KTH Computer Science
and Communication

Crowdsourcing public transport data via live mobile tracking

Feasibility study of a system capable of collecting mobile data to build a database of public transit routes, stops and timetables, using machine learning techniques and graph theory

PAUL LAGRÉE

Master's Thesis at Amadeus
Academic Supervisor & Examiner: Stefan Carlsson
Industrial Supervisor: Domenico De Fano

TRITA xxx yyyy-nn

Abstract

Millions of people use public transport systems everywhere in the world, while the number of smartphones connected to the Internet is dramatically increasing. The aim of this thesis is to study, design, and prototype a system to collect data from the mobile devices of regular public transport users and analyse them in order to provide useful information to travellers all over the world. To study the feasibility of this project at a large scale, data will be created to simulate what will be collected via mobile phone applications. The main objective of this paper is to study the effect of data mining techniques on mapping original transport routes with associated timetables.

Referat

Crowdsourcing av kollektivtrafikdata via direkt mobilspårning

Miljontals personer använder kollektivtrafik överallt i världen och antalet smartphones med internetanslutning stiger dagligen. Avsikten med den här avhandlingen är att undersöka, skapa och testa ett system som samlar och analyserar data från mobiltelefoner av kollektivtrafikanvändare såatt anväÃ¤ndbar information kan bli tillgänglig till resande världen över. För att generellt undersöka möjligheten med ett sådant projekt kommer data att skapas för att simulera informationen som är tänkt att senare samlas från resandes mobiletelefoner. Huvudsyftet med den här avhandlingen är att undersöka data mining effekter i kartläggningen av ursprungliga transportsträckor och dess tidtabeller.

Acknowledgements

My first thanks go to my tutor at Amadeus Domenico de Fano who gave me advice and support along the time I spent in the company. I also want to thank Stefan Carlsson for accepting to supervise my thesis as Academic professor.

The project on which relies this thesis was carried out at Amadeus, in Sophia Antipolis, France. This is a 6-month work which was done between August 2013 and January 2014 and along which I met very nice people both among other interns and staff employees. Many thanks to all of them.

Finally, I would like to thank my family who has not stopped supporting me all along my studies.

Contents

Chapter 1

Introduction

1.1 Amadeus

Amadeus is a originally a *global distribution system* (GDS) founded by Air France, Iberia, Lufthansa and SAS, four major European airlines. A GDS is a service that automates transactions between service providers (e.g. airlines) and booking agents. Over the last years, the group has tried to enlarge its services to cover different different aspects of the travel industry. Today, besides its ticketing, pricing and booking services, Amadeus provides additional services to airlines, airports and travel agents, such as tools for flight scheduling, delay managements or departure control.

Figure 1.1: Amadeus logo

Amadeus has also enlarged its offer to other actors of the travel industry such as railways companies and hotels, in order to be able to serve the largest number of actors and improve the whole travel experience. With this aim in mind, covering all the transportation actors from the bus lines to go to the airport to the airlines themselves has become one of the important challenges for Amadeus.

1.2 The public transport problem

Nowadays, public transport data are really scattered. Many companies providing transportation in cities do not share their timetables. For a user, it means that he has to use a specific website (or mobile application) for every city. If there are different operators in the same city, one can even expect to be obliged to use the different services independently, one application for each operator, without any easy connections between them. Some actors offer the possibility to operators to upload their transport network information on services such as Google Maps. However, these data are not largely accessible and rely mostly on operators' will to share their transport data, which brings problems at a world scale.

Thanks to the Open data trend, one can expect that having access to these data will get easier in the future. However, here comes another problem: how can we collect public transport data of cities where bus line maps do not even exist? Dhaka for instance is considered as one of the most blocked city in the world, even though only a low 1% of people owns a car. The bus network is composed of many bus lines operated by numerous different companies. Nevertheless, there was not any map available before a group of MIT students tried to map existing lines with the help of mobile phones [1]. This project was the original idea of the current project hosted at Amadeus and developed in this thesis.

1.2.1 Crowdsourcing mobile data to feed transport transit routes

This thesis relies on the proposal of a platform for manual and automated data collection that would allow service user information aggregation such as GPS trajectories and associated times. Then, these data would be analysed and clustered in order to gradually build a global database of public transportation systems that everyone could access to get information about routes, timetables, etc.

Besides the data collection, algorithms would run on the current state of the database to extract information such as stop locations, stop order in a given direction or transport timetables. The current thesis focuses on this information extraction and proposes algorithms to merge user journeys into lines with stop sequences and associated timetables. The database feeding via mobile phone tracking is not discussed in the current paper.

1.2.2 Problem definition

The current thesis aims at proposing a solution to extract the ordered sequence of stops with their geographic locations, and if possible, the timetable associated to a line. The solution presented in the current thesis relies on a dataset of journey recordings which could be collected with service user participation. Journeys are objects which contain data such as the GPS track followed during the recording and its line number.

Chapter 2

Simulation

To the best of our knowledge, there is no large dataset available containing GPS tracks of journeys on public transports. Vieira et al. [2], use an interesting dataset of 145 moving scholar buses around Athens, Greece [3] in their article about on-line discovery of flock patterns. However, to test the robustness of the method developed in this thesis, we decided to simulate a large dataset using data from the bus line operator around the area of Nice.

2.1 Web transport data extraction

A public company operates in the region of Nice and provides users stop names and locations for each line with the associated timetable on the website `www.ceparou06.fr`. The dataset used in the following work has been created by scraping this website and simulating journeys which would be undertaken on these lines.

After running a script to scrap a bus line, we have in a local database its stop names and locations and the timetable for the whole day (schedule at each stop for every ride). The process can be looped through several days or months to store the timetable for different days (week or weekend) and periods (working period or holidays). We will try to map all these data once we will have simulated mobile users journeys.

2.2 Data simulation

2.2.1 Geographic simulation

The objective was to be as close as possible to reality when simulating data. In order to achieve this, noise and errors have been added to the simulated data. The spatial simulation follows the process explained here. Two stops are randomly chosen following a uniform distribution. Then, all intermediate stops are used to create an ordered list from the departure stop to the arrival stop. Using the open source software OSRM [4] (Open Source Routing Machine) which relies on the

OpenStreetMap data, a physical path is computed for every 2 consecutive stops. The path is transformed into a sequence of points with noise to simulate a GPS signal. The noise has two components:

- noise on a point following a Gaussian distribution
- noise on the distance of 2 consecutive points (also following a Gaussian distribution even if reality would probably not give such a distribution). This noise is supposed to simulate variations of speed along a GPS track.

2.2.2 Timetable simulation

Mobile devices tag GPS signals with timestamps: a time is associated to every spatial location. After generating a noisy path between 2 randomly chosen stops of a given line, we compute a temporal path based on the real timetable scraped in the first part. Once again, noise is added to the simulated temporal sequence of timestamps. Indeed, as many people can experiment it all over the world, public transports sometimes suffer from regular delays due to weather, traffic or even accidents. Once again, noise has been divided into 2 components:

- *accumulated delay* corresponds to the delay which piles up from the beginning of a line until a given point further. It means that the delay at a stop S_i depends only on the delay at the previous stop S_{i-1} to which we add a random number corresponding to the delay accumulated between stops S_{i-1} and S_i (possibly negative if the vehicle caught up
- *uniform noise* corresponds to the small accelerations and slowing downs along a journey. It follows a Gaussian distribution and is added to the accumulated delay.

Obviously, a corrector scans the computed timestamps to check that the generated sequence is strictly increasing and if not, a local correction is done.

This whole process is run for each simulated journey. The whole simulation takes two parameters into account: the number of required journeys and the ratio of wrong journeys (mistakes deliberately shared or not). In our model, mistakes correspond to journeys between two random points around the region of the chosen line. Departure and arrival names are chosen among the real ones. A simulation typically creates 3 000 journeys with 5% of mistakes.

Part I

Spatial mining

Chapter 3

Density-Based Clustering

3.1 Introduction

3.1.1 Motivation

Physical stops (and logical stops on a wider scale) are basic elements which are required to solve most further problems such as finding ordered stop sequences and line timetables. This observation led us to focus on extracting stop locations first.

In order to model our system we assume that for every user journey we would obtain at least departure and arrival stops, together with the journey GPSi tracks. On a large scale, the database contains many journeys for every line with the departure and arrival locations tagged by their name. This chapter aims at proposing a solution to extract an unknown number of physical stops. Indeed, even though examples given to the learner are labelled (stop names), we cannot rely on this information because noise has been introduced to data and because two distinct physical stops may have the same name (on the opposite directions for example). We will consider the current problem as a problem of unsupervised learning, that is to say we will not use name labels attached on stop locations.

3.1.2 Cluster analysis

Unsupervised learning is the problem of finding hidden structures in unlabelled data. In our case, we aim at finding physical stops in a dataset composed of many geographic dots by grouping them in such a way that GPS signals transmitted near the same stop will end up in the same group.

The task of grouping objects from a dataset in such a way that similar ones end up in the same groups is called **clustering analysis**. Various algorithms exist to cluster datasets. However they differ significantly in the method employed to create clusters. The most popular classes of clustering algorithms include:

– *Hierarchical clustering*: It is a family of clustering algorithms which aims at building a hierarchy of clusters usually presented in a tree (also called dendrogram). Two types can be employed to build the tree. The agglomerative strategy merges successively small similar clusters to make bigger ones until all samples end up in a unique cluster. At the beginning, each observation has its own cluster, and by grouping them in successive steps, one can build the final dendrogram. Joe H. Ward, Jr. proposed a general procedure to follow the agglomerative method [5]. An objective function must be maximised to decide which clusters to merge at each step. In the article, Ward uses the error sum of squares as the objective function, but any function can be chosen as long as it "reflects the relative desirability of groupings". The divisive strategy employs the opposite strategy. All observations start in a unique cluster. Clusters are divided successively to build the tree in a top-to-bottom strategy. Once a dendrogram has been built, the user can choose the depth of the tree to use according to a clustering measure. An example of a dendrogram is available in figure 6.1

– *Partitioning clustering*: Also called centroid-based clusterings, these algorithms represent each cluster by a central vector. They produce spherical clusters around k centroids where k is given as an input parameter. The algorithm starts with an initial partition of the dataset D (can be done with a random initialisation) and iterates on the k centroids trying to minimise an objective function based on the cumulated distance of every sample to its centroid. In the famous k-means, the gravity centre is used as centroid of the cluster. Mathematically, k-means can be summarised as the following problem. Given a dataset $D = (\mathbf{x_0}, \mathbf{x_1}, ..., \mathbf{x_n})$ where each $\mathbf{x_i} \in E$ and E is a metric space of dimension d, k-means aims at minimising the within-cluster sum of squares (WCSS):

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x_j} \in S_i} \|\mathbf{x_j} - \boldsymbol{\mu}_i\|^2$$

where each $\boldsymbol{\mu}_i$ is the mean of all observations clustered in set S_i and $S = S_1, S_2, ..., S_k$ is a partition of the dataset S. In the end, each observation is assigned to the cluster's closest centroid. This rule implies that the partition is equivalent to a Voronoi diagram (c.f. figure 3.1). The standard version of k-means was first published in 1965 [6] but Lloyd [7] had already proposed a similar algorithm in 1957 which was not published publicly before 1982.

– *Density-based clustering*: The two previous classes of algorithms have problems with our GPS location dataset. Hierarchical algorithms are quite slow (in general case $O(m^2 \log m)$ where m is the number of samples in the dataset as shown in [8]) and therefore limit the size of datasets used as input. In our case, we aim at clustering relatively big datasets so the hierarchical clustering does not seem to be a good method to process our data. Moreover, choosing the right depth where to cut the dendrogram would be really complicated because data contain noise and because the number of clusters to be found is unknown. Partitioning clustering finds clusters with a specific shape (spheres) and every cluster is included in a cell of a Voronoi partition of the dataset

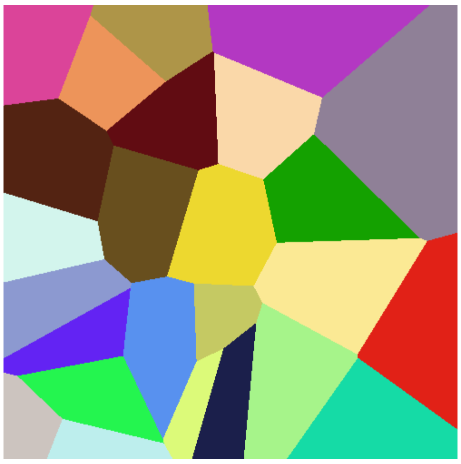


Figure 3.1: A Voronoi diagram.

space. GPS positions agglomerated through mobile user participation may create clusters of any shape and therefore centroid-based algorithms are not a good solution. Moreover, once again, automatically finding k would be a problem because every line is different. A third major class of algorithms has therefore been introduced to "Discover clusters in large spatial databases with noise" in 1996 [9]. This technique relies on a new definition of clusters: regions where the density of samples is much higher than in the rest of the dataset and which are separated by low-density regions. Another important advantage compared to the two previous classes of algorithms is that density-based clustering handles noise in the dataset. In the two next sections, we will focus on DBSCAN and OPTICS, the two major methods of density-based clustering.

3.2 DBSCAN

In this section, we remind the main definitions behind DBSCAN clustering. In the first part, we introduce definitions to handle densities. We give the DBSCAN algorithm in the second part. The whole section relies on the original article [9]. The samples from the dataset are elements of a metric space E with a metric d.

3.2.1 Density measures for clustering

Intuitively, we may consider that a group of points forms a dense cluster if, given a surface, more than a certain number of points are inside. The DBSCAN algorithm relies on this intuition. However, it uses a local definition of density to find clusters by expanding small dense areas. We defined the ϵ-neighbourhood of a point P in E as the closed ball of radius ϵ centred at P.

Definition 1 (ϵ-neighbourhood)**.**

$$N_\epsilon(P) = \{Q \in E \mid \ d(P,Q) \leq \epsilon\}$$

The main idea given by DBSCAN is the definition of a density-reachability, allowing us to connect points to each others.

Definition 2 (density-reachability)**.** A point P is *directly density-reachable* from a point Q with respect to ϵ and $minPts$ if:

1. $P \in N_\epsilon(Q)$
2. $\text{Card}(N_\epsilon(Q)) \geq minPts$

A point P is *density-reachable* from a point Q with respect to ϵ and $minPts$ if: it exists $n \in \mathbb{N}$, $P_1, ..., P_n$ with $P_1 = Q$ and $P_n = P$ such that for all (P_{i+1}, P_i), P_{i+1} is directly density-reachable from P_i.

Note that the density-reachability is not a symmetric concept. Indeed, if P is density-reachable from Q with respect to ϵ and $minPts$, it means that $\text{Card}(N_\epsilon(Q))$ is superior to $minPts$, but it does not necessarily imply that the cardinal of the ϵ-neighbourhood of P is also greater than $minPts$. Two points are said *density-connected* if there are density-reachable from a same point $R \in E$, with respect to ϵ and $minPts$.

A cluster C is then defined as a maximal set of density-connected points: all its elements are density-connected to each others and any other point in E is not density connected to any element in C. Points which do not belong to any cluster are called *noise*. Other points belong to a cluster and can be divided in both groups defined below:

Definition 3 (core points and border points)**.** A point P is a *core point* if P belongs to a cluster C and if $\text{Card}(N_\epsilon(P)) \geq minPts$.
A point P in a cluster C which is not a core point is called a *border point*.

3.2.2 Algorithm

DBSCAN is given in the algorithm 1. An arbitrary point P is chosen and the algorithm retrieves all density-reachable points from this point. If P is a core point, the algorithm extends the point to an entire cluster, looking for every point Q so that P and Q are density-connected to each other. The whole algorithm ends when all points have been analysed (by a core point expansion, or by the big loop in the main procedure). In the end, every point is tagged as a *core object*, a *border object* or *noise*.

3.2.3 DBSCAN limitations

The main disadvantage of DBSCAN is its lack of flexibility to handle heterogeneity inside spatial data. Clusters of different densities may be discovered as a

Algorithm 1 DBSCAN algorithm

```
procedure DBSCAN($SetOfPts, \epsilon, minPts$)
    $clusterId \leftarrow nextId(NOISE)$                ▷ if $NOISE = 0$, first cluster is 1
    for each point $P$ in $SetOfPts$ do
        if $P.ClId = null$ then
            if $ExpandCluster(SetOfPts, P, clusterId, \epsilon, minPts)$ then
                $clusterId \leftarrow nextId(clusterId)$
            end if
        end if
    end for
end procedure

procedure ExpandCluster($SetOfPts, Point, clId, \epsilon, minPts$)
    $NeighbourPts \leftarrow SetOfPts.regionQuery(Point, \epsilon)$
    if $sizeof(NeighbourPts) \leq minPts$ then                ▷ no core point
        $Point.clId \leftarrow NOISE$
        return $false$
    else            ▷ all points in NeighbourPts are density-reachable points
        $SetOfPts.changeClIds(NeighbourPts, clId)$
        remove $Point$ from $NeighbourPoints$
        while $NeighbourPts \neq \emptyset$ do
            $currentP \leftarrow NeighbourPts.first()$
            $Result \leftarrow SetOfPts.regionQuery(currentP, \epsilon)$
            if $sizeof(Result) \geq minPts$ then
                for each point $resultP$ in $Result$ do
                    if $resultP \in \{null, NOISE\}$ then
                        if $resultP = null$ then
                            append $resultP$ to $NeighbourPts$
                        end if
                        $resultP.ClId \leftarrow clId$
                    end if
                end for
            end if
            remove $currentP$ from $NeighbourPts$
        end while
        return $true$
    end if
end procedure
```

single one if the input parameters are chosen for the least dense cluster. On the opposite, if we adjust parameters to fit dense clusters, low-density clusters will be considered as noise by DBSCAN.

OPTICS was proposed in 1999 to fix these drawbacks. The following section covers this second density-based clustering algorithm.

3.3 OPTICS

OPTICS was introduced by the same group of researchers, including two common authors [10], and relies mostly on the same definitions as DBSCAN. As said previously, it gives more flexibility to input parameters and handles clusters of different densities.

3.3.1 Idea underlying OPTICS

The core idea of OPTICS is the construction of a diagram giving a score to each object (called *reachability-distance*) by scanning the dataset. The reachability-distance is defined as follows:

Definition 4 (reachability-distance)**.** Let O be a core point and P a point. The *reachability-distance* of P with respect to O, ϵ and $minPts$ is defined as:

$$reachability - distance_{\epsilon,minPts}(P,O) = \max\{MinPts - distance(O), d(O,P)\}$$

where the n-distance of a core point is the distance from the point to its n-th farthest neighbour. The reachability-distance is **undefined** for non core points.

Intuitively, the reachability-distance corresponds to the distance between two points, except if P is too close to O, in which case we normalise the distance by the core distance of O ($MinPts - distance(O)$).

3.3.2 Algorithm

OPTICS is given in the algorithm 2. Every point in $SetOfPts$ has its $reachaDist$ attribute value initialised to $null$. Then all points are scanned, using a similar process to DBSCAN. The $n - distance$ used in the algorithm corresponds to the $minPts - distance$ used in the previous definitions. After running the algorithm on $setOfPts$, all points have non undefined values to their $reachaDist$ attribute value.

Using the ordered list created to save the scan order, we can then build a diagram summarising the effect of the algorithm (c.f. figure 3.2). On the x-axis, we represent all scanned points in the visit order. On the y-axis is plotted the reachability-distance. Valleys and peaks stand out from the diagram called *reachability plot*. Clusters correspond to valleys and can be extracted easily by choosing a threshold shown as a blue line in the figure.

Algorithm 2 OPTICS algorithm

```
procedure OPTICS($SetOfPts, \epsilon, minPts$)
    Create an empty list $orderedList$
    for each point $P$ in $SetOfPts$ do
        if $P.processed = false$ then
            $NeighbourPts \leftarrow getNeighbours(SetOfPts, P, \epsilon)$
            $Point.processed \leftarrow true$
            Append $P$ to the $orderedList$
            $orderedSeeds \leftarrow$ empty queue
            if $Point$ is a core point then
                $P.coreDistance \leftarrow n - distance(P, setOfPts, minPts)$
                $orderedSeeds.update(NeighbourPts, P)$
                while $orderedSeeds$ not empty do
                    $P' = orderedSeeds.pop()$          ▷ removes and returns first item
                    $NeighbourPts' \leftarrow getNeighbours(SetOfPoints, P', \epsilon)$
                    $P'.processed \leftarrow true$
                    Append $P'$ to the $orderedList$
                    if $P'$ is a core point then
                        $P'.coreDistance \leftarrow n - distance(P', setOfPts, minPts)$
                        $orderedSeeds.update(NeighbourPts', P', \epsilon, minPts)$
                    end if
                end while
            end if
        end if
    end for
end procedure

procedure ORDEREDSEEDS::UPDATE($NeighbourPts, Point$)
    $coreDistance \leftarrow Point.coreDistance$
    for each $Q$ in $neighbourPts$ do
        if $Q.processed = false$ then
            $currReachaDist \leftarrow \max\{coreDistance, d(Point, Q)\}$
            if $Q.reachaDist = null$ then                    ▷ $Q$ not in $orderedSeeds$
                $Q.reachaDist \leftarrow currReachaDist$
                Insert $Q$ in $orderedSeeds$ given $Q.reachaDist$
            else
                if $currReachaDist < Q.reachaDist$ then                  ▷ Update
                    $Q.reachaDist \leftarrow currReachaDist$
                    Reorder $orderedSeeds$ given previous update
                end if
            end if
        end if
    end for
end procedure
```

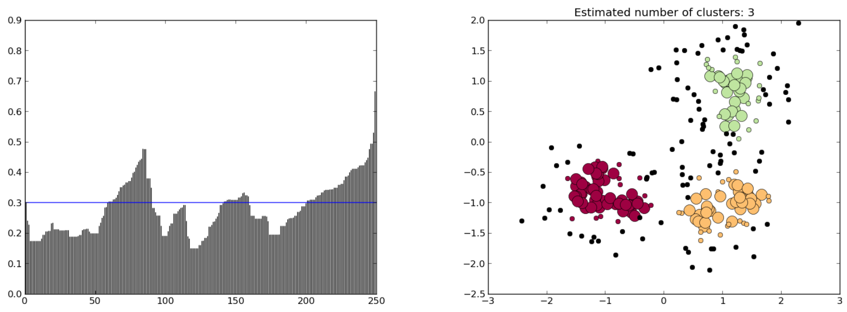


Figure 3.2: Reachability plot with clusters found by OPTICS

Intuitively, the reachability-distance of a point P corresponds to the distance to the closest point among the already visited ones, normalised by the distance of its $minPts$-th closest neighbours. Note that some points may keep a *null* reachability-distance (the first point if it is a core point for example). This is not a problem since we consider that points with such a value has an infinite reachability-distance which is then superior to the threshold shown in figure 3.2.

3.4 Implementation and results

In the model presented in the previous chapter, the following information is available for each journey:

– Departure stop name
– Arrival stop name
– noisy GPS path (sequence of locations/times from departure stop until arrival stop)

3.4.1 Stop clustering with DBSCAN

The first step is to cluster departure and arrival points, every group of points representing a physical stop. Indeed, even if every GPS point has a margin error close to 10 meters, these points still create groups of higher density around the real stops. To do so, the DBSCAN algorithm is used since it allows noise and unknown number of clusters. OPTICS was not used in this work because it is not available on the **scikit-learn** [11] master branch yet, the python library used in this work. The results of a run of the algorithm on line 230 simulated data (used as the main example in this paper) is given in figure 3.3.

The x-axis corresponds to the longitude, the y-axis to the latitude. The small black dots are noise, approximately filling the region around the bus line. We can notice an empty area on the bottom-right of the map, it corresponds to the sea. The medium-size coloured dots correspond to stops which have been discovered by the algorithm whereas the big-size dots correspond to real stops (coordinates used at the simulation step). As we can see, the result is quite satisfactory and most isolated stops are found.

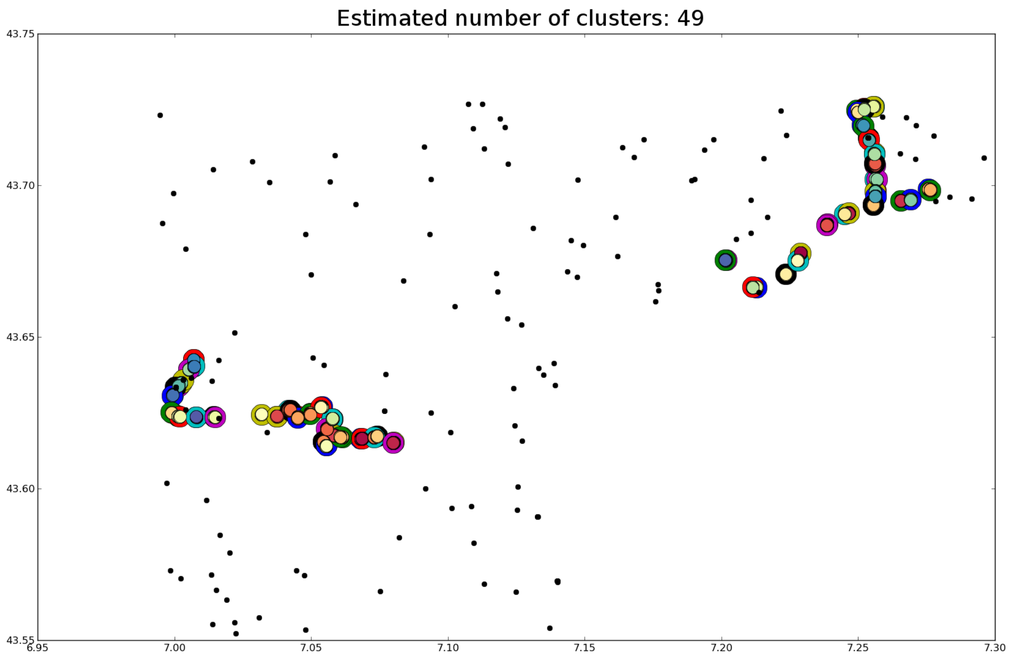


Figure 3.3: DBSCAN on departure and arrival locations

Chapter 4

Graph representation

In the previous chapter, we introduced a method to extract the physical stops of a public transport line. After running the algorithms, almost all regularly crossed stops are found. The objective in this chapter is to group together stops in the same direction and to design the line structure in a given direction. Using a graph representation is quite natural since a single route can be split into two branches, or at the opposite, two branches can be grouped together at a given stop. Graphs can easily model such a structure.

4.1 Graph theory

In this section, we remind some graph theory basics. Readers familiar with this topic may skip this section and go directly to the following section about **Graph Communities**.

A graph G is a pair of sets (V, E) where V is the set of *vertices* (also called *nodes*) and E is the set of *edges*. The number of vertices $n = Card(V)$ is the *order* of the graph while the number of edges $m = Card(E)$ is called the *size* of the graph. An element of E is a pair of vertices (v, w) with v and w in V and corresponds to a connection between these two vertices. If the graph is *undirected*, the two endpoints of an edge are unordered so we can write $\{v, w\}$ instead of (v, w). On the opposite, the pair is ordered in the case of a *directed* graph. We say a graph is *complete* if, for each pair of vertices v and w in V, there is an edge (v, w) in E connecting them. A *weighted* graph is a graph on which a function weight $w : E \longrightarrow \mathbb{R}$ is defined. We can think about the distance as the weight function of a graph representing geographic locations as vertices and routes as edges. The *adjacency matrix* A of a graph G is a matrix of size $n \times n$ where

$$a_{ij} = w(i, j) \text{ for all } (i, j) \in V^2$$

If the graph is not weighted, w is just the function which outputs 1 if there is an edge between the two vertices and 0 otherwise.

A *subgraph* S of G is a graph whose vertices and edges constitute respectively a subset V_S of V and a subset E_S of E, such that for each (v, w) in E_S, v and w are in V_S.

The *degree* of a vertex v is the number of edges which have v as an endpoint. In case of a directed graph, we define the *indegree* as the number of edges incident on a given vertex v (v is then called a target for these edges) and the *outdegree* as the number of edges leaving from a given vertex (v is called the source).

A *walk* is an alternating sequence of vertices and edges starting and finishing with a vertex. Each edge has as endpoints the previous and following vertices in the walk sequence. The *length* of a walk corresponds to the number of vertices.

4.2 Graph community detection

Using departure and arrival locations of every journey, we build a directed graph whose vertices are stops and whose weights correspond to the number of journeys recorded between sources and targets. This approach allows us to avoid to connect stops which are crossed but where the vehicle does not stop on a given direction. If a user checks in at intermediary stops, the journey is stored as a set of small journeys, each one connecting two intermediary stops. The created graph contains logically two dense areas (dense meaning containing many edges), one for each direction. This section aims at finding these two areas with an algorithm.

4.2.1 Communities and modularity

Dense areas correspond to communities in the graph vocabulary. However, defining formally the notion of community is difficult as explained in [12] since many definitions have been proposed. We will use in this paper the definition used in the algorithm detailed in the following section. A partition $\mathcal{P} = C_1, ..., C_k$ is considered as a good partition of V if it has a high *modularity*. The algorithm aims at finding the partition $\mathcal{P}$ which maximises this measure. Less formally, a good community structure has many edges inside every C_i and few edges crossing communities. The modularity, introduced in [13], measures this property.

Definition 5 (modularity)**.** Let us consider a matrix $\mathbf{e}$ of size $k \times k$. Every element e_{ij} corresponds to the number of edges which link the communities i and j normalised by the total number of edges m. Thus $\mathbf{e}$ is a symmetric matrix whose elements are real values in $[0, 1]$. The trace of this matrix $\operatorname{Tr}\mathbf{e} = \sum_{i=1}^{k} e_{ii}$ gives the fraction of edges which link vertices inside the same community. Let us define r_i as the sum of the i-th row of $\mathbf{e}$, it corresponds to the fraction of edges pointing to a vertex in community i. The modularity Q is defined by

$$Q = \sum_{l \in [\![1,k]\!]} \sum_{(i,j) \in C_l^2} \left[a_{ij} - \frac{k_i * k_j}{(2m)(2m)} \right] = \sum_{i=1}^{k} \left(e_{ii} - r_i^2 \right) = \operatorname{Tr}\mathbf{e} - \left\| \mathbf{e}^2 \right\| \tag{4.1}$$

where k_i stands for the degree of vertex i.

We may think that the trace of $\mathbf{e}$ would give a satisfactory measure of a community structure since it corresponds to the fraction of edges which link vertices of same communities. However, in case of a partition placing all vertices in a single set, the trace would be maximum whereas, obviously, this is not what we aim at. The trick used here is to subtract to the trace the fraction of edges connecting vertices in the same community in a network N generated as follows. N is a graph with the same node distribution as G. It means that there is a bijection mapping N vertices and G vertices: each vertex in N corresponds to a vertex in G and share its degree. However, the edge repartition is different in N. The edge connections are generated randomly, with no regard to communities. When generating a new edge to N, the only constraint is to respect the degrees of nodes: as soon as a vertex u has a degree $d_G(u)$ in N, no further new edges are connected to it.

In the case of a unique community, the modularity becomes equal to equal to zero with this definition. Indeed, the fraction of edges inside communities in N and G remains the same. With a good community structure however, the first term should be higher since a random distribution of edges will make the communities less dense.

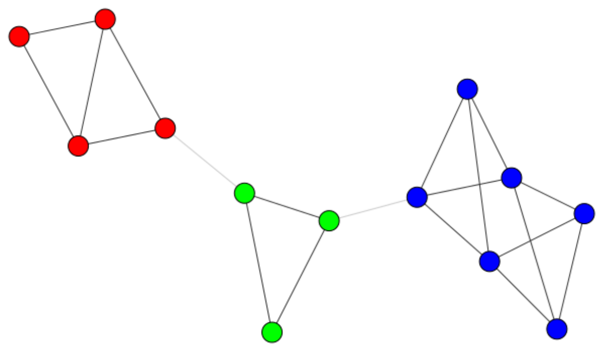


Figure 4.1: A graph containing three communities

Schaeffer presents in a survey most graph clustering (community detection) methods [14]. In this paper, we focus on the method presented in 2005 by P. Pons and M. Latapy [12] and relying on random walks.

4.2.2 Random walks

Let us consider a directed graph G with positive weights and characterised by an adjacency matrix A. A random walk of length l is a sequence of l connected edges (the target of e_i is the source of e_{i+1}). A random walk of length l is built as follows: at each step, a walker at a vertex v chooses randomly among the neighbours of v (vertices directly linked to v) the next vertex towards which it moves. The walker repeats this process until l edges have been crossed, creating a *markov chain* of length $l+1$ whose states are vertices. The next vertex is chosen using the transition matrix T where an element $t_{ij} = \frac{w(i,j)}{\sum_{u \in Neighbourhood(i)} w(i,u)}$. This definition favours the walker to move towards vertices which are linked to the current vertex by an edge of high weight. If two vertices are not linked in G, the edge connecting them has a zero weight.

The general idea of the algorithm is that short walks should end up in the same community. A good community structure has highly connected vertices within communities, thus, short random walks should stay inside the community. One can prove that when the length of a walk tends towards the infinity, the probability of being at a vertex v does not depend on the starting vertex u. We understand easily why we use short walks. On the other hand, if they are too short, they do not reflect the community structure. Typically, a value of 3 or 4 can be used.

4.2.3 Distance on vertices using random walks

Let us introduce a distance d_g between vertices. Two vertices from the same community should have a small distance whereas two vertices in different communities should have a large distance.

Let us consider a matrix T^t of size $n \times n$ where n and t are integers. Each element T^t_{ij} of the matrix T^t corresponds to the probability of going from vertex i to vertex j through a random walk of length t. This matrix can be computed easily since it is the matrix T (seen in previous section) to the power t.

Definition 6 (Distance d_g on vertices)**.** Let us consider i and j two vertices of a graph G. We define d_g by

$$d_g(i,j) = \sqrt{\sum_{k=1}^{n} \frac{(T^t_{ik} - T^t_{jk})^2}{d(k)}} = \left\| D^{-\frac{1}{2}} T^t_{i\bullet} - D^{-\frac{1}{2}} T^t_{j\bullet} \right\| \tag{4.2}$$

where $d(k) = \sum_{u \in Neighbourhoodk} w(k,u)$, D is the diagonal matrix whose elements are $d_{ii} = d(i)$ and $T^t_{i\bullet}$ is the $i-th$ row of the matrix T^t (transposed into a column vector).

The main idea behind this definition is that vertices in the same community should have similar probabilities to reach any other vertex in the graph in a walk of length t. Mathematically, this property corresponds to similar values of T^t_{ik} and T^t_{jk}.

Let us now extend this definition to communities. The distance $d_g(C_1, C_2)$ measures an average value of the previous definition. To do so, we define the variable T^t_{Cj} which corresponds to the probability of reaching vertex j from any vertex in the community C in a random walk of length t. This probability can be computed mathematically by:

$$T^t_{Cj} = \frac{1}{|C|} \sum_{i \in C} T^t_{ij} \tag{4.3}$$

Let us define the distance $d_g(C_1, C_2)$ between two communities by:

Definition 7 (Distance $d_g(C_1, C_2)$ on communities)**.** Let us consider two subsets $C_1, C_2 \subset V$. We define d_g (same notation as before, but distance on communities this time) by:

$$d_g(C_1, C_2) = \sqrt{\sum_{k=1}^{n} \frac{(T^t_{C_1 k} - T^t_{C_2 k})^2}{d(k)}} = \left\| D^{-\frac{1}{2}} T^t_{C_1 \bullet} - D^{-\frac{1}{2}} T^t_{C_2 \bullet} \right\| \tag{4.4}$$

This definition allows us to define a distance between a vertex u and a community C by using $\{u\}$ as a community in the equation 4.4.

4.2.4 Algorithm

We now have a distance between all vertices in the graph. It has become a simple problem of clustering where samples are vertices and distance d_g. The authors of the article [12] propose a hierarchical method using an agglomerative approach to cluster vertices.

As we usually proceed in agglomerative methods, we start with a partition where every vertex has its own community. At every step, we merge the two communities which verify the criterion used in Ward's method (minimising the squared distances sum between each vertex and its community). While merging these two communities, we update the distances between communities. The process is not so computationally heavy because most of the needed values at step k (like the squared distances sum of a community C_i) were already computed at the previous step (in this case, if community C_i has not been merged, we do not need to compute this value again). We then understand easily that only updates on merged communities have to be computed. And this computation is easy to compute as a linear combination of the two merged communities. The above algorithm allows us to create a complete dendrogram of the graph with n leafs grouping in $n-1$ steps into a single community containing all vertices. We cut the dendrogram at the depth which maximises the **modularity** (see subsection 4.2.1).

4.3 Implementation and results

We saw at the previous chapter how we can cluster GPS signals to extract physical stop locations. However, all stops found on a given line with a density-based algorithm are mixed together, whatever their direction. By using connections between stops, we aim at finding two main communities in case of two directions.

4.3.1 Stop clustering according to direction

At this step, we have a set of stops not yet ordered with any logical representation. The final objective of this chapter is to create an ordered sequence of stops in a given direction.

Let us consider a set of physical stops S of a given line. We build a graph G whose vertices are the stops in S and for which each edge e_{ij} connecting i and j has a weight corresponding to the number of journeys from i to j. For example, if n journeys connect stop A to stop B, the weight of the directed edge e_{AB} will be n.

The graph population algorithm can be described as follows:

Algorithm 3 Graph population algorithm

```
procedure PopulateGraph($S, SetOfJourneys$)
    $G \leftarrow$ Graph of $|S|$ vertices
    for each journey $\boldsymbol{j}$ in $SetOfJourneys$ do
        $departure \leftarrow Label(\boldsymbol{j}.departure)$        ▷ $Label$ is given by DBSCAN step
        $arrival \leftarrow Label(\boldsymbol{j}.arrival)$
        $edge \leftarrow G.getEdge(departure, arrival)$
        if $edge$ is null then
            Add edge to $G$ between departure and arrival of weight 1
        else
            Increment $edge.weight$ by 1
        end if
    end for
    return $G$
end procedure
```

The created graph is a **directed** graph. However, we can expect that there are few opposite edges in the graph since a stop A which reaches a stop B should not allow the opposite edge (our stops already contain a direction). We then simply apply the walktrap algorithm on our created graph and extract the community structure which maximises the modularity. An example of this method applied on our simulated dataset is given in figure 4.2.

As we can see, some stops have not been well split at the previous step (Les Belugues in figure 4.2, the stop highly connected to both communities). We can split stops significantly connected to both communities into two new vertices, one for each community. When adding a new stop to a community, only edges connecting the original stop to another vertex in the same subgraph as the new node are added. After this step, we have two sets of stops in most cases, one set for each direction. The objective in the next section is to sort them into a well designed graph, as close as possible to the official transport line. We will work on every community separately.

4.3.2 Public transport route design

Let us consider a set of n stops which have been clustered together through the previous step. Let us create a new graph G_2 with these stops, using the whole information contained in the journey GPS track. Indeed, the previous graph used only the departure and arrival stops data, avoiding heavy computation. For each

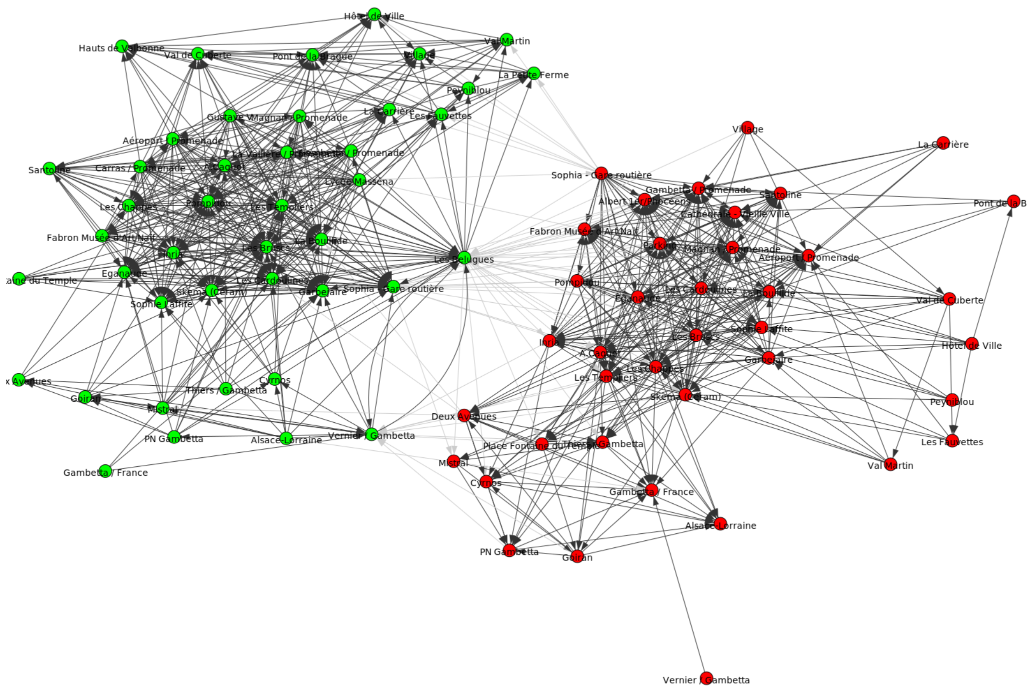


Figure 4.2: Two communities found on line 230.

journey, all observations are run through and every time an observation is both close enough to an existing stop and in the good direction, the previous crossed stop is connected to this new stop (if the edge already exists, its weight is incremented). Obviously, a stop cannot be connected to itself.

This new population algorithm is really slow with a dataset of 3 000 journeys for approximately 300 000 GPS signals. In our Python implementation, more than 1 minute is needed to create the graph, but no time has been spent to optimise this critical section. A future work may improve this algorithm by using a better representation of data (like *k-d tree* for stops to avoid entire loop for each new observation) or coding this critical segment in C. We are quite sure that many improvements can be done here, but we did not want to focus on these questions to go further in our work.

After running the population algorithm, G_2 is a highly connected directed graph. It contains cycles and many unwanted edges with low weights. This behaviour has not been well explained, but it seems that it comes from simulation issues. An example of bug encountered was a strange path taken to reach a given stop. We did not understand the reason of this unexpected behaviour from OSRM, but it turned out that the stop towards which the GPS signal went was on a junction and OSRM did not choose the right road. The algorithm to populate the graph was run on a few real journeys and gave almost perfect sequences of stops (one stop skipped maximum for a given sample). To conclude, most issues seem to come from the simulation step (OSRM or high noise). To continue our work, a real dataset will be required.

First, we clean our graph G_2 from low-weighted edges. Given a fixed threshold, every edge whose weight is inferior to this number is removed from the graph (connectivity must be preserved however). After this step, G_2 is a graph already really close to what we aim at building. Nodes with null indegree (resp. outdegree) correspond to sources (resp. targets) of the line. We then run a longest path algorithm between every source to every target to clean our graph from edges connecting stops i to $i+2$ which correspond to GPS signals which have not emitted around stops $i+1$. We chose the Bellman-Ford algorithm since it allows negative weights. The shortest path found by this algorithm with negative weights corresponds to the longest path with positive weights. However, this algorithm is not robust to cycles so we first clean G_2 from 2 and 3-cycles. For every p-cycle, we create p graphs: one for each edge removed. The solution kept in the end is the one giving the longest paths between sources and targets. Figure 4.3 gives the result of our algorithm on 3 000 journeys (for both directions).

These figures highlight two main points:

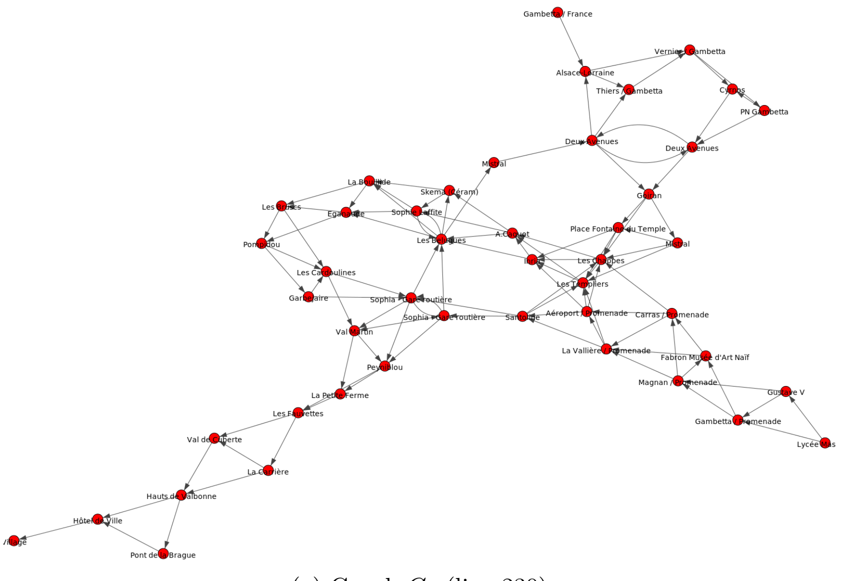
– G_2 is really dirty before cleaning. This strange behaviour corresponds to the **"simulation curse"** as we explained before. We expect reality to give much cleaner graphs (resulting in better final designs after running algorithms).

– after running the algorithms, we can see an isolated node (null indegree) directly connected to a big branch. This node should actually remain connected as a target by an edge. We can expect a too heavy cleaning on low-weighted edges. Once again, we hope that reality will give better results.

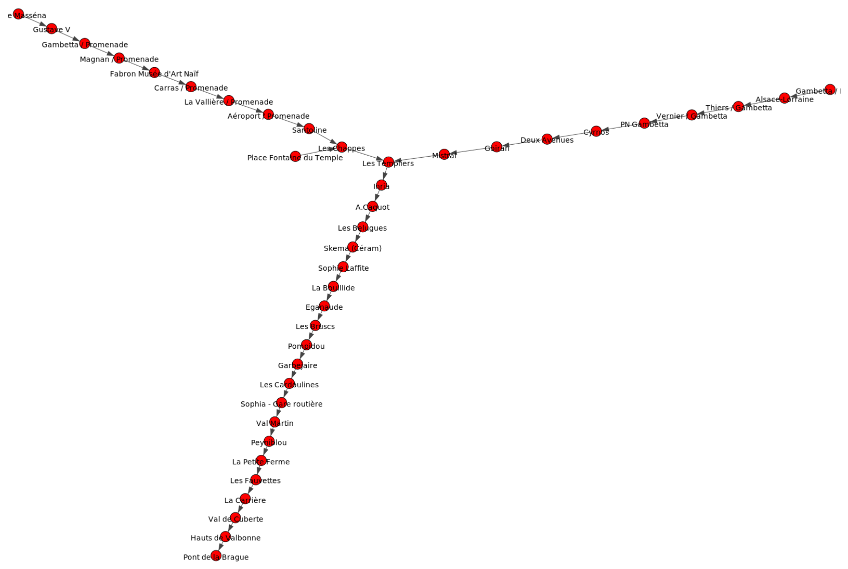
4.4 Conclusion

We now have built a graph for every direction for a given line. This procedure has been tested on several simulated datasets, and small mistakes remain on most results. We believe that these mistakes are mostly created by imperfections from our generator. Even if it is true, we still expect other issues to come up with real dataset. However, as we explained before, real tracks seemed to give interesting sequences of crossed stops (what introduced problems with simulated GPS signals).

In the next part, we will focus on spatio-temporal clustering techniques to build reliable timetables for transport lines.



(a) Graph G_2 (line 230)



(b) G_2 after cleaning and Bellman-Ford

Figure 4.3: Reconstruction of the structure of line 230

Part II

Spatio-temporal mining

Chapter 5

Spatio-Temporal Clustering

GPS signals give both spatial and temporal information. Every location recorded by the mobile application is associated to a **timestamp** such that each journey in the database is a sequence of geographic observations at given times. In the first part, we focused on spatial data to build logical designs of transport lines. In this second part, we will aim at building robust timetables for lines given sets of journeys. This chapter first describes two well-known techniques for clustering spatio-temporal tracks. The second section describes how we implemented the CMC technique in our work and what results we got with our simulated datasets.

5.1 Introduction

Spatio-temporal clustering is the process of grouping objects containing both spatial and temporal information based on similarities to be defined. As explained in [15], different definitions have been proposed for the objects and the criteria used to define similarity between them. This survey on spatio-temporal clustering techniques published in 2010 references different algorithms for several applications:

– *trajectory clustering:* extraction of trajectories which can be a generalisation of different trajectories in the dataset. In our example, it may be a route a bus follows in a given direction.
– *extracting important locations:* extraction of important places such as famous buildings, tourist locations. Given a dataset of trajectories in a city, these algorithms aim at finding important areas. Naturally, the importance of a location is something really vague and different definitions can be given, resulting in several algorithms.
– *trajectory patterns:* extraction of movement patterns frequently observed.
– *moving clusters:* extraction of groups of trajectories which move together at the same location and time. This is the problem we want to solve with our journeys. Three algorithms solving this problem are referenced in [15]. The next section details the solution given in [16] (2008) which relies on [17] (2005). The third article about moving cluster detection referenced in the survey is

quite different since it gives a solution which allows on-line feeding of inputs whereas the two previously cited articles requires an entire dataset before running the algorithm.

5.2 Coherent Moving Cluster (CMC)

The CMC algorithm is similar to the methods for discovering moving clusters in spatio-temporal data [17]. The main idea of this algorithm is to run DBSCAN on several consecutive snapshots (trajectories O at a chosen time t). The consecutive clusters found by DBSCAN should be similar since elements of moving clusters are likely to stay close in the spatial vector space at consecutive steps.

5.2.1 Definitions

As said before, the definition of a "moving cluster" can be very different from a paper to another. In [16], a moving cluster is called a *convoy* and is defined as follows:

Definition 8 (Convoy)**.** Let us consider a set of trajectories O, a real ϵ and two integers k and $minPoints$. An object c is a convoy with respect to ϵ, k and $minPoints$ if:

1. c has at least k consecutive clusters $c_t, c_{t+1}, ..., c_{t+k-1}$ found by DBSCAN with respect to ϵ and $minPoints$.
2. The following inequation is satisfied by the k consecutive clusters

$$|c_t \cap c_{t+1} \cap ... \cap c_{t+k-1}| \geq minPoints$$

The integer k is called the *lifetime* of the convoy.

In [17], a moving cluster is defined differently. An inequation must be verified with a chosen threshold θ too. However, the inequation is tested with the intersection of 2 consecutive clusters, possibly N times with N an integer. In the definition 8, an inequation is tested just once with the intersection of k clusters. It means that many trajectories have to move together during a lifetime k whereas with the other definition, the components of the moving cluster can be completely different at the beginning and at the end. As expected, this method relies on density-based clustering.

5.2.2 Algorithm

In the original paper, all trajectories have spatial positions at regular timestamps $(t, 2t, ..., nt)$. The following algorithm 4 is a slightly different version allowing real temporal domain. Instead of having discrete timestamps, we create a partition of our real temporal domain. The created sets must be intervals of constant length.

Algorithm 4 CMC algorithm

```
procedure CMC($O, minPoints, k, \epsilon$)
    $V \leftarrow \emptyset$                                        ▷ Convoy candidates
    for each time interval $T$ (in ascending order) do
        $V_{next} \leftarrow \emptyset$
        $O_T \leftarrow EmptyOrderedSet()$
        for each trajectory $o$ in $O$ do
            if $o$ has observation in $T$ then
                append average $o$ position within $T$ to $O_T$
            end if
        end for
        if $O_T.size < minPoints$ then
            Skip iteration
        end if
        $C \leftarrow DBSCAN(O_T, \epsilon, minPoints)$
        for each convoy candidate $v \in V$ do
            $v.assigned \leftarrow false$
            for each cluster $c \in C$ do
                if $|c.objects \cap v.objects| \geq minPoints$ then
                    $v.assigned \leftarrow true$
                    $v.objects \leftarrow |c.objects \cap v.objects|$
                    $v.endTime \leftarrow T.rightBound$
                    increment $v.lifetime$ by 1
                    append $v$ to $V_{next}$
                    $c.assigned \leftarrow true$
                end if
            end for
            if $v.assigned$ is $false$ and $v.lifetime \geq k$ then
                append $v$ to $V_{next}$
            end if
        end for
        for each cluster $c \in C$ do
            if $c.assigned$ is $false$ then
                $c.startTime \leftarrow T.leftBound$
                $c.endTime \leftarrow T.rightBound$
                append $c$ to $V_{next}$
            end if
        end for
        $V \leftarrow V_{result}$
    end for
    return $V_{result}$
end procedure
```

We then compute an "average position" of trajectories at every time interval which can be the mean of all spatial observations in a given interval).

This version of the algorithm needs an extra parameter for creating the partition of the temporal domain (time interval length). Moreover, this method requires much computation since DBSCAN is run possibly $\frac{24\times60\times60}{l}$ times where l is the length of time intervals. [16] proposes a family of algorithms to improve CMC. The following section presents CuTS, a convoy discovery method using trajectory simplification.

5.3 Convoy discovery using Trajectory Simplification (CuTS)

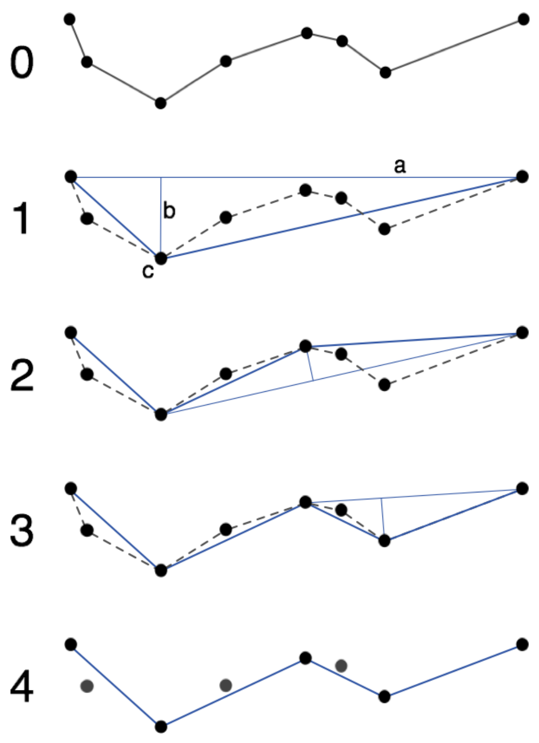
CuTS is an algorithm which gives the same results as CMC in a faster way and slightly different method. In CMC, the main reason why the algorithm is so slow is the call to DBSCAN every new timestamp. CuTS tries to avoid this computational limitation by modifying the density-based clustering algorithm. Instead of considering points, it uses trajectories as samples. Trajectories are polylines, that is to say sequences of consecutive segments. [16] calls the modified version of the density-based algorithm TRAJ-DBSCAN. Of course, a distance has to be defined on trajectories.

Definition 9 (Trajectory distance)**.** Let us consider two trajectories t_1 and t_2. The distance between these two trajectories $D(t_1, t_2)$ is defined as the shortest euclidean distance between any two points belonging to t_1 and t_2.

CuTS proposes to apply TRAJ-DBSCAN at regular time intervals (as CMC runs DBSCAN every l seconds, with l the time interval length). The time domain has to be partitioned once again into time intervals of length l. At every new time interval τ, we extract all the consecutive segments whose time interval intersects τ for all trajectories in O. For each trajectory, we obtain a polyline which is sent as a sample to TRAJ-DBSCAN.

In CuTS algorithm, the new major time-limiting code segment is the algorithm TRAJ-DBSCAN. The less segments in any polyline there is, the faster the density-based algorithm will be executed. Before running the whole CuTS algorithm, trajectories are simplified with Douglas-Peucker method, a well-known algorithm introduced in 1972 presented in definition 10.

Definition 10 (Douglas-Peucker)**.** Given a polyline o (sequence of n points p_1, p_2,... , p_n) and a real δ, the Douglas-Peucker algorithm returns a new polyline o' built as follows. Let us consider p_i the farthest point from the segment $\overline{p_1p_n}$. If its distance to the segment $\overline{p_1p_n}$ is inferior to δ, this segment is reported as the simplified trajectory o', otherwise, we recursively apply this process to the two polylines $p_1, ..., p_i$ and $p_i, ..., p_n$. In the end, the concatenation of reported segment is returned as the final simplified trajectory.



Credit: *uploaded by Leupold,* `de.wikipedia.org`*, CC-BY-SA-2.0-DE*

Figure 5.1: Steps of Douglas-Peucker algorithm on 8-point polyline

An example of a trajectory simplification is given in figure 5.1. In this example, 3 points are discarded so that from an 8-point polyline the algorithm creates a simplified trajectory containing 5 points. Convoys found by CuTS are then processed with a classic CMC algorithm, a step called CuTS refinement in [16]. It permits to give exactly the same results as CMC and is quite fast since just a few candidate convoys will be given as input to the CMC call.

According to the results presented in the original article, CuTS algorithm is from 3 or 4 to 10 times faster than original CMC algorithm.

5.4 CMC on GPS tracks

5.4.1 Implementation & first results

Spatio-temporal clustering is a recent search area and few implementations are available. No implementation in Python has been found, so we implemented our own version of CMC using scikit-learn [11] implementation of DBSCAN. Low-level optimisations with Cython and complex spatial structures are used to make DBSCAN fast. For that reason, implementing CuTS would have been much more time consuming since low-level optimisations on TRAJ-DBSCAN would have been required to make this implementation worth it. This paper relies only on an implementation of CMC. A further work may improve this part by implementing CuTS.

The dataset sent to CMC has to be homogeneous. Journeys from different period timetables should not be mixed. We will see in the next chapter how to recognise period patterns. Results presented in this section are done with a dataset of journeys simulated with a unique timetable.

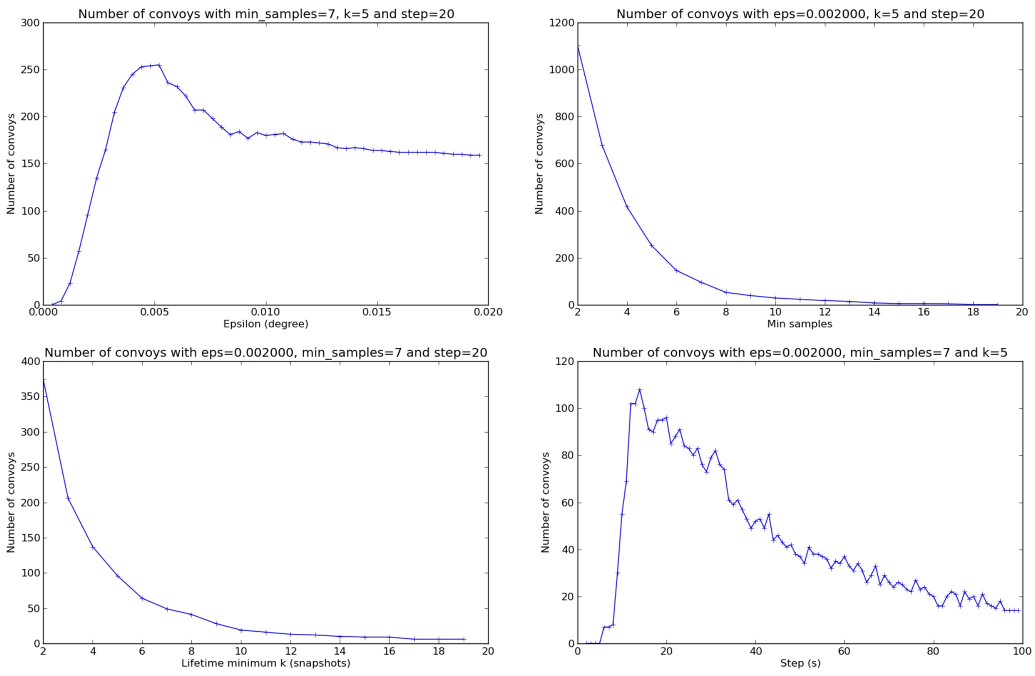


Figure 5.2: Evolution of number of convoys found

Many parameters are required when running CMC. To choose them properly, we draw the evolution of convoys found while varying a single parameter. Figure 5.2 shows the evolution of the number of convoys found on line 230. We finally chose the following values for our parameters since they mapped reality the best: $minSamples = 4$, $\epsilon = 0.0025$, $k = 5$ and $step = 16s$. Of course, depending on the number of journeys considered, the optimal parameters may vary. ϵ is measured in degrees, which explains this very low value.

5.4.2 Convoy fusion

When analysing results more deeply, we noticed that many convoys were found several times. Indeed, a convoy can be found at the beginning of a route, then lost because they are not close enough and later found again as a convoy. This behaviour explains the really high number of convoys found. With perfect results, we should find exactly n convoys if a given timetable has n rides per day in a period pattern.

Definition 11 (Same convoys)**.** Let us consider c_1 and c_2, two convoys reported by CMC and θ an integer. c_1 and c_2 are two instances of the same convoy with respect to θ if

$$|c_1.objects \cap c_2.objects| \geq \theta$$

After running a spatio-temporal clustering algorithm, we compare candidates as explained in definition 11. Every time two convoys satisfy the inequation, they are merged into a single one. After this step, we have a set $\mathcal{C}$ of convoys, in other words, groups of trajectories. Figure 5.3 shows the evolution of convoys found while varying parameters before and after convoy fusion. The line on which we tested had 45 rides per day. As we can see, while varying parameter *Step* length, the curve oscillates close to this expected value. By analysing more in details the results given by figure 5.3, we noticed that the same ride sometimes appeared twice when two convoys moved really closely in time. In the rest of this section, we present a distance to measure these "very close" rides to detect repetitions.

Each convoy found after this fusion step is supposed to represent a ride of the timetable. However, all trajectories have different departure and arrival stops: trajectories (journeys) are not homogeneous. Thanks to the previous part, a design of the line is available in both direction. Let us consider the stop sequence. The objective when filling the timetable is to find the schedule at which the bus (in the case of a bus line) arrives at each stop. To do so, we average all existing journey timestamps (in a given convoy) for each stop. If a stop does not have any journey crossing it, we let its schedule unknown in the created ride. After running this algorithm, for each convoy c_i in $\mathcal{C}$, a ride r_{c_i} is associated. Once again, two rides r_i and r_j can represent the same ride on the timetable. Indeed, two groups of trajectories can create two convoys at different sections of the route whereas they represent the same ride. We introduce a distance on rides to detect similar rides (definition 12.

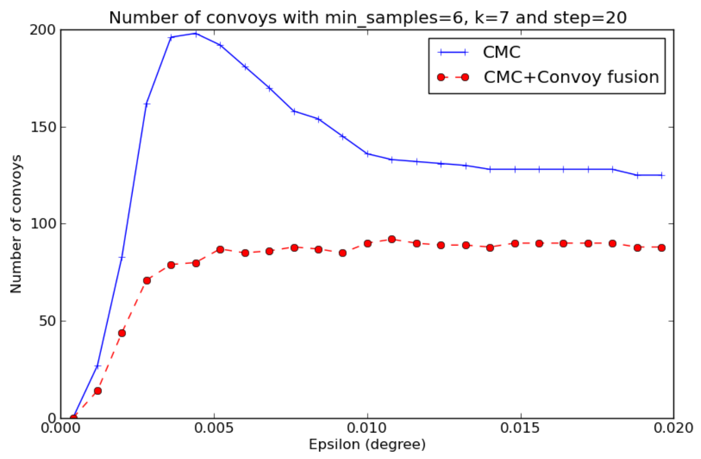
Definition 12 (Ride distance)**.** Let us consider two rides r_1 and r_2. Each ride is a sequence of stops with associated schedules. We note $r_1 = \langle (s_1, t_1), (s_2, t_2), ..., (s_n, t_n) \rangle$ the first ride (and similarly r_2 with primes). Time values may be null in case of unknown schedules. We define the ride distance D_{ride} as

$$D_{ride} = \frac{1}{p} \sum_{\substack{i=1 \\ t_i \neq null \\ t'_i \neq null}}^{n} (t'_i - t_i)^2$$

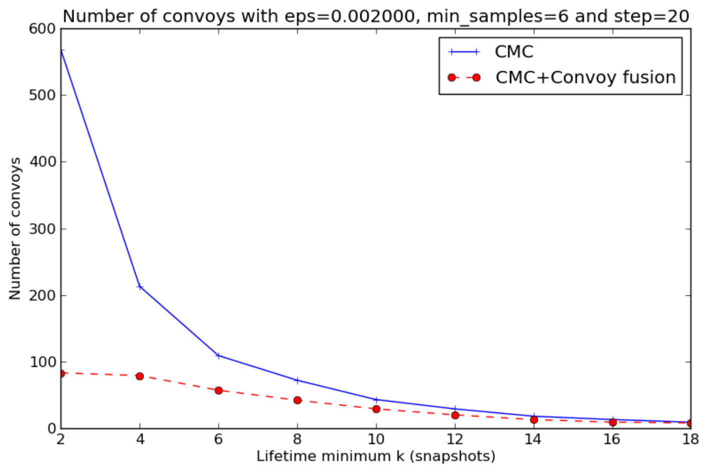
where p is the number of terms in the sum (that is to say, numbers of stops which have a non-null schedule in both rides). The normalisation is necessary to make this distance robust to any line, regardless the number of stops in the sequence.

We compute the distance between all rides which have similar times (e.g. around 1pm). Every time this distance is inferior to a chosen threshold, we merge them by averaging times (if one time is null and not the other, the non-null one is chosen in the merged ride).

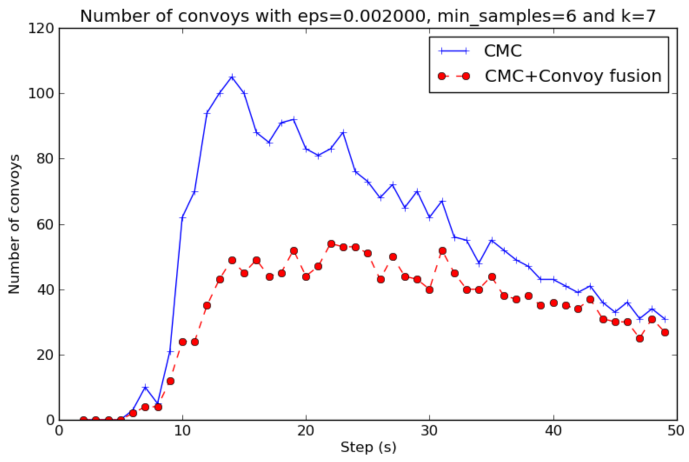
After this step, we are supposed to have found unique rides. With a perfect dataset, we are supposed to have exactly n rides if the targeted timetable has n rides.



(a) ϵ on x-axis



(b) Lifetime k on x-axis



(c) Interval length $Step$ on x-axis

Figure 5.3: CMC algorithm with different moving parameters

Chapter 6

Repeated pattern recognition

6.1 Motivations & objective

As explained in the previous chapter, we first applied spatio-temporal algorithms on homogeneous data. This means that our dataset contained only journeys done with the same timetable. However, this model is not realistic since transport lines have different periods: holidays, public holidays, normal days, etc.

In the current chapter, we aim at proposing a solution to extract repeated patterns, that is to say days (or weeks) which have the same timetable. Obviously, our clustering algorithms rely on an important number of journeys so, we generate this time at least 20 000 journeys over a year. As we will see later, the results we obtained in this chapter rely a lot on our model and on the behaviour of service users.

6.2 Distance on days

Each day d contains a number n_d of journeys. The distribution of n_d over a year may be a first indicator to classify days or weeks in periods. However, the first difficulty brought by journeys is their complete heterogeneity. Indeed, their parameters can vary a lot:

– time intervals of journeys (both times in the day and lengths)
– departure and arrival stops

Since days are only characterised by the journeys they contain, comparing them amounts to compare their journeys. However, days cannot be represented in an easy way with a metric or vector space. Indeed, they contain different numbers of journeys whose characteristics vary a lot. To tackle this major problem, we decide to represent days by the rides which can be found by a spatio-temporal clustering. This means that each day has a large amount of data, which is not very realistic. In section 6.3, we propose a method to avoid this highly restrictive constraint. In the rest of this section, we consider that each day d has a number $n_{r,d}$ of rides.

In the previous chapter, we defined a distance on rides to measure their similarity. Let us define an extension of the ride distance for days.

Definition 13 (Day distance)**.** Let us consider two days d_1 and d_2. Each day is a set of rides, that is to say a set of sequences of stops with associated schedules. For example, d_1 is defined by:

$$d_1 = \{r_{1,i} | i \in [\![1, n_{r,d_1}]\!]\}$$

We define the day distance D_{day} as follows:

$$D_{day}(d_1, d_2) = \max\left(\frac{1}{n_{r,d_1}} \sum_{i=1}^{n_{r,d_1}} \min_{r_2 \in d_2} D_{ride}(r_{1,i}, r_2), \frac{1}{n_{r,d_2}} \sum_{i=1}^{n_{r,d_2}} \min_{r_1 \in d_1} D_{ride}(r_{2,i}, r_1)\right)$$

The max is necessary because if a day has few rides (either because the spatio-temporal algorithm found few rides, either because d has indeed few rides), the distance may be wrongly small. For example, its rides may be strictly included in the other day ride set. Moreover, to make the distance symmetric, this max function is required.

To make it simple, D_{day} computes for each day the sum of every ride to its closest ride from the other day. The closest ride from the other day is supposed to be the common ride in the timetable in case of days from same periods.

We now have defined a distance over days. Each day can be compared to any other day. Let us cluster days in periods. A traditional clustering algorithm such as k-means cannot be applied since we do not work in a vector space. However, since the distance between any days can be computed with D_{day}, a graph with days as nodes and distances as weights can be built. We apply a community detection algorithm such as the walktrap algorithm to extract days with similar rides. Figure 6.1 shows the detection of two period patterns: one for the weekend and one for weekdays.

As shown in the dendrogram, with a modularity $m = 2$, two patterns are found by the algorithm.

6.3 Smart scanning

The previous method to detect patterns has one major drawback, it relies on a high number of journeys for each day. This model may not be satisfied in reality. We propose here an algorithm to reduce the quantity of journeys required.

To make the dataset bigger (in case of few journeys per day), we group days which are likely to have the same patterns. The first objective is to detect groups within a week: weekdays and weekend in most examples. To identify such a pattern, we select a sufficient number of weeks which are likely to be "normal weeks" and we remove from them public holidays. If we selected n weeks, we should have maximum n Mondays, Tuesdays, ... (and less in case of days removed). We then perform a

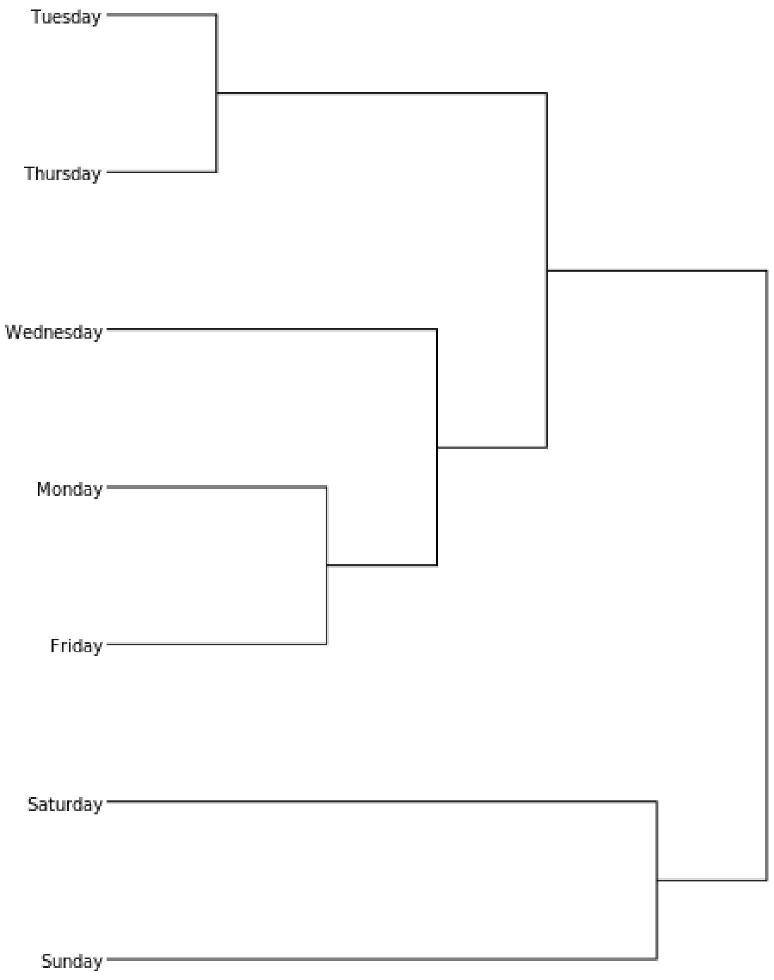


Figure 6.1: Period clustering – dendrogram

day recognition pattern as presented in the section above. We should then identify day patterns.

The following step is to cluster together weeks which have the same timetables. For every week, we group together days with the same pattern (found previously), and we apply the algorithm over weeks. A distance on weeks can be easily computed by summing the day distances over each day. The number of samples may be not sufficient for using each week separately. We create groups of two weeks (holidays are often superior to one week). Of course, this method has many limitations and may be not sufficient in many cases. Giving a timetable over a year for any line seems to be a too ambitious objective.

In the following chapter, we discuss the limitations of the method we presented in the thesis and give ideas to continue and improve this work.

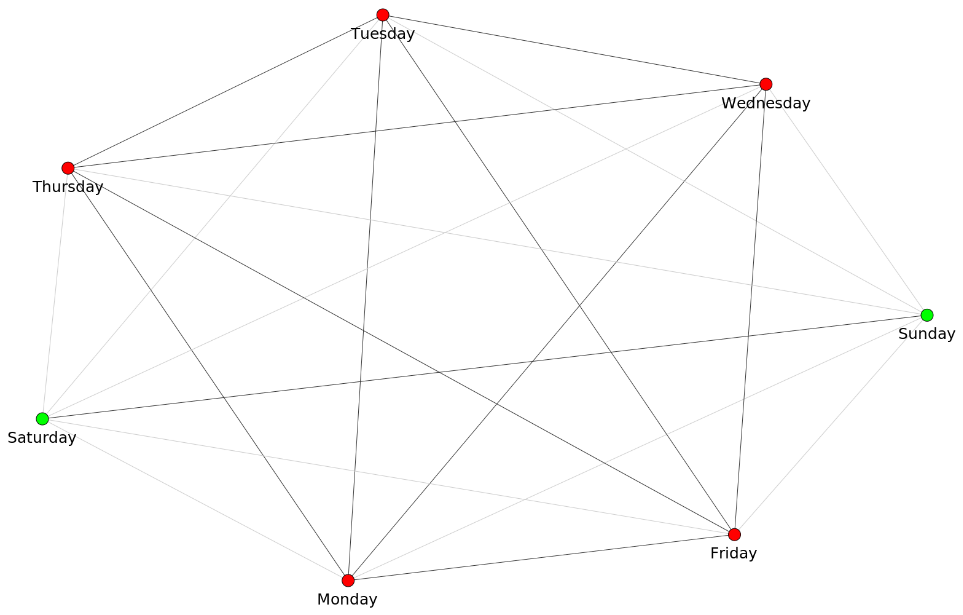


Figure 6.2: Period clustering – graph with communities

Chapter 7

Conclusion

Our work has been done with **simulated** data since no dataset was available. In this chapter, we first present conclusions about what has been done. In the second section, we propose guidelines to continue our work since the thesis mainly focused on giving a proof of concept.

7.1 Conclusions and limitations

In this paper, we presented a solution to extract information on public transports, using a set of journeys as unique source of data. We designed a system based on density-based clustering algorithms capable of finding most stop locations for a given line. The only situation where some stops lack appears if certain stops are rarely crossed (e.g. isolated stops, with a unique ride per day). Nevertheless, we can also imagine that since they are rarely crossed, the rides reaching them may be quite crowded, giving us much information. Anyway, only a collection of real data may reflect behaviours of service users. Furthermore, our system gives us good sequences of stops, even if in many cases, a few disconnections are introduced in the graph representing the structure of the line. However, we noticed that in most situations, it seemed to come from simulation issues. We expect the results of our method using a real dataset to remain close to the ones we got, and probably even improving certain parts such as final graph designs of lines.

In the second part of the thesis, we presented two spatio-temporal clustering algorithms which detect moving clusters (or convoys). This method allows us to extract interesting schedules but relies on homogeneous and numerous data. This situation seems difficult to achieve in the real life. Testing convoy detection algorithms with real data seems necessary to study the robustness of our method. Moreover, our simulation of time variations (traffic jams, rain) is probably really far from reality. It seems difficult to imagine how good timetables built with our method would be with a real dataset.

Obviously, the main limitation of our work comes from our dataset. Indeed, our simulated data cannot reproduce perfectly real data and even more so, human behaviour. Most of further work should consist on testing our method on data collected with a mobile application in a real situation. In addition, the system in itself relies on an important collection of journeys and so, by a significant amount of sample providers. This means that the success of our algorithms highly depends on the number of users who would use our mobile application to crowdsource the database which feeds our algorithms and their motivation.

Despite these limitations, the results we got lead us to think that continuing this project should be worth it and allow the construction of a global database giving at least line structures and stop locations. Building timetables may be more complicated and human intervention may remain necessary to solve specific situations.

7.2 Further work

Much work still has to be done to complete a system capable of extracting public transport data using tracking via a mobile application. As said many times before, the main weakness of the previous work is that it relies on simulated data. Two main developments have to be performed to continue this work:

- development of a mobile application and setting up of a beta testing on a few lines with a significant panel of users. This beta testing aims at collecting real data and at observing service user behaviour.
- testing of the current method on the beta testing dataset and highlighting of weaknesses. Improvements or modifications of the method described in the thesis may be performed, according to the results obtained with the dataset of real journeys.

We think that these two works should be performed at the same time if possible because limitations and issues brought by algorithms may require modifications in the information we ask via the mobile application.

In addition to these new developments, we think that many optimisations can be performed in our algorithms. We mentioned in this paper that we never focused on making our code as fast as possible and we chose to lead our research as far as possible.

Bibliography

[1] Urban Launchpad. Bringing the first map of Dhaka's bus network to life for its millions of daily riders. `http://www.kickstarter.com/projects/urbanlaunchpad/first-bus-map-of-dhaka`.

[2] M. Vieira, P. Bakalov, and V. Tsotras. On-line discovery of flock patterns in spatio-temporal data. *GIS '09: Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information System*, pages 286–295, 2009.

[3] Elias Frentzos. 276 trajectories of 50 trucks delivering concrete to several construction places around athens metropolitan area in greece for 33 distinct days. `http://www.chorochronos.org/?q=node/5`.

[4] Dennis Luxen & community. C++ implementation of a high-performance routing engine for shortest paths in road networks. `http://project-osrm.org/`.

[5] Joe H. Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58:236–244, 1963.

[6] E. W. Forgy. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *Biometrics*, 21:768–769, 1965.

[7] Stuart P. Lloyd. Least squares quantization in pcm. *Bell Telephone Laboritories Paper*, 1957.

[8] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining.* Addison-Wesley Longman Publishing Co., 2005.

[9] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. *SIGKDD*, pages 226–231, 1996.

[10] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 49–60, 1999.

[11] Scikit learn developers. Simple and efficient tools for data mining and data analysis built on numpy, scipy, and matplotlib. `http://scikit-learn.org/stable/`.

[12] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. *J. of Graph Alg. and App. bf*, 10:284–293, 2005.

[13] M. Girvan M. E. J. Newman. Finding and evaluating community structure in networks. *Physical Review E*, 69(2), 2003.

[14] Satu Elisa Schaeffer. Graph clustering. *Computer Science Review*, 1:27–64, 2007.

[15] S. Kisilevich, F. Mansmann, M. Nanni, and S. Rinzivillo. Spatio-temporal clustering: a survey. *Data Mining and Knowledge Discovery Handbook*, pages 855–874, 2010.

[16] H. Jeung, M.L. Yiu, C.S. Jensen, and H.T. Shen. Discovery of convoys in trajectory databases. *Proc VLDB Endow*, 1(1):1068–1080, 2008.

[17] P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatio-temporal data. *Advances in Spatial and Temporal Databases*, pages 364–381, 2005.