Distance algorithms discription

Base algorithm

Preparation

- First, you need a way to define **zones** : a delimited area with a given centroid to separate postcodes.
- Second, fill a database with all the distance between each pair of **zone**, by route and as the crow flies.

Get distance between postcodes

Given two postcodes (postcode, country iso code) :

1. Get the distance as the crow flies between the two postcodes (named C)
2. Find to which group belong each postcode
3. Extract from the database the two kinds of distance between the two *zones* and calculate the ratio $route/crow$ (named R).
4. Finally calculate : $C * R$

2-digits version

The orinal proposal was to use the 2 first digits of the postcodes to define the *zones*. By *"2 first digits"*, I think was meant : *Administrative region* (such as *Bundesland* in Germany, *Département* in France or *State* in the U.S.A.) ; and indeed in France and Germany, these *zones* are indicated by the two first digits of the postcodes.

This, however, cannot be generalized to every country in Europe, so I didn't want to just use the first two digits directly at first. I looked for a way to define admninistrative regions and their contained postcodes but couldn't find a proper solution for that.

Another reason why I didn't think i would be good reason was the shapes of the **admninistrative regions** that weren't best suited for the distance of the centroids.
For instance, we can look at the map of the German Bundesländer below, in the area between Bayern, Baden-Württemberg and Hessen : If the 2-digits algorithm is used, the Bavarian part would be placed at the Bavaria-centroid between Nürnberg and München and the cities around it in Hessen would be considered to be near Marburg which creates a distance of approximatelly 350km.

Dortmund
Kassel
Leipzig
Erfurt
Weimar
Marburg
Mainz
Frankfurt am Main
Würzburg
Nürnberg
Mannheim
Heidelberg
Karlsruhe
Stuttgart
Tübingen
Ulm
Passau
Augsburg
München
Salzburg
Freiburg
Konstanz
Hohenschwangau

Finally, to implement this version of the algorithm, I used grouped every city with their forst two digits and country code, **even though this doesn't necessarily make sense for all countries, and can cause inaccuracies**.

To get the centroid of these *zones*, I just calculated the average of the postcodes coordinates. This is maybe not a good way, but I don't see any other solution.

Using this method, we get **1840** *zones*, which makes **1 691 880** combinations.

Grid version

To solve the problems stated above (and make the development easier), I thought of using a **grid** instead of the 2-digit region. The grid is made by *meridians* and *circles of latitude*, at a certain step (every degree, tenth of degree, ...).

This solution has these advantages :

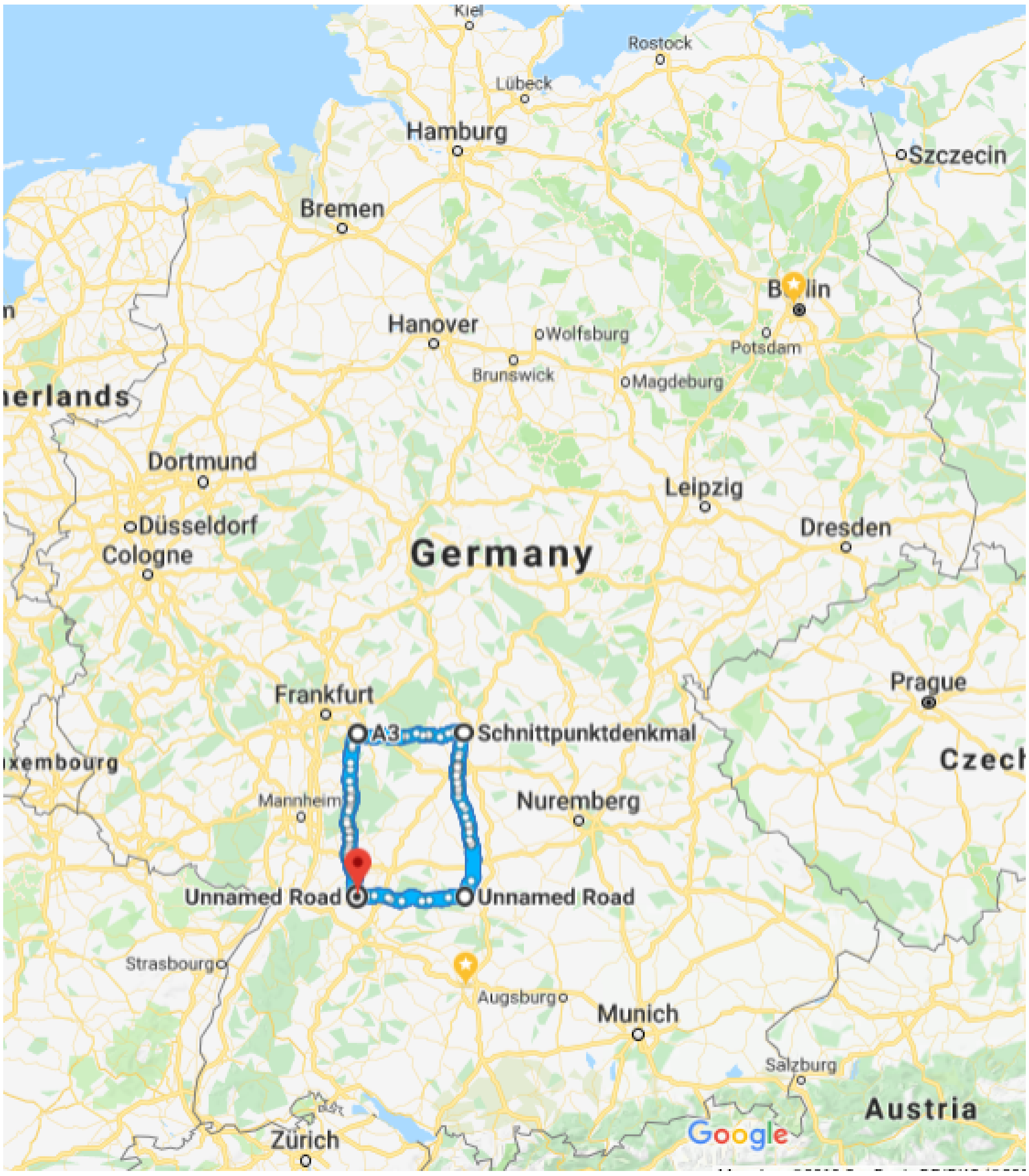
- It is much easier to define in which region is located each postcode (directly look at the coordinates)
- The *zones*' shapes are standardized and *approximatly* the same size.
- The *zones* have a *squary* shape which could provide better accuracy.

Coordinates in Europe (continental + Ireland and UK + Mediterranean Isles) roughly goes from 71°N,11°W (Norway and Ireland) to 35°N,41°E (Spain and Ukraine). So we have a difference of latitude and longitude : 36 and 52.

If we cut every degree, we have a total of **1872** *rectangles*. So we would have as a maximum number of distances between each couple of *rectangle* of $\binom{1872}{2} = 1,75 * 10^6$. By filtering out the *rectangles* that don't contain any postcodes, we have only **1056** rectangles, which gives **557040** combinations.

If we cut every tenth of degree, we have a total of **187200** *rectangles*, **29 010** only containing postcodes, which gives **420 775 545** combinations.

To get an idea of the size of such *rectangle*, here is an example in Germany between 49°N,9°E and 50°N,10°E, it is 85km wide and 130km long:



Graphhopper problem

Graphhopper has sometimes trouble tracing roads between two points when one of the coordinates are located not enough close from the road or something, **the request returns an error**.

This creates a problem when requesting the distance by road between two *zones*' centroids : an error occurs and the corresponding line is not entered in the database. For this reason, calculating the distance using any of the algorithms from the database won't work.

Solution

Probably one of the best solutions would be to *circle around* the centroids until we find coordinates that work, that would increase a lot the time spent generating the distance database and it would be difficult to implement.

For these reasons, when these errors happened, I first chose to default to the distance *as the crow flies*. After some tests, I found that, in average, the distance *as the crow flies* is **76%** of the distance by route. So I set the default to **dist_crow / 0.76**.

> **Note :** This is also applied when the two requested postcodes are **in the same *zone***.