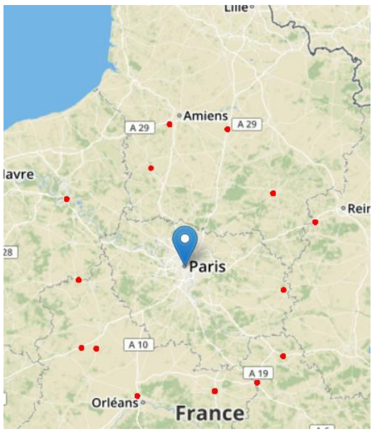
We can see that there is kind of a circle around Paris, which seems logical, and there are some peaks above the main axes, that enable to travel faster.

We can also see some circles, which can correspond to interchange of big axes: all the points around this interchange are reached in the same time approximately. One example is indicated with the black arrow.

(1.2) We did the same with a distance of two hours. The allure of a star is even more accentuated for a two hours' travel.



(1.3) We used the second function **generatePerimeter**, with distanceLimit=1h and distanceDestination=2h)



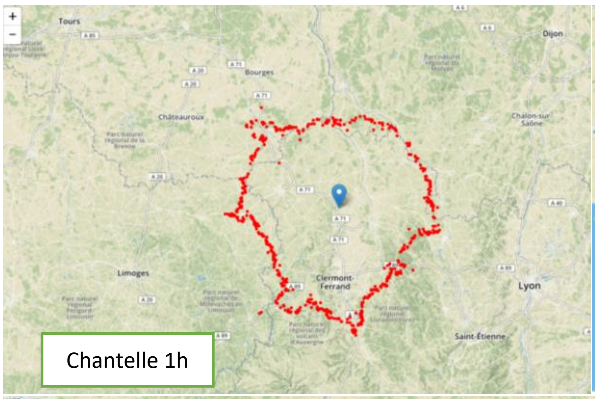
We can see that only a few points remain (compared to the map in (1.1)), which seems logical: to travel far, you only use big axes in the middle of your travel. The smaller axes are only used at the beginning and the end of the travel, to reach a special (and maybe isolated) point.

That analysis may be the starting point to design better algorithm: instead of scanning each road, we can scan all the roads (including the small ones) leading to the big axes, then analyse only those axes to get close to the destination and then scan the local roads between the point reached and the destination.

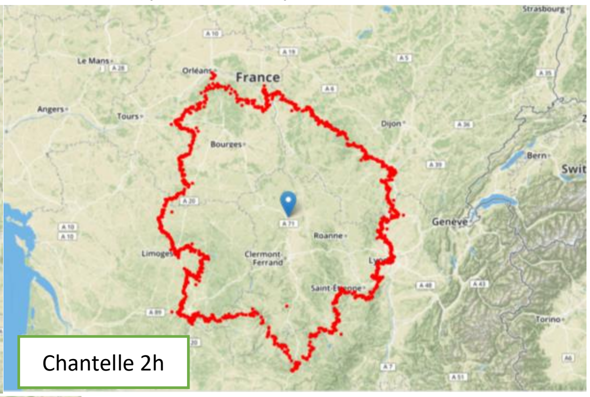
(1.4) The algorithm we used, the Dijkstra algorithm, with a min-heap priority queue. Therefore, according to the theorem on section 5.4.2 of the poly, it is a O(Elog(V)) complexity.
When t1 and t2 are small, the complexity is smaller than that, because we can restrict to a smaller range of point: we stop when the distance t1 or t2 has been reached. For example, with the map of France, only 10 percent of the vertex are considered by the algorithm.

More experiments

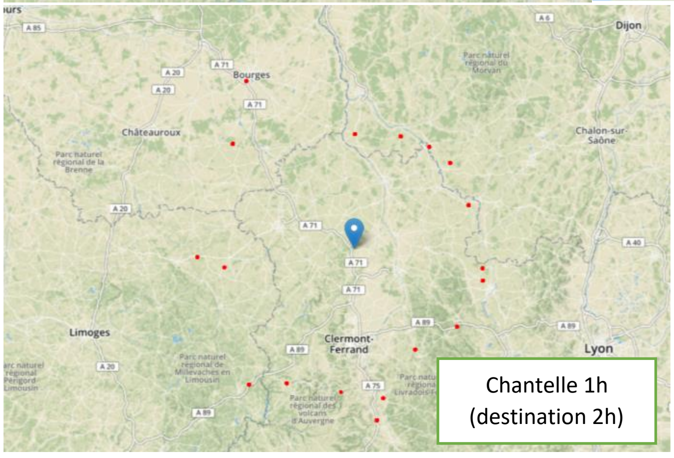
(A.1) We have chosen Chantelle (in the Creuse department) and the 72th building in the Campus of the School) as starting points for these three simulations.
We can see that for remote points, the isochrone is less symmetrical than the one for starting in Paris, and this isochrone is also smaller, because the network is less developed on this place.



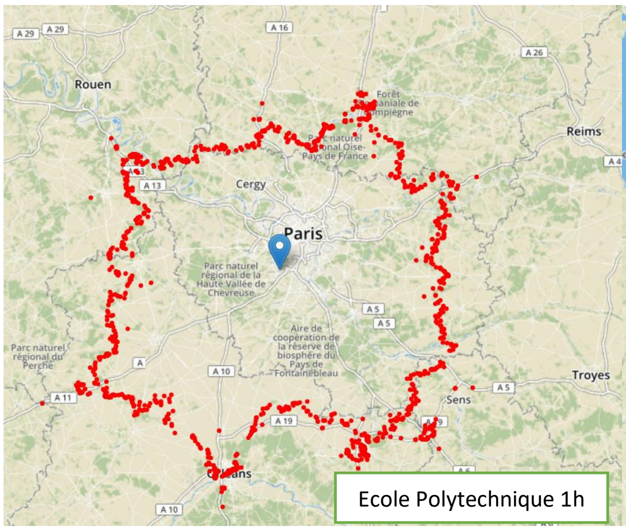
Chantelle 1h



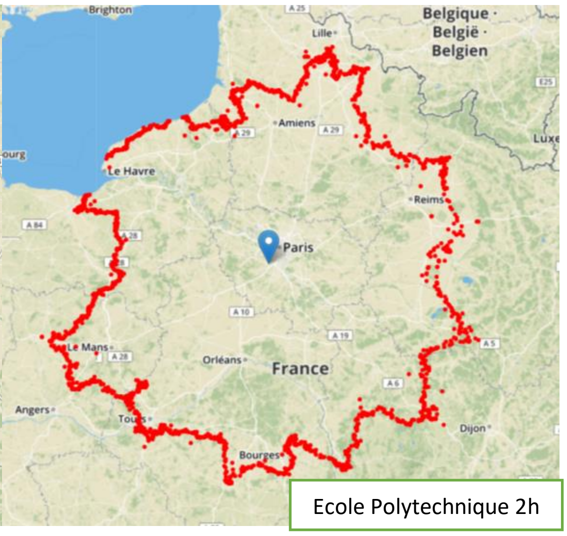
Chantelle 2h



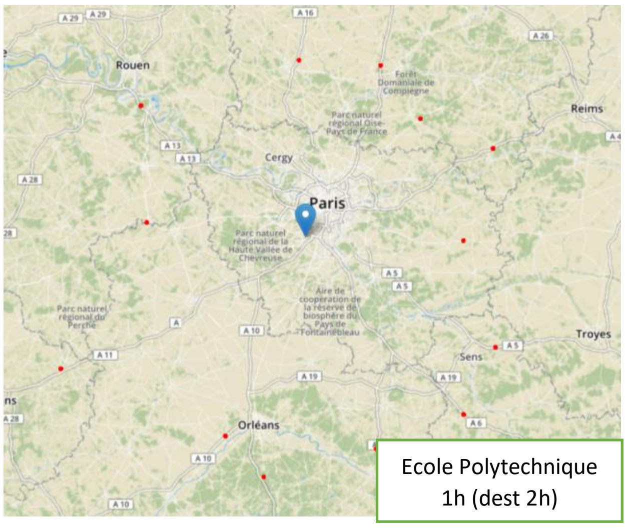
Chantelle 1h
(destination 2h)



Ecole Polytechnique 1h



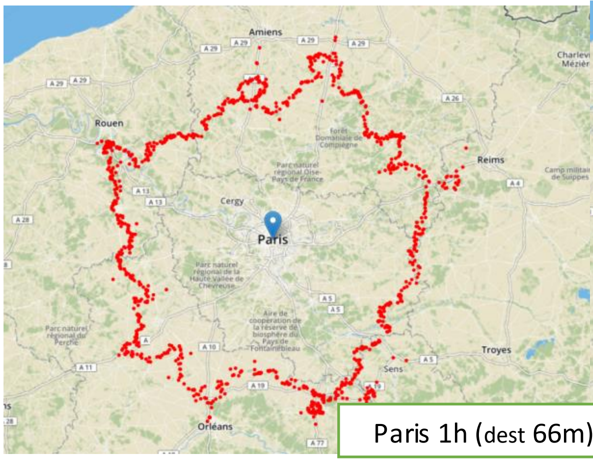
Ecole Polytechnique 2h



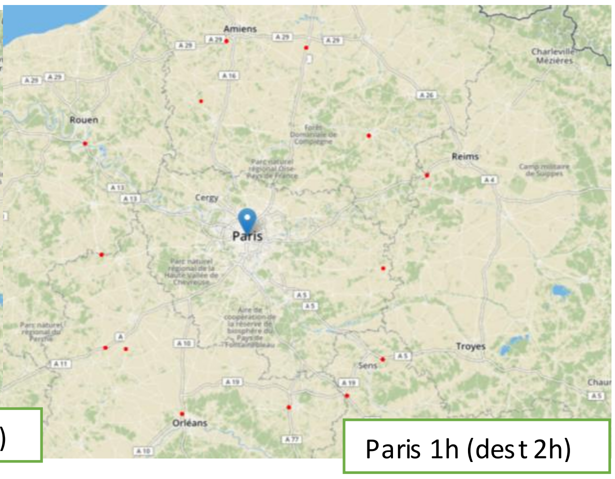
Ecole Polytechnique 1h (dest 2h)

(A.2) We can observe concentric circles

(A.3) We see that less and less intermediary points exist at t=1h.



Paris 1h (dest 66m)

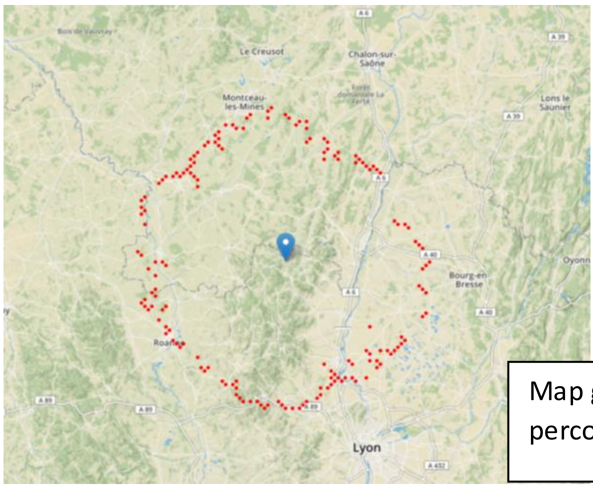


Paris 1h (dest 2h)

(A.4) We tried to create of France with the technic of bound percolation, in order to compare our results with those obtained the previous map.

The pattern is globally similar, but we can notice the absence of some details, like peaks above the big axes or circles around interchanges of big axes. This comes from the fact that all our edges have the same length (there are no notion of roads faster than others). Moreover, some regions are completely inaccessible when the percolation factor is too low.

It would be interesting to further analyse those conclusions with maps created with variable percolation factors depending on the geographic area (higher in towns and very low in mountains, in order to describe better the reality.



Map generated through bond percolation (perc1-2)