Automatic Inference of Roadmaps from Raw Mobility Traces

Mickaël Duruisseau, Romain Rouvoy

Automatic Inference of Roadmaps from Raw Mobility Traces

Mickaël Duruisseau, Romain Rouvoy

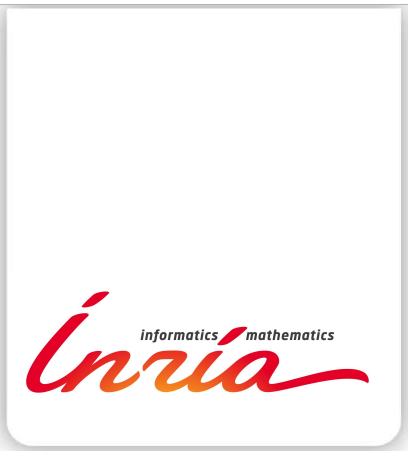
RESEARCH REPORT

N° 8585

August 2014

Team Spirals

ISSN 0249-6399 ISRN INRIA/RR--8585--FR+ENG



Automatic Inference of Roadmaps from Raw Mobility Traces

Mickaël Duruisseau, Romain Rouvoy

Team Spirals

Research Report n° 8585 — August 2014 — 19 pages

Abstract: The popularization of smartphones in daily life offers numerous opportunities in terms of urban sensing. More and more users are ready to share certain information as part of scientific research, including their GPS location. From these mobility traces, we developed a roadmap inference algorithm using raw mobile data supplied by users of smartphones. This algorithm can generate a map composed of oriented routes, which are annotated by a certain amount of metadata.

Key-words: clustering, roadmap inference, smartphone, urban sensing

RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Inférence automatique de routes à partir de données mobiles brutes

Résumé : Le succès des *smartphones* dans la vie quotidienne ajoute de nombreuses opportunités en matière de collecte de données urbaines. De plus en plus d'utilisateurs sont prêts à partager certaines données dans le cadre de recherches scientifiques, notamment leurs positions GPS. À partir de telles traces de mobilités, nous avons developpé un algorithme d'inférence de cartes routières à partir de données mobiles brutes, fournies par des utilisateurs de *smartphones*. Cet algorithme permet de générer une carte routière composée de routes orientées, ainsi qu'un certain nombre d'informations sur celles-ci.

Mots-clés : clustering, inférence, carte routière, smartphones

Contents

1 Introduction

Nowadays, there are many maps produced manually or via the use of special cars, such as Google Maps[1] or OpenStreetMap[2]. Although these maps are pretty accurate, they are not always up-to-date, and requires continuous interventions to update them.

Inferred maps, however, may be used to detect changes to the road network or errors in existing maps, as well as being updated in real-time. Map inference may also be used to produce custom maps for certain classes of travelers, such as pedestrians or cyclists, by feeding it with data from different sources.

However, inference requires a lot of data to be effective, which was a limitation until recently. The GPS[3] traces needed for producing such maps are becoming increasingly available due to the popularity of smartphones and GPS navigators, which capture vast volumes of user location traces on a daily basis.

The main goal and motivation of this work is to be able to infer roadmaps on a large scale, which is not possible when using pictures [4, 5] Also, keeping the road bearing or accuracy is not easy with images, nor is the ability to extract several lanes on the same roads.

Additionnally, our goal is to update this map in real-time, just by adding data, collected via crowd-sensing systems, for example. This can be particularly useful when a road is closed due to roadworks, hazard or anything that could block it. Moreover, depending of the data, we can infer pedestrian ways, bike trails or anything like that.

The contributions described in this paper are:

1. **A three dimensional clustering algorithm** for GIS[4] datasets, based on coordinates and orientation,
2. **A weighted distance calculation** using distance, bearing and accuracy,
3. **A graph database storage** to represent roads by relationships between nodes,
4. **A road-maps inference algorithm** and several filtering algorithms based on this database.

2 State of the Art

In this section, we introduce various roadmap inference approaches, which have been already reported in the literature.

2.1 K-Means Clustering

This family of algorithms, originally described in Edelkamp and Schrödl [6], consists in distributing a serie of cluster seeds at different locations based upon input trace data. These seeds are placed following constraints on bearing and distance and are used as initial guesses for a standard K-Means clustering algorithm [6]. Once the clusters have been built, they are linked to form road segments based on the patterns that the raw traces used to follow initially.

[1] `http://maps.google.com`
[2] `http://openstreetmap.org`
[3] Global Positioning System: `http://www.gps.gov`
[4] Geographic Information Systems: `http://en.wikipedia.org/wiki/Geographic_information_system`

Edelkamp and Schrödl [6] The paper by Edelkamp and Schrödl, is the original k-means based method, but it also refines the road network model by fine-tuning the location of intersections, representing the road center-line by a fitted spline rather than a series of straight lines.

Schroedl et al. [17] This paper describes a process to refine the intersection geometry and model individual lanes. This process is accomplished by identifying intersections and their bounding boxes, and group traces by entry and exit points. Then, a spline-fitting technique produce the final turn-lane intersection.

Agamennoni et al. [1] This paper uses an approach similar to Schroedl et al. [17], but focuses to extract principal road paths.

2.2 Trace Merging

Trace merging algorithms iterate over each GPS trace recorded by adding them to a map. For each segment from the trace, a test is performed to determine if it already exists on the map or if it is "similar enough". In this case, the segment is merged into the existing one, otherwise it is added as a new segment. When a segment is added to an existing one its weight is increased. Segments with a weight below a certain threshold are removed.

Niehoefer et al. [13] The method described by Niehoefer et al. [13] evolves the merging approach by adjusting the position of road segment when merging a new trace. This technique allows the location of road segment to be refined as more traces are added. This paper also describes a way to automatically classify road types, such as streets or highways.

2.3 Kernel Density Estimation

Kernel Density Estimation (KDE) algorithms begin by discretizing the geometric space covered by the set of GPS points into a grid of pixels, recording the approximate density of samples for each cell. A threshold is applied to the density map in order to produce a binary representation of the road network, and the center-line is extracted using a variety of image-processing methods.

Biagioni et al. [4] Kernel Density Estimation is heavily dependent on the threshold parameter used to extract the boolean map. This introduces the tradeoff between map accuracy and road coverage. Since it is such a sensible value, it is proposed by Biagioni and Eriksson [4] that no single threshold can produced the desired map thereby a methodology for constructing a map with various thresholds is proposed.

3 Contributions

3.1 Input Datasets

In this work, we used two different datasets. The first one is provided by the state of the art, and we use it to compare the accuracy of our algorithms. The second is a dataset we created with the help of a colleague who works on smartphone energy consumption.

Chicago Bus Shuttle This dataset is used by Biagioni et al. [4, 5] in their comparison and evaluation of the state of the art. Almost one 120,000 locations are reported, but with only their locations and timestamps. Furthermore, each location (except the first and the last ones) is linked to the next and previous locations in the mobility trace.

Villeneuve d'Ascq Daily Trips As described earlier, this dataset was created by Inria and contains more than 22,000 locations. Compared to the Chicago one, this dataset provides more metadata for every location, especially the accuracy and the bearing. This information is very useful to filter spurious locations before applying our algorithm.

3.2 Clustering Algorithm

The first contribution reported in this paper is a clustering algorithm used to group raw locations into groups of correlated locations. We first describe the data structures used to store and manipulate the location traces we extracted from the datasets and then report on the algorithms defined to process these structures.

3.2.1 Data Structures

We use several structures to represent data. In our program, we employ Scala [5] as main language, and choose to implement those structures as classes:

GPSNode - A raw location, it contains all the information retrieved by a GPS fix, plus the trip and cluster to which the node belongs, and its neighbours nodes.

GPSCluster - A group of correlated nodes, its own location (latitude, longitude and bearing) is updated whenever a `GPSNode` is added and computed as the geometric center of the nodes it encloses.

Trip - A list of consecutive nodes, linked to each other.

Figure 1 is the UML[6] class diagram of our project.

[5] `http://www.scala-lang.org`
[6] Unified Modeling Language: `http://www.uml.org/`

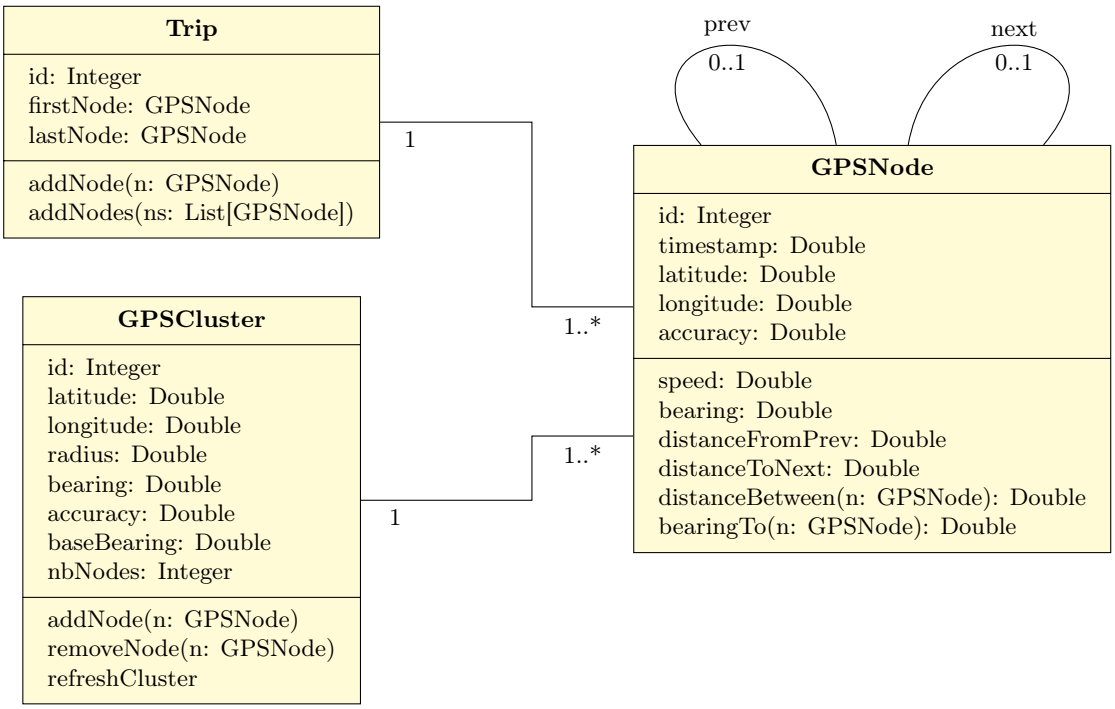


Figure 1: Class diagram

3.2.2 Procedure Overview

This procedure describes the program flow and uses the structures introduced in Section 3.2.1. The first step of our procedure is to create trips (`Trip`) filled with nodes (`GPSNode`). We parse the dataset files, create a `GPSNode` per line with all the data provided, and make links between them, in order that nodes are linked by their properties `next` and `prev`. For example, $N_1.next = N_2,\ N_2.prev = N_1$, as depicted in Figure 2.

Those links are used to loop over a trip or a road, and to calculate the velocity of a node, together with its bearing, if it was not already set (depending of the dataset).

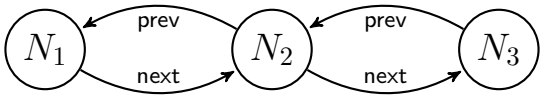


Figure 2: Links between two nodes

Then, we use Algorithm 1 to create the location clusters from the raw locations, this algorithm iterates over the list of trips and the locations each trip contains.

Additionally, we feed a database with all the locations, clusters and roads created by this algorithm.

Then, we apply several filters on the model to clean the clusters and inferred roads, as lated explained in Section 3.5.

Algorithm 1 Clustering algorithm

```
for all node n in nodesList do
    if n.accuracy ≤ tripAcc then
        nearC ← getNearbyClusters(n)
        cluster ← null
        minD ← Double.MaxValue

        for all cluster c in nearC do
            d ← distance(c, n)
            if d < 0.5 && d < minD then
                cluster ← c
                minD ← d
            end if
        end for

        if cluster == null then
            cluster ← Cluster(n.lat, n.lon, n.bearing)
            cluster.addNode(n)
            database.insertCluster(cluster)
        else
            cluster.addNode(n)
            database.updateCluster(cluster)
        end if

        database.addNode(n)
        makeRoad(n.prev.cluster, n.cluster)

    else ▷ If the node is not accurate enough, we delete it, and relink its neighbours together
        if n.next! = null then
            n.next.prev ← n.prev

            if n.prev! = null then
                n.prev.next ← n.next
            end if
        end if
    end if
end for
```

3.3 Distance Computation

Our clustering algorithm uses a specific distance computation (function *distance* in Aglorithm 1), to create clusters depending on three parameters:

- The *spatial distance*,
- The *bearing difference*,
- The *location accuracy*.

The distance formula we defined is customizable with a weight p. This weight allows us to change easily the importance of the distance or the bearing, depending of what we think will make better clusters.

$$distance(N_1, N_2) = p \times f_d(d_{N_1,N_2}) + (1-p) \times f_b(bd_{N_1,N_2}) \tag{1}$$

3D distance formula

In this formula, f_d and f_b are Gaussian functions parameterized by a threshold, they return 0.5 for a given value.

The result of that computation is a value normalized between 0 and 1. 1 meaning that the two input locations are exactly similar, 0 meaning that they are too far or with a high bearing difference.

In this paper, we consider that two nodes are close only if the distance is more than 0.5.

3.3.1 Spatial Distance

We use the Haversine Formula [19] to compute the distance *sd* between two points. This formula gives great-circle distances between two points on a sphere from their longitudes and latitudes.

$$\begin{aligned}
&R = 6371.0//km \\
&dLat = rad(lat_{x2} - lat_{x1}) \\
&dLon = rad(lon_{x2} - lon_{x1}) \\
&lat1 = rad(lat_{x1}) \\
&lat2 = rad(lat_{x2}) \\
&\\
&a = (\sin(\frac{dLat}{2}))^2 + (\sin(\frac{dLon}{2}))^2 + \cos(lat1) \times \cos(lat2) \\
&c = 2 * atan2(\sqrt{a}, \sqrt{1-a}) \\
&d = R \times c \\
&sd = d * 1000//m
\end{aligned} \tag{2}$$

The Harvesine formula

Even if this formula uses an approximate value for the Earth radius, the results given are accurate enough for short distances. Once we have computed that distance, we apply an accuracy threshold, explained in Section 3.3.3. Then, we use a Gaussian function on the result, to normalize the distance into a value between 0 and 1, that represents the similarity between two nodes.

$$
\begin{aligned}
&s = 5.0 \\
&f_d(d) = e^{-(\lambda_d \times d)^2},\ \lambda_d = \sqrt{\frac{\log 2}{s^2}}
\end{aligned}
\tag{3}
$$

Gaussian function for distance normalization

This formula is set by the value s, meaning that it will return 0.5 if $d == s$, more for a lower s, and less for a higher s.

In our algorithm, we set s to 5.0 because we wanted to be able to separate multiple lanes of the same road, for example on a highway.

It also means that our clusters will be constituted of the closest nodes with a maximum distance of 5 meters.

3.3.2 Bearing Difference

In addition to the spatial distance, we also use the bearing in our distance computation. It allows us to create oriented clusters and thus oriented roads. We apply the same procedure as for the spatial distance—*i.e.*, we calculate the difference between the two bearings—and then apply a Gaussian function.

First, we have to retrieve the bearing of a node. There are two ways: either the bearing is returned by the satellite when it fixes the location, or we calculate it with the following formula.

$$
\begin{aligned}
&dLat = rad(lat_{x2} - lat_{x1}) \\
&dLon = rad(lon_{x2} - lon_{x1}) \\
&lat1 = rad(lat_{x1}) \\
&lat2 = rad(lat_{x2}) \\
\\
&y = \sin(dLon) \times \cos(lat2) \\
&x = \cos(lat1) \times \sin(lat2) - \sin(lat1) \times \cos(lat2) \times \cos(dLon) \\
&b = deg(atan2(y, x)) + 360 \pmod{360}
\end{aligned}
\tag{4}
$$

Bearing between two locations

Then, we obtain an angle in degrees between the two bearings, we want to get the difference between two angles, modulo 360. To do that, we use the following formula, where a and b are two angles in degrees.

To finish, we apply a Gaussian function, formula 6, as we did for the spatial distance, that returns the similarity between two angles. This function will return 0.5 for a given value s, more for a lower s, and less for a higher s. In our algorithm, we set s to 15.0, it allows us to detect any kind of change in the road direction.

The result of this function is then added to the spatial distance with a weight (the complement of p, $(1-p)$) to get the final result of our distance computation.

$$
\begin{aligned}
&bd_{x1,x2} = 360 + b_{x1} - b_{x2} \pmod{360} \\
&if(bd_{x1,x2}) > \frac{360}{2}) \\
&\quad bd_{x1,x2} = 360 - bd_{x1,x2}
\end{aligned}
\tag{5}
$$

Difference between two angles, modulo 360

$$
\begin{aligned}
&f_b(b) = e^{-(\lambda_b \times b)^2},\ \lambda_b = \sqrt{\frac{\log 2}{s^2}} \\
&s = 15.0
\end{aligned}
\tag{6}
$$

Gaussian function for bearing

3.3.3 Accuracy Threshold

Depending on several physical conditions, a GPS fix can be more or less accurate. For example, in large cities, urban canyons [18] can reduce the vibility of available satellites, and so the accuracy. Every fix is therefore supposed to be annotated with an accuracy.

We chose to be optimistic in our approach: we deduce the average accuracy of the two nodes to their distance, as shown in the formula 7.

$$
d_{x1,x2} = max(0,\ sd_{x1,x2} - \frac{acc_{x1} + acc_{x2}}{2}) \tag{7}
$$

Distance between two nodes, with accuracy

This method allows to merge nodes that are far from each other but also inaccurate, as they could be close if the fix was more accurate. This also helps to create cleaner roads, since we merge nodes and move the clusters according to them.

3.4 Database Storage

The database storage was one of the most sensible point to take care of, because we wanted our solution to be scalable, fast, and efficient. Thanks to our program architecture, we can easily change the database type we want to use, this allowed us to try our algorithms on several technologies.

At first, we tried to use a Scala R-Tree implementation, named Archery [3], and to store our dataset in memory. It worked fine, but there was no data persistence, and the associated query engine was not powerful enough.

Then, we used some relational database, like PostgreSQL [15], with a spatial plug-in to easily store and request on geo data. This kind of database works well, but relationships between nodes are not easy to store neither to query.

Finally, we switched to graph databases, that fits all our requirements, as it can store nodes and edges, and have a fast and powerful query engine. We will see in section 3.4.1 more about graph database.

3.4.1 Graph Database

The main advantages of a graph database—compared to a relational one—is the ability to link nodes to each others, and be able to execute powerful query using those edges. For our research, we study several graph databases, we wanted one that was fast, scalable and with spatial support.

The first one we tried was Neo4J [12], which comes with a very powerful query engine, and a data visualization website. Our algorithm was pretty fast with it, but despite those advantages, Neo4J does not handle very well concurrency, and its REST API is not fast enough.

Then, we used Mahout and Hadoop, but even if, at first, our algorithm was using MapReduce [8, 11, 9, 14], they did not fit our requirements. GraphX and Spark, however, was a good alternative. Fast and with a great scalability and concurrency, despite some bugs, the main problem was the limited API, which did not allow us to store multiple type of data, neither make fast requests or clusters.

Finally, we chose to use Titan DB [20], that we will present in section 3.4.3.

3.4.2 Graph Structure

Beside the database, our graph structure did not change. We store nodes and clusters with all their properties, and relations between them.

There are three types of relations, as shown in Figure 3.

`road` - Represents a road segment, between two clusters

`inside` - Means that a node is inside that cluster

`next` - Links the nodes to their neighbours

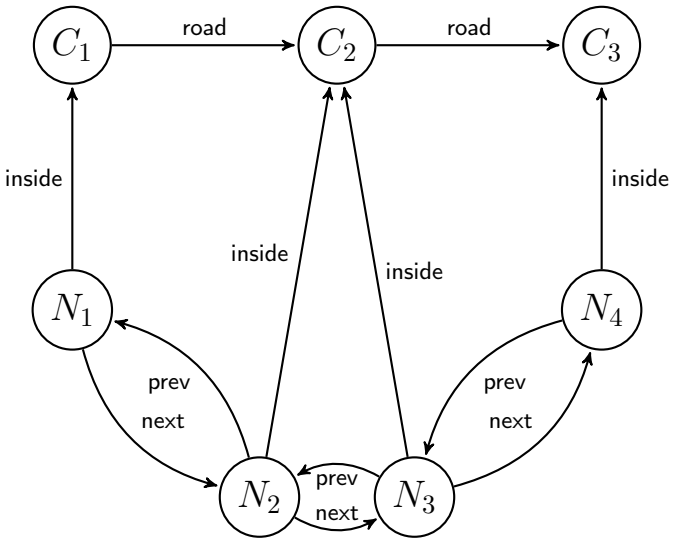


Figure 3: Graph DB structure

3.4.3 Titan Distributed Graph Database

Titan [20] is a scalable graph database that supports transactions and so concurrency. It also supports several storage and index back-ends, as well as geographic data. So, everything our algorithm needs is provided by Titan.

It comes with a query language, named Gremlin [10], a powerful Java API, and a graph server called Rexster [16], which is useful to visualize and query data.

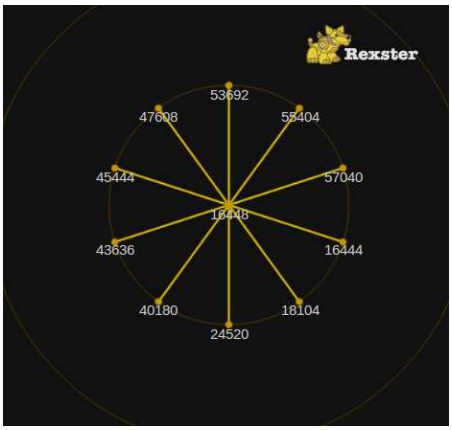


Figure 4: Rexster visualization of a cluster and its nodes

For our work, we used version 0.4.4 of Titan, with Rexster, Cassandra [2] as storage back-end and ElasticSearch [7] as index back-end.

With Titan, we had to describe the graph first. As presented in Section 3.2.1 and in Figure 3, we store `GPSNode` and `GPSCluster`, with all their properties, plus a property `type`, which indicates whether the node is a *node* (`GPSNode`) or a *cluster*. With that, we defined the three possible relations: *inside*, *next*, and *road*.

There was no need of a *prev* relation, as in the structure, because Titan allows us to get edges that comes in and out of a node. So almost every nodes have two *next* relations, one that goes out, and one that goes in, which can be used as *prev* relation.

We also created several indexes on our data. The first one is on the *type* property, and allows us to quickly get every nodes of a given type. The second is for the *geo* property, to get nodes close to a given location. And the third is a combination of the two first, *type* and *geo*, to get nearby clusters as used in Algorithm 1.

3.5 Filtering Algorithms

Once our algorithm 1 is completed, we obtain a large number of clusters and roads. Some of these clusters are incorrect or should be merged with a neighbour, and this applies also for roads. In this section, we will present four ways to improve data results, without loosing information.

3.5.1 Inconsistent Nodes

Even with our accuracy threshold (cf. Section 3.3.3), sometimes a cluster is created with one or two nodes, especially if the dataset does not provide a decent accuracy—it is the case for the Chicago dataset, for example. In this case, we apply a threshold filter on clusters. For every clusters, we calculate the average number of nodes in nearby clusters, and we delete the cluster if it has less nodes than half of this average (we chose to delete those with less than $\frac{avg}{2}$, but this is easily customizable).

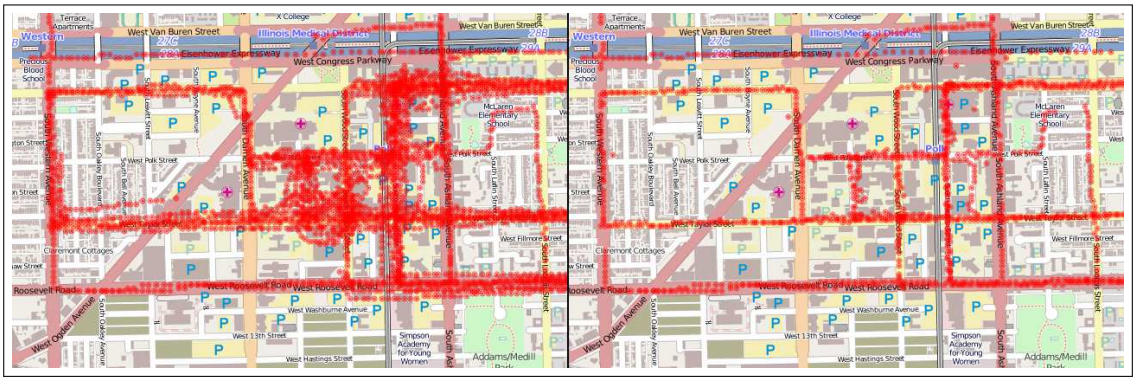


Figure 5: Clusters filter

Figure 5 illustrates the quality of this filter. On the top image, we can see all the clusters, while only the clusters that are large enough are shown on the other. One can easily notice that a large number of spurious clusters have been deleted.

This method is useful to delete clusters created from inaccurate nodes, even if accuracy is not set, and so remove the noise from GPS, or potential errors due to urban canyon or something else that reduces the number of available satellites.

However, with data whose accuracy is defined, it is less useful, because inconsistent clusters are less likely to exist.

3.5.2 Duplicated and Loop Roads

Our next clean-up filter focus on roads, more precisely, roads that already exist, or that come and go in the same cluster. Duplicate roads could be hard to find if we had to iterate on every roads twice, but instead of doing that, we chose to iterate over clusters. For each cluster, we check if roads are unique or not—*i.e.*, if an outgoing cluster has already been found—and we delete the associated road.

One good way would have been to set a road as unique when we defined the graph, but it was not possible with a *many to many* relation.

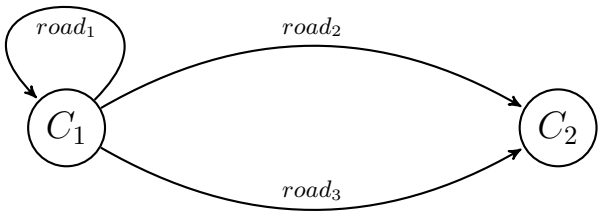


Figure 6: Loop and duplicated roads

The second—road loops—is easy to find and only requires one loop. For every road, we just have to check if the start node is the same as the final node. In this case, we remove the road, since there cannot be roads which come and go in the same cluster. Figure 6 shows these cases. Here, $road_1$ and $road_3$ are automatically removed by our second filter.

3.5.3 Triangle & Trapeze Paths

As shown in Figure 7, triangle paths appear when two nodes have a *next* in different clusters. This can happen if the time between two nodes is longer than usual (*e.g.*, because of traffic).

Trapeze paths appear in the same conditions, but with an intermediate in addition, C_3 in the graph.

Generally, roads created because of that are "shortcuts" between two nodes, and are not correct. It can happen with just one (triangle) or two (trapeze) intermediates, but also with more. We chose to detect and delete only triangle and trapeze, as this algorithm can be applied several times, the remaining shortcuts would be removed on the next execution.

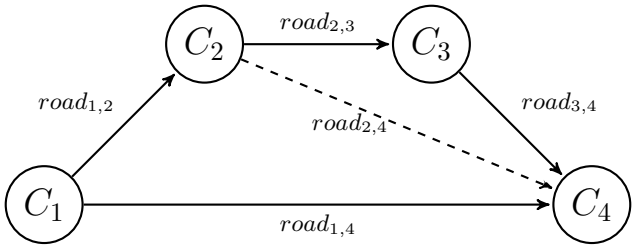


Figure 7: Triangle & Trapeze paths

In order to do that, we had to iterate over clusters. For each cluster, we get every clusters at a distance of one "road" (*i.e.*, clusters are direct neighbours), and we check if it is not also at a distance of two or three. In this case, we remove the edge with the distance of one which is the incorrect "shortcut".

In Figure 7, a triangle is visible with clusters C_1, C_2 and C_4, the shortcut is $road_{1,4}$ and will be removed by our algorithm. It is the same for trapeze, the shortcut is $road_{1,4}$, as a path between C_1 and C_4, already exists.

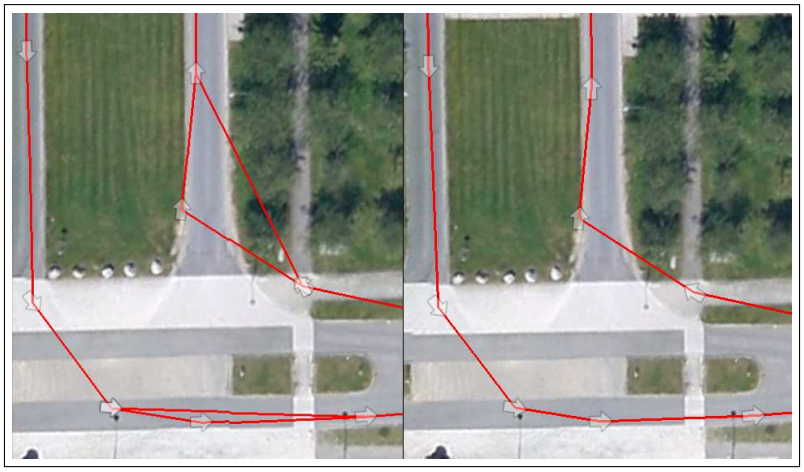


Figure 8: Triangle path - Before (left) & After (right)

In Figure 8, a triangle is visible on the top right of the image, and another at the bottom, but less visible. Our filter deletes the shortcut road, that goes across the trees.

3.5.4 Square Paths

Our last clean-up filter is about square paths. As shown in Figure 9, it is possible to find roads that form a "square". In this case, since we cannot know which clusters are incorrect and we chose to merge the two clusters. Because our fusion updates the cluster location, the new cluster is at the weighted average location between the two merged clusters, according to the enclosed nodes.

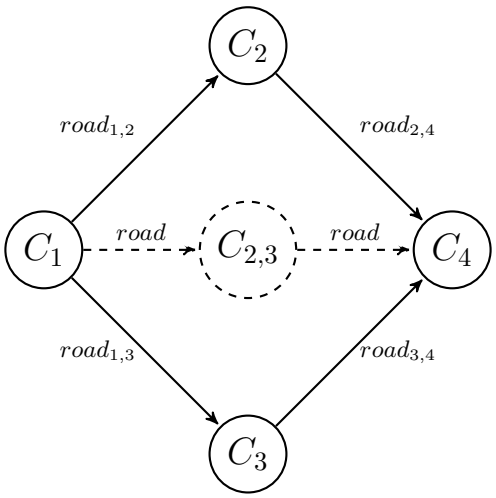


Figure 9: Square paths

This means that if one cluster has one hundred nodes and the other only two, its location will not change much. However, if the two clusters are about the same node number, the new location will be between them. For example, if C_2 and C_3 have the same nodes number, their fusion would create cluster $C_{2,3}$.

Then, we recreate the roads. To do that, we iterate over every clusters, and for each neighbour at a distance of one, we check if they have a same neighbour in common. In this case, if the distance between those two is not too high, we merge them.

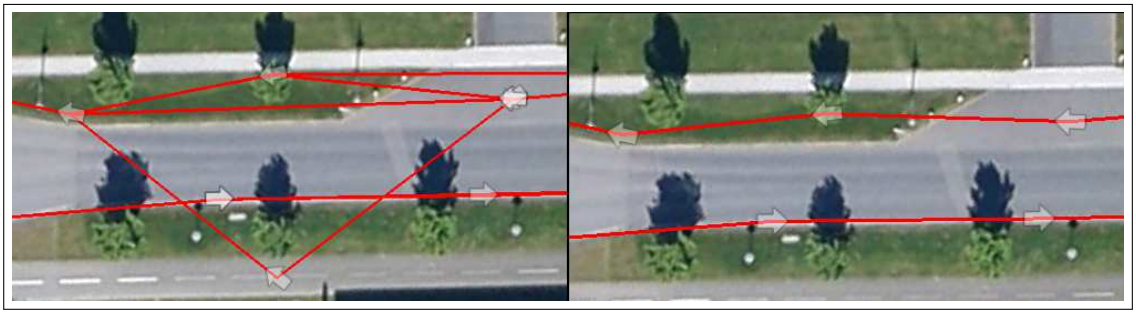


Figure 10: Square path - Before (left) & After (right)

In our datasets, this cleaning is useful when a road has a part with less available satellites, leading to less accurate nodes, and possibly two "roads" on the same path.

As we can see in Figure 10, due to a lack of precision, two lanes were created, and then, the fusion centers the road. The top cluster was slightly down when merged.

4 Discussion

As mentioned in Section 3.1, we used two different datasets to test our algorithms. Our dataset provides the nodes accuracy, which is very useful in our distance computation to infer clusters. An inaccurate node will be added to a cluster even if it is slightly too far. Since a cluster location changes when a node is added, this merging does not add inconsistency to the cluster. Figure 11 depicts the resulting roads.

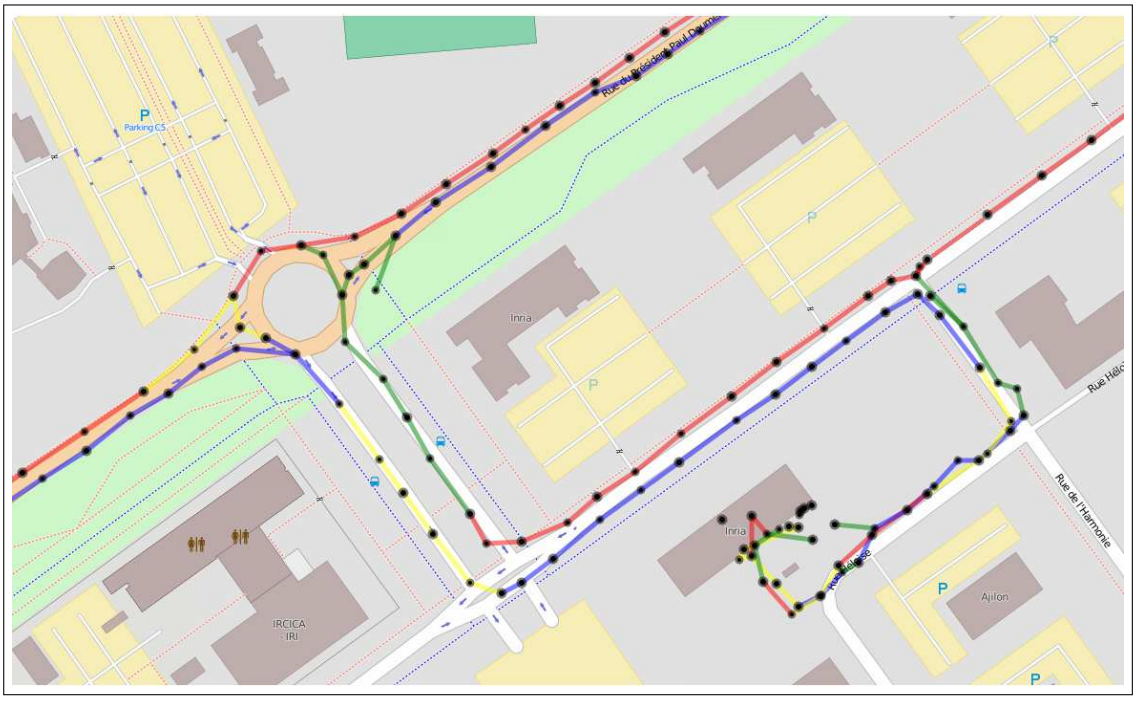


Figure 11: Inferred roads

The Chicago dataset, however, was more difficult to infer, due to the lack of accuracy. We tried several methods, as described in sections 3.3.3 and 3.5. Obviously, results are less accurate but remain correct.

All our formulas and algorithms are customizable, and we tested multiple values for each of those parameters in our work.

The main parameters are:

- Gaussian distance *threshold*, fixed at 0.5 meters,
- Gaussian bearing *threshold*, fixed at 15 degrees,
- Spatial distance weight, fixed at 0.5.

These parameters greatly affect the final result, and may be different depending on the dataset.

5 Conclusion

In this paper, we presented an updated state of the art, as well as a new method for road-maps inference. This technique consists in creating oriented clusters, and linking them to form roads. Compared to state of the art methods, this one is scalable, and stores many information about the clusters and roads, which bring new opportunities.

Indeed, our algorithms are just a proof of concept of roadmaps inference on a large scale, without using image processing. In particular, we can still improve and optimize our models, especially the distance computation and the clustering itself.

Furthermore, some new features could be added, such as stop or red lights detection, but also roads cutting to create routes and navigation helps. Activity detection of the user could also help us to separate the different types of paths, such as bike trails, roads, or pedestrian paths.

References

[1] Gabriel Agamennoni. "Robust inference of principal road paths for intelligent transportation systems". In: *Intelligent Transportation ...* 12.1 (Mar. 2011), pp. 298–308. ISSN: 1524-9050. DOI: 10.1109/TITS.2010.2069097.

[2] *Apache Cassandra*. URL: http://cassandra.apache.org.

[3] *Archery*. URL: https://github.com/meetup/archery.

[4] James Biagioni and Jakob Eriksson. "Map inference in the face of noise and disparity". In: *Proceedings of the 20th International Conference on Advances in Geographic Information Systems - SIGSPATIAL '12* (2012), p. 79. DOI: 10.1145/2424321.2424333.

[5] James Biagioni and Tomas Gerlich. "Easytracker: automatic transit tracking, mapping, and arrival time prediction using smartphones". In: *Proceedings of the 9th ...* (2011).

[6] Stefan Edelkamp and Stefan Schrödl. "Computer Science in Perspective". In: (2003). Ed. by Rolf Klein, Hans-Werner Six, and Lutz Wegner, pp. 128–151.

[7] *ElasticSearch*. URL: http://www.elasticsearch.org.

[8] Robson Leonardo Ferreira Cordeiro et al. "Clustering very large multi-dimensional datasets with MapReduce". In: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '11* (2011), p. 690. DOI: 10.1145/2020408.2020516.

[9] Sergej Fries, Stephan Wels, and Thomas Seidl. "Projected Clustering for Huge Data Sets in MapReduce". In: (2014), pp. 49–60.

[10] *Gremlin - Graph Traversal Language*. URL: https://github.com/tinkerpop/gremlin/wiki.

[11] Younghoon Kim et al. "DBCURE-MR: An efficient density-based clustering algorithm for large data using MapReduce". In: *Information Systems* 42 (June 2014), pp. 15–35. ISSN: 03064379. DOI: 10.1016/j.is.2013.11.002.

[12] *Neo4J - A graph database*. URL: http://www.neo4j.org.

[13] Brian Niehöfer et al. "GPS Community Map Generation for Enhanced Routing Methods Based on Trace-Collection by Mobile Phones". In: *2009 First International Conference on Advances in Satellite and Space Communications* (July 2009), pp. 156–161. DOI: 10.1109/SPACOMM.2009.31.

[14] Poster Paper. "Applying Hadoop's MapReduce Framework on Clustering the GPS Signals through Cloud Computing 1". In: (2013), pp. 644–649.

[15] *PostgreSQL*. URL: http://www.postgresql.org.

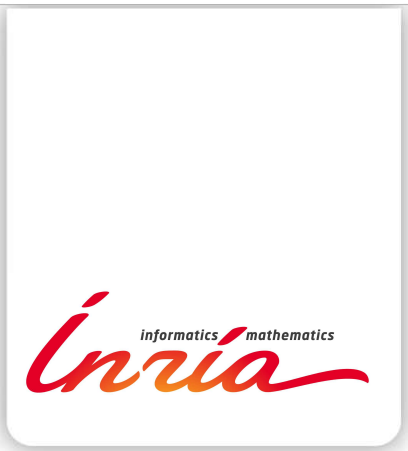
[16] *Rexster - A graph server*. URL: https://github.com/tinkerpop/rexster/wiki.

[17] Stefan Schroedl et al. "Mining GPS Traces for Map Refinement". In: *Data Min. Knowl. Discov.* 9.1 (July 2004), pp. 59–87. ISSN: 1384-5810. DOI: 10.1023/B:DAMI.0000026904.74892.89.

[18] N Schuessler and KW Axhausen. "Processing raw data from global positioning systems without additional information". In: *Transportation Research Record: ...* 2 (2009).

[19] Roger W. Sinnott. "Virtues of the Haversine". In: *Sky Telesc.* 68 (1984), p. 159.

[20] *Titan Distributed Graph Database*. URL: http://thinkaurelius.github.io/titan.



RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399