4

Data Preprocessing

It is common knowledge in data analytics that virtually all data sources come with issues and limitations that the wise analysts should consider before starting their work. Mobility data is no exception to this rule, and indeed there is an extensive set of operations that we can apply to improve the quality of the data and its usability. In this chapter we will introduce some of the simplest and most common operations aimed to this task, mainly focusing on GPS trajectories: identifying and/or correcting error points, removing redundant data, reconstructing portions of movement, inferring information from the context.

Finally, we will complete the discussion with notes on mobile phone data, which can benefit from most of the techniques mentioned above, and yet has its own specific issues and methods.

4.1 Filtering noisy points from trajectories

The most common issue in trajectory data is the presence of points with position errors. In some cases the error can be moderate, for instance due to a partial occlusion of GPS satellites in some areas of the city, resulting in deviations of a few dozen meters from reality that are hard for a human to identify at first sight if the trajectory is simply plotted; in other cases the position is completely wrong, due to malfunctioning or cold-start issues that generate basically random coordinates. In both cases we need algorithms able to detect the potentially incorrect points in the data, which are usually simply removed.

We can devise two general approaches to the problem:

- Context-based filtering assesses the reliability of each single point separately from the others, by looking at the geographical context around it, such as comparing against a map or similar background knowledge.

- Movement-based filtering, instead, considers the sequence of points (not single ones isolated) and identify unlikely movements.

We briefly discuss each of them in rest of this section through examples and reviewing some simple practical algorithms.

Context-based filtering. Let start with an example taken from a rather common public dataset, namely the S.F. taxi trajectories. Figure 4.1 below shows a sample of points plotted beside a map of the same area. Teal circles A, B and C highlight three specific points in the data.

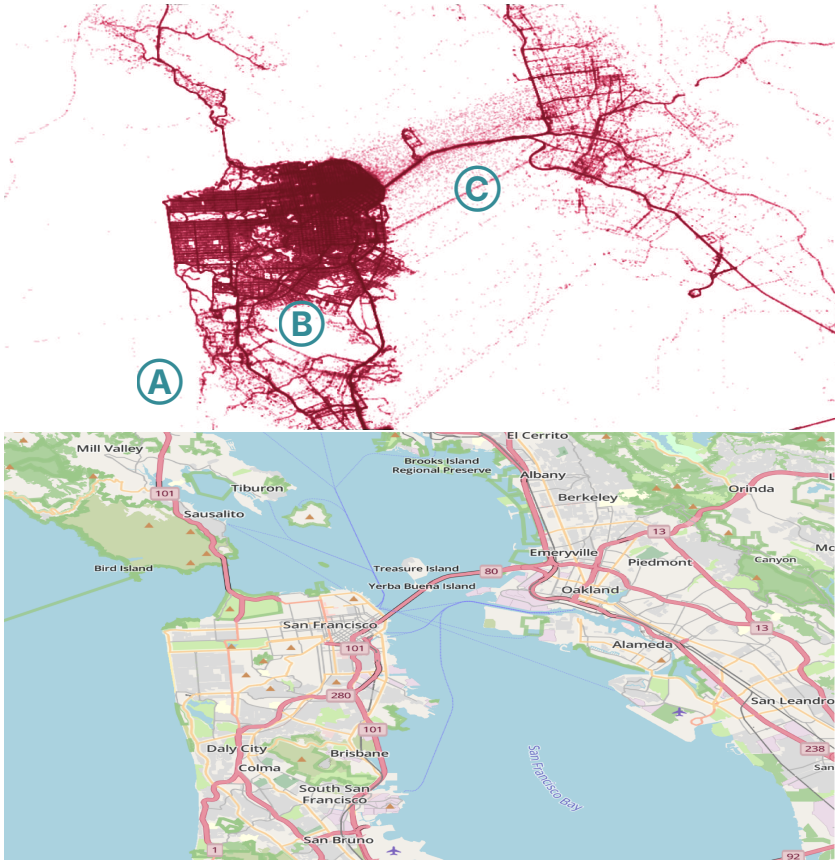


Figure 4.1 The San Francisco Bay. On the top, a sample of points from the S.F. taxi open dataset, plotted over a blank background. On the bottom, a map of the area (taken from OpenStreetMap, MapTiler OMT layer).

With a bit of knowledge of the city we can immediately appreciate that the

points distribution looks very reasonable, showing dense areas in the city center (NE coast of S.F. peninsula) and in residential areas (NW coast), and clearly highlighting the main roads. Notice that points are plotted against a completely blank background. A visual comparison with the map allows us to spot (at least) three types of points that could be errors to filter, exemplified by the three teal circles in the plot:

- Type A: these are isolated points located in the middle of water, which is physically impossible (for a working taxi). They clearly represent wrong locations and can be discarded;
- Type B: these points are located on the ground, yet they are isolated from the others and also far from any road. Considering that taxis cannot drive outside the road network, these points are clearly incorrect;
- Type C: these points are not isolated, and indeed the data plot might suggest that they are following a road crossing the bay. However, the map shows that there is no bridge there. While this might call for investigation about how is that possible (and indeed, the reader is invited to think about possible explanations), that represents clearly some exceptional behaviour that in most cases we do not want to consider and/or affect our analyses, thus we probably want to remove them as well.

The cases seen above suggest us a very simple approach to filter out problematic points by checking each of them against the set of *legit places*, in our example represented by the road network. Algorithm 4.1 synthesizes the process, also allowing a tolerance δ, representing how far a point can be from a road to accept it.

Algorithm 4.1: Context-based points filtering

Input: Trajectory $T = [p_1, p_2, ..., p_n]$, Road network R, Distance threshold δ
Output: Filtered trajectory T'

```
1 T' ← [ ];
2 foreach point p in T do
3     d ← min_distance(p, R);   // Dist. from closest road
4     if d ≤ δ then
5         Append p to T';
6 return T';
```

The key operation of the algorithm is `min_distance`(p,R), which finds the road segment closer to the point p and computes their distance. Notice that this operation can be very expensive if implemented in a naive way, such as

scanning all the segments for each point. On large road networks efficient data representations and algorithms are recommended.

Can we apply the approach above in different contexts, like ships navigating the sea or pedestrians free-roaming or hiking in a park? The answer is 'Yes', of course. Yet, the legit places need to be represented by an appropriate geographical entity that replaces the road network R of our example. For instance, legit places for boats might be all waterways, usually represented as sets of polygons; and legit places for hikers might be the area accessible to visitors, again represented as polygons. Consequently, the `min_distance`(p,.) function needs to be adapted, e.g. computing points-to-polygon distances.

Movement-based filtering. When the legit places for points are very dense, for instance in city centers crowded with streets, it can happen that noisy points still fall on apparently correct places, thus making the context-based filtering not a definitive solution to the problem. A different angle from which to face it consists in ensuring that the sequence of points of a trajectory, now seen as a whole, respects physical movement constraints, for instance by checking that the estimated speed of the moving object never exceeds some reasonable thresholds.

Figure 4.2 provides an example that illustrates the idea. The sequence of (teal) points that form the trajectory all lie on or close to the road network. Yet, we can recognize the presence of some strange jumps at points 4 and 8, most likely due to a location error.

There are several ways to formalize our intuition and make it an algorithm for automatic detection of potential noisy points, the simplest and most convenient being to simply check the average speed between two consecutive points. Indeed, a side effect of a wrong point location is that the distance from the previous point in the trajectory is most likely disproportionate to the elapsed time. For instance, assuming that a constant sampling rate is adopted, the distance between point 4 and point 3 in Figure 4.2 is almost six times the distance between any other pair of consecutive points (excepted point 8, another candidate), which translates into a much higher average speed. Notice that we are currently assuming that objects can travel along a straight line, which can significantly lower the speed estimates, thus our speed checks in some cases are still rather conservative.

These considerations suggest us a simple and efficient filtering strategy, summarized in Algorithm 4.2.

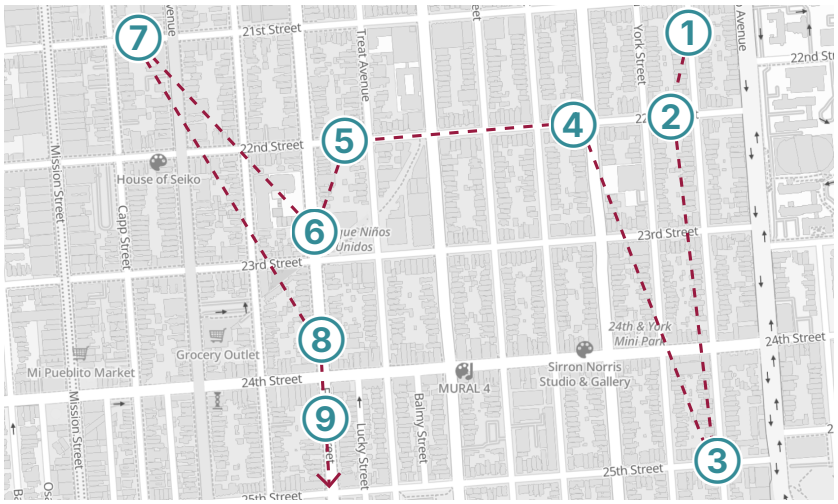


Figure 4.2 Sample trajectory with some points affected by large errors. Notice that noisy points are not recognizable through context-based methods. (Background map taken from OpenStreetMap, MapTiler OMT layer)

Algorithm 4.2: Speed-based points filtering

Input: Trajectory $T = [p_1, p_2, ..., p_n]$, Maximum speed σ
Output: Filtered trajectory T'

1 $T' \leftarrow [p_1]$;
2 $p_{last} \leftarrow p_1$;
3 **for** $i = 2, \ldots, n$ **do**
4 $\quad d \leftarrow$ `distance`(p_{last}, p_i) ; `// Euclidean or Haversine`
5 $\quad t \leftarrow$ `time_diff`(p_{last}, p_i);
6 $\quad$ **if** $d/t \leq \sigma$ **then** `// Check avg. speed`
7 $\quad\quad$ Append p_i to T';
8 $\quad\quad p_{last} \leftarrow p_i$;
9 **return** T';

To start with, the algorithm assumes the first point p_1 is correct. Then, each other point is scanned in chronological order and compared to the last *correct* point p_{last}, checking that the estimate average speed between them is lower than a user-provided threshold σ. Notice that points labeled as noise (thus violating the condition checked in line 6 on the algorithm) are not only discarded from the output, but also not used for checking the next point in the trajectory. That means, for instance, that when point 4 is checked against point 3 and discovered

to need an excessive average speed, at the next iteration of the the cycle point 5 will be compared against point 3 instead of point 4, since the latter has been skipped. Distance between 5 and 3 is clearly larger than the average in the trajectory, yet so is also the time interval between them, making the average speed normal.

Before leaving this section, it is useful to point out two limitations of this approach that the reader should remember. First, what can happen if the initial point p_1 of the trajectory is noisy, e.g. what if our sample trajectory in Figure 4.2 actually started from point 4? The short answer is: almost all the trajectory disappears. Indeed, p_1 = 'pont 4' will be used as reference (p_{last}) throughout the execution, and every other point appears too far from it to be correct (excepted maybe points 9 and 10). The output in most cases like this will be just $T' = [p_1]$. If that is a significant possibility[1], the analyst should adopt some countermeasure, such as removing p_1 when too many initial points are filtered out and re-run the algorithm. Second limitation: speed-based filtering can be not very effective if the errors are moderate – yet large enough to be annoying. For instance, if points 5 and 6 were switched, the estimate speeds would be still moderate, and yet the resulting movement looks rather incoherent. More sophisticated approaches able to capture these aspects should be developed, if considered important for the analysis, for instance by constraining the direction changes (basically avoiding U-turns and similar).

4.2 Matching points and paths to a map

In the section about context-filtering we saw that road networks and similar context knowledge can help improving the quality of data by enforcing some commonsense constraints, such as restricting points to be not too far from roads. Here we go beyond filtering, and focus on *aligning* the input data to such constraints, with the ultimate objective of correcting the location noise and reconstructing portions of movements that were not covered in the data. As we will see, these two tasks are tightly connected.

4.2.1 Point map matching

If a location point of some vehicle lies outside of a road, it is clear that our input data are affected by some source of noise, and we expect that the real position

[1] Rather than being just a matter of bad luck, the event outlined above could be very likely in some scenarios. For instance, most GPS devices suffer from cold start errors, i.e. the location estimates in the first instants after switching on are rather noisy, yet in some systems they are recorded as good ones without any labeling.

of the point should be somewhere on a road segment. But, which segment, and where exactly on the segment should the point be moved to fix the error?

Closest segment mapping. The simplest answer is that the best candidate location is the closest point on the road network. That means that for each point p_i of a trajectory we should consider the portion of space R covered by the roads and identify the single point $p' \in R$ that minimizes the distance from p_i.

We can summarize the idea with a single formula:

$$match(p_i) = \arg\min_{p' \in R} dist(p', p_i)$$

How to compute it. Although from an abstract geometric viewpoint R contains infinite points, the computation can be achieved quite easily with a limited number of basic operations. Indeed, we recall that the road network R is just the union of its segments $r_1, \ldots, r_N$, and we can break down the minimization as $match(p_i) = \arg\min_{p' \in C} dist(p', p_i)$, where C contains for each segment its closest point to p_i, namely $C = \{\arg\min_{p \in r_j} dist(p, p_i) | j = 1, \ldots, N\}$. Since single segments r_j are typically represented as (or can be broken down to) a simple straight line, we can use high-school geometry formulas to directly compute the projection of p_i on segment r_j, which will be our candidate – unless it falls outside the segment, in which case we will take instead one of its endpoints (the closest one to p_i, clearly).

An example. Figure 4.3 depicts the general idea of point mapping, together with a sample trajectory mapped to the road network. We can see how points 1 and 3 are mapped to internal points of a segment, whereas point 2 is mapped to a "corner", namely the endpoint joining two segments.

Figure 4.3 actually shows also an example of what could go wrong with a simple approach like this. Indeed, if the three points depicted are parts of the same trajectory we would expect them to be mapped on roads that form a reasonable path on the network. Here, instead, point 3 is assigned to a road that has no connection with the others, and thus it is quite unlikely that it is visited right after the first two. Introducing this kind of reasoning, yet, requires to know how to infer a reasonable path along a road network starting from just a few points. We will briefly recall a few elements of path optimization, and later on come back to the original problem.

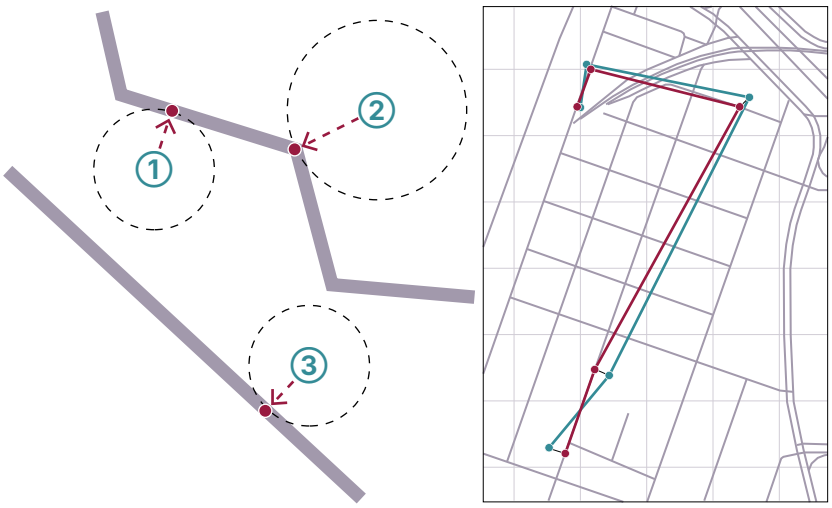


Figure 4.3 Point mapping schema (left) and example (right) with noisy points (teal) and their map matching (red). (Original figures)

4.2.2 Path optimization

Trajectories describe a path connecting two points, usually corresponding to what the individual believes to be the best way to reach the destination. For instance, if there exists a straight main road that we can follow to reach our destination we most likely prefer it to a longer path that performs several detours and takes much more time. Path optimization means to identify the path over the road network that optimizes a given preference criterion. We introduce the problem in a formal way, making use of the graph representation of road networks and encapsulating the path preferences into a cost function.

Formal definition. Let consider a graph representation G of the road network, where $G =< N, E, cost >$. N is the set of nodes, corresponding to road intersections, E is the set of edges, each representing the road segment that connects two intersections, and $cost$ is a function associating a cost to each edge/segment. For instance, $cost(e)$ could be computed as the length of the road segment e. Then, the *shortest path problem* consists in finding the minimum-cost sequence of connected segments that leads from a given origin node o to a given destination node d, namely:

$$SP(o,d) = \arg\min_{p \in \mathcal{P}_{o,d}} \sum_{e \in p} cost(e)$$

where $\mathcal{P}_{o,d}$ is the set of all possible finite paths over G that start from node o and end in node d: $\mathcal{P}_{o,d} = \{< (p_0, p_1), (p_1, p_2) \ldots, (p_{n-1}, p_n) > \mid \forall_i \ (p_i, p_{i+1}) \in E \wedge o = p_0 \wedge d = p_n\}$.

Examples. Figure 4.4 illustrates an instance of the problem where the origin and destination are the highlighted nodes, resp. in yellow and white.

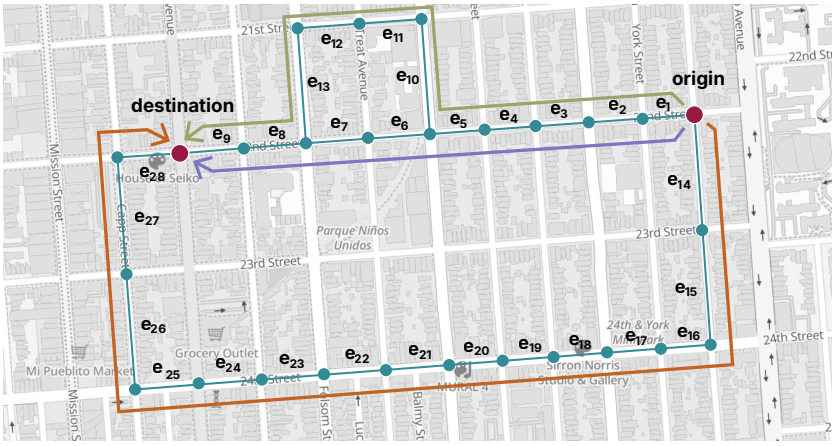


Figure 4.4 Three sample paths connecting the same pair of origin and destination nodes. (Background map taken from OpenStreetMap, MapTiler OMT layer)

The figure shows three of the several possible ones:

- The violet solution follows a straight path from the origin to the destination, crossing just 9 road segments: $p_{violet} =< e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9 >$
- The green solution is similar to the previous one, yet it makes a deviation after road e_5, basically skipping segments e_6 and e_7. This introduces two additional segments to the count: $p_{green} =< e_1, e_2, e_3, e_4, e_5, e_{10}, e_{11}, e_{12}, e_{13}, e_8, e_9 >$
- The orange path goes in a completely different direction, making an apparently much longer tour to reach the destination, traversing 15 segments in total: $p_{orange} =< e_{14}, e_{15}, e_{16}, e_{17}, e_{18}, e_{19}, e_{20}, e_{21}, e_{22}, e_{23}, e_{24}, e_{25}, e_{26}, e_{27}, e_{28} >$

Which one is the optimal path? To answer this question we need to specify what is the cost function we want to use. Let see three examples:

- Trip length: as the name of the problem (shortest path) suggests, this is the most intuitive criterion for optimization. Here, $cost(e)$ is the length of the road segment, and the overall $SP(o, d)$ will be the path with minimum total length. In our example we can easily see that the violet path will be the winning option, as a straight line is the shortest path you can follow to reach a destination.

- Travel time: this is the most common criterion that travelers try to optimize in real file, especially on daily routine trips. Here, $cost(e)$ should express the time needed to traverse the segment e. This information is much harder to obtain since the speed on roads can be variable and depend on several factors, yet we might obtain a first reasonable approximation by taking the speed limit of the segment. This allows us to compute the traversal time as $cost(e) = length(e)/max\_speed(e)$. In our example, assuming that segments on the South (e_{16} to e_{25}) have a much higher speed limit than the Northern ones (e_1 to e_{13}), the orange path could be the best solution, since the time saved on the East-to-West portion of trip overbalances the added time caused by the deviation.

- Number of segments/intersections: while not very intuitive, we can see this as an alternative measure that quantifies the complexity of the trip, since traversing intersections is typically more distressing than driving along a simple road. In this case, the violet path would be again the winner. A similar concept could involve the number of turns (in our example: 0 for the violet path, 4 for the green one, and 3 for the orange) or the number of traffic lights crossed (however, this information is typically much more difficult to obtain).

Computing optimal paths. Once defined the cost function $cost()$, computing the shortest path over graph G becomes a very standard and deeply studied problem. For those who are unfamiliar with the topic, here we briefly discuss the best known and most traditional solution[2] which brings the name of its inventor, *Dijkstra's algorithm*. Algorithm 4.3 provides the pseudocode of the method, which is rather simple and compact. It comprises two phases: the first one, from line 1 to 12, is the core where the path is actually discovered; the second one, from line 13 on, just reconstructs the path found to return it.

[2] *As old as computer science* would be a more appropriate expression: indeed, the algorithm was published in 1956, around the same period the term 'computer science' was coined.

Algorithm 4.3: Dijkstra's Algorithm

Input: Graph $G = (V, E)$, origin node o, destination node d
Output: Shortest distance $dist[d]$ from o to d, shortest path P

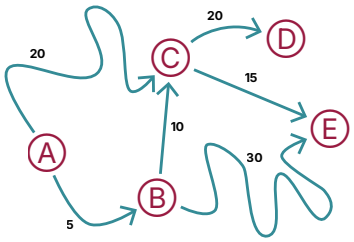
```
1  foreach v ∈ V do
2      dist[v] ← ∞;
3      prev[v] ← undefined;
4  dist[o] ← 0;
5  Q ← V;
6  while Q is not empty do
7      u ← extract node in Q with minimum dist[u];
8      if u = d then break // Stop when d is reached
9      foreach neighbor v of u in Q do
10         if dist[u] + cost(u, v) < dist[v] then
11             dist[v] ← dist[u] + cost(u, v);
12             prev[v] ← u;
   // Reconstruct shortest path from o to d
13 P ← [d];
14 u ← prev[d];
15 while u is defined do
16     prepend u to P;
17     u ← prev[u];
18 return dist[d], P;
```

For the sake of simplicity, let assume that our $cost(.)$ function represents the travel time. The algorithm starts its exploration from the origin point o and builds and updates step-by-step two arrays that describe for each node v reached so far at what time it was reached through the best path ($dist[v]$) and which is the previous node traversed to reach it ($prev[v]$). At the beginning (lines 1–3) all travel times are set to infinity and there is no previous node ($prev[v]$ == undefined). Only o is updated (line 4) with travel time 0, since the path starts from it and thus it is reached instantaneously. The queue Q represents the set of nodes yet to process, that at the beginning consists of all nodes in the graph. Now, the main cycle in lines 6–12 will actually visit all the nodes in chronological order, i.e. each time we move to the unvisited node u which can be reached sooner than anyone else through its best path (line 7). At the very first iteration, such node will be o, since all the others have infinite arrival time. Then, the arrival times of all u's neighbors v are updated: this is done by computing the arrival time of v passing through u and checking if this option

is better than the best one found so far (line 10), which is contained in $dist[v]$. If that is the case, this becomes the new best path for v (lines 11–12). At the first iteration, this means that all nodes directly reachable from o with one step are updated, and their cost will be exactly the traversal time of the edge (o, v). At the second iteration, the algorithm will *pop* from the queue Q the closest node to o, since we are sure that no longer path will be able to reach it with less time. The same steps are repeated as above, and at each iteration we remove from Q one node, till Q is empty or the current best node is our destination d, in which case there is no need to go on with the exploration. Notice that neighbors checked for the first time at line 10 will be for sure updated, since their initial arrival time is infinite – namely, this is the first path that reaches it, though not necessarily the best one overall. If instead the neighbor v was already checked previously (and thus it has a finite arrival time), updating or not depends on whether the new path is better than the previous ones.

The second phase of the algorithm is very simple: since $prev[v]$ contains the previous node traversed to reach v along its fastest path, we can walk back from d (line 13) to o by jumping from d to $prev[d]$, from $prev[d]$ to $prev[prev[d]]$ and so on (lines 14 and 17). At each step we add the currently visited node to an array, in reverse order. Notice that origin o has no previous node defined, since we started from it.

Example. The graph in Figure 4.5 on the left is an instance of shortest path problem, where the origin is node A and the destination is node E. Costs (e.g. traversal time) of road segments are reported on the corresponding edges. The table on the right of the figure shows the sequence of operations performed by the algorithm.



	A	B	C	D	E
Init	0	∞	∞	∞	∞
step 1	**0**	5	20	∞	∞
step 2	**0**	**5**	15	∞	35
step 3	**0**	**5**	**15**	35	30
step 4	**0**	**5**	**15**	35	**30**

Figure 4.5 Sample shortest path problem (left) and resolution with Dijkstra's algorithm (right). The table reports the cost $dist[v]$ of each node v for each iteration of the algorithm. Squares point to the nodes that are visited at each step, while underlined numbers represent updates of $dist$ values.

It is quite clear that the optimal path would be $A \rightarrow B \rightarrow C \rightarrow E$. Indeed,

while alternatives with less edges are available, e.g. $A \to C \to E$, the sum of their costs is larger – in our example, 20 + 15 = 35, which is larger than the optimal 5 + 10 + 15 = 30. After initialization (Init step), Dijkstra's algorithm starts by visiting node A, which had cost zero, and thus updates the nodes directly reachable from it, namely B and C (step 1). Now B has become the unvisited node with the smallest cost (5), thus it is visited at step 2. The cost of its neighbor C is updated, since passing through B is cheaper than arriving directly from A. Similarly, also the cost of E is updated. At step 3 the best unvisited node is C (with cost 15), which leads to update both D (never checked before) and E, which lowers the cost from 35 (the cost of the old path $A \to B \to E$) to 30 (corresponding to $A \to B \to C \to E$). The next node to be visited is then E, where the algorithm stops, since it is our destination.

Final notes on Dijkstra's algorithm: first, if we do not stop when we reach the destination node (i.e. remove line 8 of the code), the algorithm will compute the optimal costs and paths for all the nodes of the graph, which can be useful in some cases, yet also over-killing if we only need the path for a single destination. Second, the efficiency of the algorithm greatly depends on how the queue Q is implemented. A trivial implementation (as in Dijkstra's original formulation) makes its cost quadratic in the number of nodes, while efficient ones make it quasi-linear – a big difference when dealing with large road networks! Last note, Dijkstra's algorithm works only when all edges have non-negative costs, i.e. $\forall e \in E.cost(e) \geq 0$. If that does not hold, different solutions are needed, e.g. the Bellman–Ford algorithm, which is computationally more expensive.

CURIOSITY 4.1

A biographical *intervallo*: Edsger W. Dijkstra (1930–2002)

Dijkstra was a Dutch computer scientist, programmer, software engineer, systems scientist, and science essayist, recipient of the 1972 Turing Award for "fundamental contributions to developing programming languages".
He is famous for the shortest-path algorithm that brings his name, but also his contributions to "self-stabilization of program computation" (recipient of the "ACM PODC Influential Paper Award", later renamed "Dijkstra Prize").
He is author of hundreds of papers on computational and science philosophy issues.
Curiosities: Dijkstra was also well known for some personal peculiarities in his professional activity, like his habit of writing everything with paper and fountain pen, his huge productivity which led him to write hundreds of papers – many unpublished, now reachable at *E. W. Dijkstra Archive*; he seldom provided references in papers, with statements like "For the absence of a bibliography I offer neither explanation nor apology"; also, he supported the idea that counting should start from 0, not 1.
Finally, he was a remarkable teacher: fond of using chalk and blackboard also in recent times, with no projectors, he used not to adopt textbooks, and often his classes were mostly improvised and featuring long pauses from time to time. His exams were quite known and feared: each student was examined in Dijkstra's office or home, and an exam lasted several hours.

4.2.3 Point mapping with route reconstruction

As already mentioned, in some cases simply mapping a GPS point to the closest road segment could be the wrong way to improve its location accuracy, and it might instead be useful to consider the effect that the point mapping has on

the overall trajectory. The example shown in Figure 4.6 consists of a trajectory with three points P_1, P_2, P_3 and a road network that we want to use for map matching. The simplest closest-point association would be $P_1 \rightarrow C_2$, $P_2 \rightarrow C_4$ and $P_3 \rightarrow C_7$, yet it seems unlikely that a user would go from C_2 to C_4 and then go in the opposite direction to reach C_7.

Figure 4.6 Sample raw trajectory with three points (P_1, P_2, P_3) and the corresponding candidate projections on neighboring segments ($C_1, \ldots, C_7$). (Background map taken from OpenStreetMap, MapTiler OMT layer)

Probabilistic matching: Newson-Crumm. Various methods exist to integrate this trajectory-level vision in the point matching process, and many of them implement the idea defining a probability or likelihood of a sequence of road segments as compared to the original points. Most of them start from a common preliminary step: assigning to each point a set of reasonable alternative candidate road segments – and the corresponding projections of the point on each of them. The typical approach to collect candidates consists in a *range-based* selection, namely a distance threshold δ is fixed, and all segments e at distance not larger than δ from input point p are taken: $C_{range}(p) = \{e \in E | d(p, e) \leq \delta\}$. Notice that the size of $C(p)$ is variable, and it can also be empty if all roads are far from p. Alternatively, the user fixes the number n of candidates to associate to point p, and the n-nearest neighbors of p are selected, namely: $|C_{kNN}(p)| = n$ and $\forall e_{in} \in C_{kNN}(p), e_{out} \in E \setminus C_{kNN}(p) : d(p, e_{in}) \leq d(p, e_{out})$. The example in Figure 4.6 follows a range-based approach, assigning only close candidates

to each point. In particular: $C(p_1) = \{C_1, C_2\}$, $C(p_2) = \{C_3, C_4, C_5\}$ and $C(p_3) = \{C_6, C_7\}$.

As a representative of probabilistic matching approaches, we describe here the Newson-Crumm method, a simple solution based on Hidden Markov Models (HMM). The process involves the estimation of two types of probabilities:

- $p(p_i \mapsto C_k)$, namely the probability that C_k is a good candidate mapping for point p_i, assuming that $C_k \in C(p_i)$;
- $p(C_k \rightarrow C_l)$, namely the probability that a trip passing through C_k could later visit also C_l.

Newson-Crumm computes these probabilities through simple heuristics. $p(p_i \mapsto C_k)$ is function of the distance between the two points, the farther they are, the smaller the probability. In particular, it adopts a Gaussian decay function, which reflects well the way GPS errors are distributed:

$$p(p_i \mapsto C_k) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2} d(p_1, C_k)^2},$$

where σ is a parameter. Instead, $p(C_k \rightarrow C_l)$ is based on how much the shortest path connecting the two candidates over the road network deviates from a straight line, the logic being that straight paths are more natural, and therefore the sequence $C_k \rightarrow C_l$ should be more likely. In particular, the method computes $P(C_k \rightarrow C_l) = \frac{1}{\beta} e^{-\Delta/\beta}$, where $\Delta = \mathit{shortest\_path\_dist}(C_k, C_l) - d(C_k, C_l)$ is the deviation, and β is a parameter. Finally, these probabilities are used to find the point mappings $p_i \mapsto C_i$ for all p_i that maximizes the following product of probabilities: $\Pi_i p(p_i \mapsto C_i) p(C_i \rightarrow C_{i+1})$.

As an example taken from Figure 4.6, the mapping $p_1 \mapsto C_2$ will likely be preferred to $p_1 \mapsto C_1$ due to the distance of C_1; however, $p_2 \mapsto C_4$ probably will not be chosen even if it is the closest candidate (thus yielding the highest $p(p_1 \mapsto C_k)$) because the shortest path between C_2 and C_4 deviates a bit from the straight line, while C_5 would have a perfect $\Delta = 0$ and thus $p(C_2 \rightarrow C_k) = 1$.

Shortest path-based: Zhu-Honda-Gonder. We can approach the matching problem exploiting shortest path also from the opposite direction: instead of trying to map points and then see how good the overall mapped path looks, we try to compute a promising shortest path and then see how well it matches the single points.

This is indeed how the Zhu-Honda-Gonder method (among others) works: (i) first, the origin and destination points are matched with corresponding road segments, for instance by simply selecting the closest one; then, (ii) the shortest path between the origin and destination segments is computed; after that, (iii) a

fitness score is computed to check how well the remaining original points match the segments of the shortest path; if the score satisfies a given threshold, (iv) the shortest path is considered acceptable, the original points are mapped to their corresponding segments along it and the algorithm stops; otherwise, (v) we split the original trajectory into two or more parts, recursively run the algorithm separately on each of them, and then join their outputs (the points matches).

Here, we need to better define a few details. The fitness score of a trajectory vs. a path on the network is computed as the sum of the fitness of each trajectory point vs. its best matching segment along the path[3]. The fitness of a single point is a simple linear function of its distance from the segment, with value 1 when the distance is zero, and zero when distance is larger than a spatial threshold ϵ. Similarly, another spatial threshold is used to identify candidate cutting points in the trajectory: each subsequence of points that exceed such thresholds is separated from the rest of the trajectory, forming the splits we want. Additionally, the point with the worst fitting score is also used as split point. Final note: step (i) of the procedure requires to choose a segment for start and end points. In general, we can consider several candidate segments for each of them, let say sets C_{origin} and C_{dest}, as in the example of Figure 4.6, compute the shortest path for each pair $(o, d) \in C_{origin} \times C_{dest}$ and select the one that maximizes the fitting score.

A sample application of the algorithm is shown in Figure 4.7. Here, the shortest path $path_0$ computed between the (mapping of) points A and F fits only part of the trajectory, and the portion between B and D needs splitting. In addition, point D is particularly far off the shortest path, thus the original trajectory is split into four portions: $A-B$, $B-C-D$, $D-E$ and $E-F$. Then, the algorithm is run on each of them, resulting in paths $path_1$, $path_2$, $path_3$ and $path_4$. Since point C deviates too much from $path_2$, the portion $B-C-D$ is further split into two parts, and the algorithm is run again on them.

4.3 Trajectory compression

Though slightly counter-intuitive, it can happen to have too much data for the analysis. On one hand, sometimes the data is rather redundant or much more detailed than what our objectives require. On the other hand, high-detail data require higher computation times (or more computational power), potentially creating a bottleneck. In the realm of movement data that means our trajectories

[3] The actual implementation of this concept in Zhu-Honda-Gonder is slightly more involved, though not too complex, making use of an optimization schema. We invite interested readers to consult the original paper of the authors.

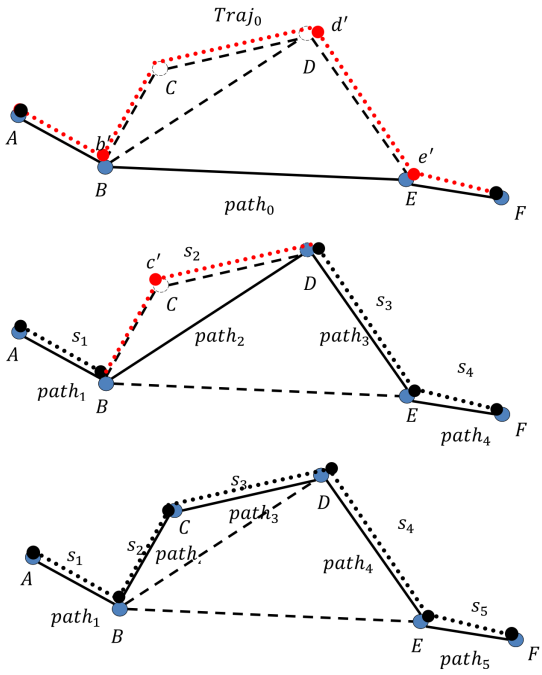


Figure 4.7 Sample point mapping according to Zhu-Honda-Gonder method. (Picture from the authors' paper. Copyright to check.)

contain more points – namely, they are denser – than needed, and thus we might want to simplify them by removing some points.

Simplifying the topic a bit, we will consider two kinds of data redundancy, illustrated in Figure 4.8. The first type consists of a sequence of points in the trajectory that make no significant movement; the second type moves significantly, yet in a straightforward way (a straight line). In both cases, we aim to identify the trajectory subsequence where the redundancy occurs, and remove as many points as possible without affecting the overall trajectory shape, as in the figure.

We introduce two well-known trajectory simplification/compression methods, each aimed to treat one of the situations just illustrated. Both of them are based on an input spatial threshold ϵ, representing an error tolerance.

Driemel–HarPeled–Wenk (a.k.a. Driemel's algorithm). This method is very simple and efficient, requiring a single scan of the data. Algorithm 4.4 summarizes it. Starting from the first point, the input trajectory is scanned sequentially,

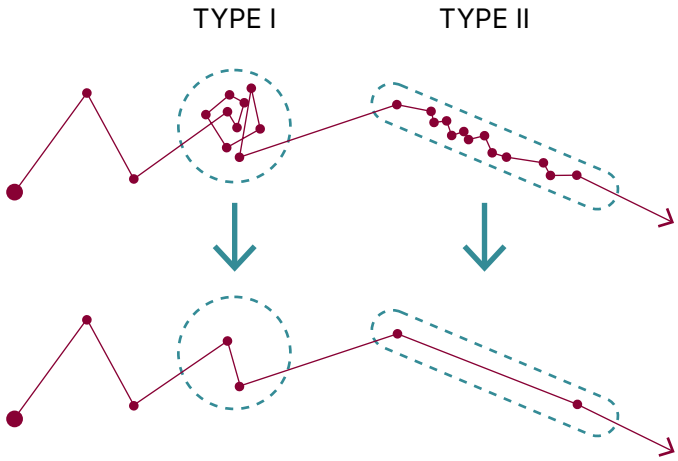


Figure 4.8 Two types of redundancy in trajectory data: (left) negligible movement and (right) straight-line movement. Below, an example of simplification of the trajectory removing redundant points.

and each point is preserved and added to the output trajectory T' only if it is significantly distant ($> \epsilon$) from the previous point in T'. Rephrasing it, whenever a point is added to the output we virtually draw a circle of radius ϵ and remove all following points till one of them exits the circle. Those within the circle are basically considered similar enough to the last preserved point to be disposable. Finally, the algorithm ensures that the last input point is preserved.

Algorithm 4.4: Driemel–HarPeled–Wenk's Algorithm

Input: Trajectory $T =< p_1, \ldots, p_n >$, spatial threshold ϵ
Output: Simplified trajectory T'

```
1 T' ←< p_1 >;
2 i ← 1;
3 while i < n do
4     q ← p_i;
5     i ← smallest index j ∈ [i + 1, ..., n] such that d(q, p_j) > ε;
6     if i is undefined then i ← n // We reached end of T
7     Append p_i to T';
8 return T';
```

Driemel's algorithm directly addresses redundancies of the first type shown in our previous figure, yet it also provides a simple way to reduce the trajectory sampling rate in general. Indeed, if we imagine a very dense trajectory T

following a straight line, Driemel's method will keep only a small subset of points that form a sequence of *jumps* of length slightly larger than ϵ. In this ideal case, only a fraction $\sim length(T)/(|T|\epsilon)$ of points would be retained.

Ramer–Douglas–Peucker (a.k.a. Douglas–Peucker or DP).

This algorithm aims to approximate a subsequence of points with a straight line segment, provided that all points are close enough (distance $\leq \epsilon$) to the segment, thus directly tackling redundacies of the second type shown in Figure 4.8. Instead of scanning the input trajectory T points-by-point, the DP method starts trying a "lucky shot", namely checking if the segment connecting the first point of T with the last one is already capable of approximating the whole trajectory. That is realized by taking the farthest point p_i from the segment (line 1) and comparing its distance with ϵ. If that works, the algorithm can already stop, achieving the maximum possible compression – just the two endpoints are enough to represent the content of T. If that fails, we split the problem into smaller ones and call the algorithm recursively twice, once on the subtrajectory that precedes the farthest point p_i, and once on the remaining part. Notice that this implicitly means that p_i will be preserved, together with the first and the last point of T.

To the attentive reader, the DP algorithm will look rather familiar. Indeed, the general idea and the recursive approach are very similar to the Zhu-Honda-Gonder method for point matching discussed in this chapter, the main differences being the usage of straight line approximations instead of shortest paths on the road network, with obvious consequences on efficiency. Also, the final objective is rather different.

Algorithm 4.5: Ramer–Douglas–Peucker's Algorithm

Input: Trajectory $T =< p_1, \ldots, p_n >$, spatial threshold ϵ
Output: Simplified trajectory T'

1 $i \leftarrow$ index $j \in [1, \ldots, n]$ such that $d(p_j, \overline{p_1, p_n})$ is maximized;
2 **if** $d(p_i, \overline{p_1, p_n}) > \epsilon$ **then** // p_i deviates too much, split!
3 $\quad T'_{left} \leftarrow DP(< p_1, \ldots, p_i >)$;
4 $\quad T'_{right} \leftarrow DP(< p_i, \ldots, p_n >)$;
5 $\quad$ **return** joined T'_{left} and T'_{right};
6 **else**
7 $\quad$ **return** T;

Notes on information loss. The methods illustrated above are designed to guarantee that the spatial footprint of the compressed trajectories is very similar to that of the original ones, with an error bound of ϵ. However, the temporal

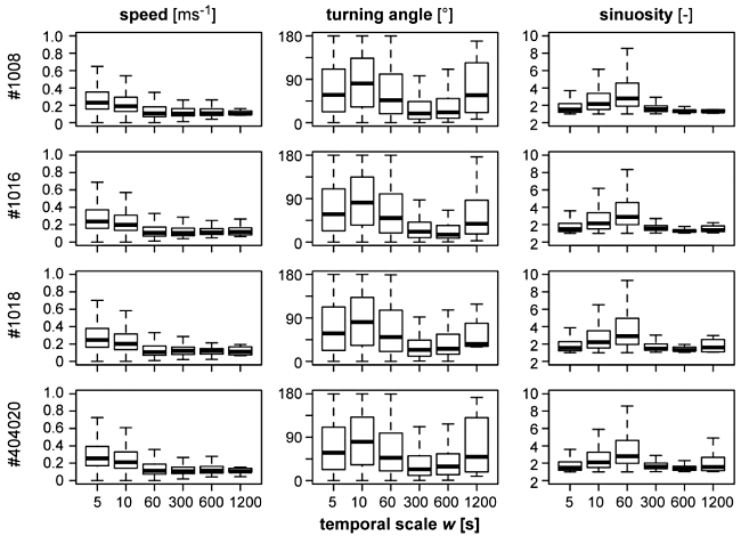
component of the movement is not considered, and consequently also all derived information such as speed and acceleration. Simplified trajectories go more straight than the original ones, thus in the same time interval they travel apparently shorter distances, shifting the distribution of speeds towards lower values. Distortions will generally appear also in the distribution of accelerations, though with more complex patterns, since in some cases smaller speeds just mean smaller speed variations, yet in others some smooth changes of speeds in the original data could be simplified into more abrupt changes.

CURIOSITY 4.2

A hot research question on sampling rates: what is the speed of a cow?

Trajectory compression and simplification techniques change data sampling and consequently also the scale of the analysis we can do with it. Simplified data fits well macroscopic analysis, whereas detailed data is better fit for microscopic analysis. Indeed, even basic movement statistics can be significantly affected by the data sampling rate. For instance, computing the distribution of speeds of cows can be rather complicated.

The research by Laube and Purves, having the rather evocative title "How fast is a cow?", shows indeed that several movement characteristics are affected just by the temporal scale of data.



As we already discussed, a movement described by fewer data points means microscopic movements are lost and thus displacements (potentially) look shorter than what they are. Hence, inferred speeds tend to be underestimated.

Laube and Purves' paper confirms this (see the first column of the figure above, showing box-plots of speed vs. temporal scale): speed becomes lower and also much less diversified. Finally, other metrics like turning angles of movements and their sinuosity are greatly affected, though with a more irregular behaviour.

Above: Photo by Filip Bunkens, Freerange Stock.
Below: Figure from Laube and Purves (2011).

4.4 Semantic enrichment

Although raw movement data only specify the spatial location of moving objects and how it changes in time, it is very often useful to associate them with additional information that can be inferred either from the context or through an analysis of the movement itself. This additional information aims to better understand what is happening in the trajectory, adding semantics to our

initial *plain* data. We will focus on two types of semantic information: stop vs. movement status, and (an educated guess of) the human activity performed.

4.4.1 Stop detection / trajectory segmentation

Trajectories simply describe the position of objects in time, not distinguishing the portions of trajectory that contain an actual movement and those that instead correspond to a stop in some location, e.g. the stay period at the work place. A well-established way to model this concept consists indeed in logically partitioning the trajectory in intervals of time that belong to the categories *stop* and *move*. If no external information is available about these intervals, we need a methodology to identify them.

While the commonsense definition of *stop* is just a total lack of movement, in practice that is not really applicable. First, we know that GPS and other localization technologies typically introduce some noise, thus even if an object remains perfectly still, its apparent position might change and total stops might be only a purely theoretical concept. This calls for some tolerance. Second, different contexts bring different notions of how large a displacement must be to consider it non-negligible, not just because not instrumentally quantifiable, but also because unimportant for the phenomenon under study. E.g. the movement that a vehicle makes within a parking lot while searching for a parking slot is not interesting for most applications, since the vehicle basically already reached its destination. We provide next a very common definition of segmentation that follows these general ideas.

Stop-and-move trajectory segmentation. Given a trajectory T, a spatial threshold δ and a temporal threshold τ, we say that a sub-sequence $T' =< p_t, \ldots p_{t+d} > \subseteq T$ represents a *stop* if all points of T' remain close to the first one ($\forall p \in T'.dist(p, p_t) \leq \delta$) and it covers a significant time interval ($time(p_{t+d}) - time(p_t) \geq \tau$). A segmentation of T can be obtained by first identifying a set of (maximal-duration) disjoint stops $T_1^{stop}, \ldots, T_k^{stop} \subseteq T$, and then define *move* sub-trajectories as the T_i^{move} exactly contained between two consecutive stops T_{i-1}^{stop} and T_i^{stop} in T. This produces a partitioning of T, schematically represented as below:

$$T = \underbrace{\mathbf{p_1}, \ldots, \mathbf{p_l}}_{T_0^{move}}, \overbrace{\mathbf{p_{l+1}}, \ldots, \mathbf{p_m}}^{T_1^{stop}}, \underbrace{\mathbf{p_{m+1}}, \ldots, \mathbf{p_u}}_{T_1^{move}} \quad \cdots \quad \overbrace{\mathbf{p_v}, \ldots, \mathbf{p_w}}^{T_k^{stop}}, \underbrace{\mathbf{p_{w+1}}, \ldots, \mathbf{p_n}}_{T_k^{move}}$$

Moves T_i^{move} represent the core movement components of the original trajectory, in other terms trips that objects perform to relocate from one position to another one. Stops T_j^{stop} are stay periods, where the object remains around (up do a distance δ from) the first point of the stop and thus can potentially be represented by a single location in space with an approximation of δ.

The formulation given above is an optimization problem that can be computationally expensive and might have several alternative solutions. A very common solution to it is the greedy approach described in Algorithm 4.6. In summary, points of the original trajectory are scanned in chronological order, and when one point is found to be an eligible start of a stop period, it is directly selected and the corresponding stop period is maximized. Then, the scan resumes by skipping all the points *captured* by the stop.

Algorithm 4.6: Stop-and-move trajectory segmentation

Input: Trajectory $T =< p_1, \ldots, p_n >$, spatial threshold δ, temporal threshold τ

Output: Set of trajectory segments *trajs_list* and stop periods *stops_list*

```
1  stops_list ← ∅;
2  trajs_list ← ∅;
3  a ← 1;
4  t ←<>;
5  while a ≤ n do
6      a′ ← max{i ∈ [a, n] | dist(i, a) ≤ δ};
7      if time(p_a′) − time(p_a) ≥ τ then // A stop: interrupt trip
8          stops_list ← stops_list ∪ {< p_a, p_a+1, ..., p_a′ >};
9          if t ≠<> then
10             trajs_list ← trajs_list ∪ {t};
11             t ←<>;
12         a ← a′;
13     else                          // Normal point: continue trip
14         Append p_a to t;
15     a ← a + 1;
16 if t ≠<> then
17     trajs_list ← trajs_list ∪ {t};
18 return trajs_list, stops_list;
```

Figure 4.9 shows a small example where two stop periods are identified, which break the trajectory into three segments: $< p_1, p_2, p_3 >$, $< p_{11}, p_{12}, p_{13}, p_{14} >$ and $< p_{19}, p_{20} >$. Notice, in particular, that the second stop selected begins

at p_15, yet according to the problem definition also the stop beginning at the next point p_16 would be equally good, actually capturing one point more (from p_{16} to p_{20}). The greedy approach, however, prefers the first acceptable one encountered, ignoring the second.

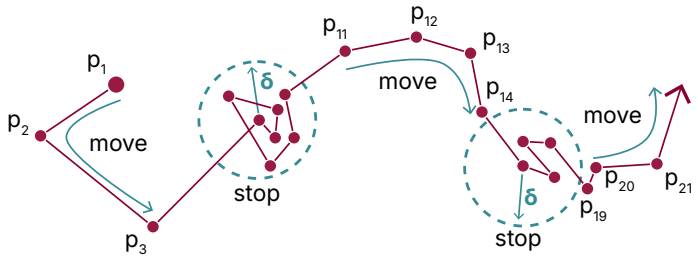


Figure 4.9 Example of stop-and-move trajectory segmentation. The algorithm identifies two stop periods, resulting into three movement segments (points 1–3, 11–14 and 19–20).

Expanding trip endpoints. From the example in Figure 4.9 it will be clear to the reader that the segments identified do not contain the start and end points of the trip performed, as they are part of the stop periods. Thus, when that is expected to be an issue it is common practice to extend each segment with an additional starting point – corresponding to the last point of the previous stop – and an additional ending point – corresponding to the first point of the next stop. Clearly, the first and last segments of the sequence are exceptions to this rule. The adjusted output for our example, then, would be: $< p_1, p_2, p_3, p_4 >$, $< p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15} >$ and $< p_{18}, p_{19}, p_{20} >$.

Transportation mode-based segmentation. When the mobility of a user is performed through a combination of transportation means, such as cars, bus, walking and bicycles, it might be useful to further split the trajectory based on which means was used. The task is clearly difficult, since modes of transportation have very variable characteristics and defining a general algorithm that works in any context might be impossible. Yet, a simple heuristics could be implemented by observing that speed is usually a good discrimination variable: walking and running people usually move in a range of speed of 5-10 km/h; bicycles are faster yet usually do not exceed 40 km/h; cars and other motorized vehicles can go faster, usually aligning with the local speed limit. An example of this approach is provided by the Spatio-Temporal Kernel Window statistics (STKW), which indirectly measures for each point in the trajectory the average speed

followed around it. More in detail, for each point p STKW measures the length of the maximal subsequence of points following p that remain within a given distance δ from it. The same thing is done for the points preceding p, and the two measures are summed up. The intuition is that in many situations the sampling rate of points is almost constant and the movements tend to be straight (at least at the small scale), which makes the number of points obtained inversely proportional to the speed of the object. Figure 4.10 plots the STKW value of points of a long trajectory in chronological order. The high values represent periods where the object moves slowly, and thus many points are needed to get far from a given location. On the opposite, very low values correspond to high speeds. We can also observe that the STKW tends to remain stable for some periods of time, confirming the intuition that a given transportation mode has its own characteristic speed.

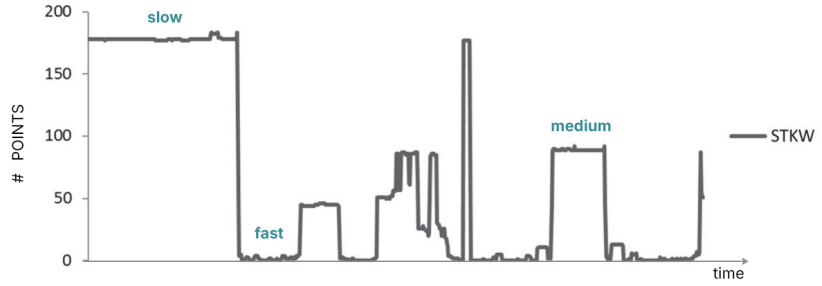


Figure 4.10 Example of Spatio-Temporal Kernel Window statistics (STKW) values for the points of a trajectory.

The STKW can be then used first to segment the trajectory by looking at the breakpoints where the value drops/increases abruptly, and then try to assign each segment to a class. This can be done either through some fixed rules (e.g. values above a threshold are stops, and values above another threshold are motor vehicles) or through some machine learning (as done in the paper which proposed STKW, in that case through a very simple neural nework).

4.4.2 Adding semantics: Home location detection

Raw trajectory data can describe to a great detail the movement of an object, and yet it lacks any explicit information about what the object is actually doing. The segmentation task described above can be seen as a very basic example of how to infer (with some approximation) some semantic information through data analysis, in that case by labeling travel periods and separating them from

stays. Here we go a step forward and try to identify the home location of an individual in the context of human mobility. The idea is very simple: while the mobility of a person can be very rich and visit (namely, stop at) several different places, in most cases they have a single reference place where they go back everyday and spend night time, typically corresponding to the main residence of the individual – home. Thus, we just need to identify the locations that they visit and then select the one that best fits the description we gave above.

From points to locations. A fist problem we have to face is the fact that if an individual stops at a location several times, each stop might happen at a slightly different GPS position. One reason is that the positioning devices bring an error which *scatters* the sensed positions around the real one at some distance. Another one could be that stops indeed happen at slightly different points, such as when the device is onboard of a car, which is parked in different slots (at a short distance) depending on parking availability.

Understanding that a set of different points are close enough to actually represent the same place is clearly a clustering problem. We discuss here one of the simplest ways to implement it, namely a radius-based grouping of points. Algorithm 4.7 summarizes it.

Algorithm 4.7: Radius-based location identification

Input: List of stop points *stops*, spatial threshold δ
Output: Set of locations L and stops-to-location mapping $f : stops \to L$

```
// Compute neighborhood size (freq) of stops
```
foreach $pivot \in stops$ **do**
  $N \leftarrow \{p \in stops \mid dist(p, pivot) \leq \delta\}$;
  $freq(pivot) \leftarrow |N|$;

```
// Build locations in frequency order
```
while $stops \neq \emptyset$ **do**
  $pivot \leftarrow \arg\max_{p \in stops} freq(p)$;
  $P \leftarrow \{p \in stops \mid dist(p, pivot) \leq \delta\}$;
  $stops \leftarrow stops - P$;
  $c \leftarrow$ mean position of P;
  $L \leftarrow$ append c;
  foreach $p \in P$ **do**
    $f(p) \leftarrow c$;
return L, f;

The expected input is the set of stops of a user, each represented by a point. Notice that the stops provided by the segmentation in Algorithm 4.6 were

actually stop periods, each containing several points (though performing no significant movement), thus some preprocessing is needed if we want to use them as input for location identification. For instance, the points of a stop period could be simply replaced by their mean position and their mean timestamp. The second parameter of the algorithm is a spatial threshold δ, representing how far can be stop points from the center of the location they belong to. The algorithm starts computing for each stop point its local density, namely the number of stops laying withing a distance δ. Then, it picks the stop with the highest density, takes it as center of a new location and assigns to it all stops within radius δ. This process is repeated several times, at each iteration assigning to the new location only points that were not previously involved in another location, till all stop points have been processed. Notice that most peripheral stop points will be processed for last, and might result in several single-point locations.

Frequency-based home detection. The number of stop points associated to a location ($|P|$ in Algorithm 4.7) represents the number of times the individual stopped there. Since home is a place an individual systematically returns to visit, in particular for resting during the night, we can typically adopt two simple approaches to select the most likely home among the locations returned from the algorithm above:

- *Pure frequency selection*: home is the location $l \in L$ with the largest number of stops, namely $home(L) = \arg\max_{l \in L} |\{p \in stops | f(p) = l\}|$;
- *Night stops*: home is the location $l \in L$ with the largest number of stops whose timestamp falls in a specific interval of hours α within the day, namely: $home(L) = \arg\max_{l \in L} |\{p \in stops | f(p) = l, hod(p) \in \alpha\}|$, where $hod(p)$ returns the hour-of-day of point p. E.g. $\alpha = [8\ p.m., 6\ a.m.]$ (or, more formally, $[0\ a.m., 6\ a.m.] \cup [8\ p.m., 12\ p.m.]$).

While the two solutions produce the same outcome in most situations, there can be cases where locations other than home are visited more frequently than home itself, therefore the second approach is considered more robust. For instance, during work days some individuals visit the workplace more than once per day (maybe twice, due to the lunch break in the middle) which might overcome the frequency of home visits. The time constraint would greatly reduce the issue.

Individual Mobility Network. The location identification process can actually be exploited to do more than just labeling home locations. Indeed, it directly allows us to translate a trajectory into a time-ordered sequence of visits to locations, each with its temporal information (stop time, departure time) and

next-location information (where the user goes after visiting that location) including features of the trips performed to move from consecutive locations (distance, travel time). This vision can be very effectively formalized as a graph where locations play the role of nodes and the trips between locations become edges connecting (corresponding) nodes. We call this representation an *Individual Mobility Network* (IMN), which is defined in a very generally way as follows.

Definition 4.1 (Individual Mobility Network) Given a user u and their mobility history H_u, we indicate with $G_u = (L_u, M_u)$ their *individual mobility network* (IMN), where L_u is the set of nodes corresponding to the locations of u, and M_u is the set of edges ($M_u \subseteq L_u \times L_u$) corresponding to direct trips between two locations. Nodes and edges are associated to a weight ω:

- $\omega(l)$ = number of trips in H_u reaching location l;
- $\omega(l_1, l_2)$ = number of trips in H_u starting from location l_1 and reaching l_2;

Also, an *extended individual mobility network* is an IMN were additional features are available, both for nodes and edges, computed through some predefined aggregation operator agg:

- $\delta(l) = agg(\{\text{durations of stops in } l\})$;
- $\rho(l) = agg(\{\text{arrival times of trips reaching } l\})$;
- $\pi_t(l) = agg(\{\text{durations of trips reaching } l\})$;
- $\pi_d(l) = agg(\{\text{lengths of trips reaching } l\})$;

The same functions are also defined on edges (movements) $(l_1, l_2) \in M_u$ in a similar way, this time considering only trips that start from l_1 and reach l_2.

Most applications focus on the ω functions, namely the frequency of nodes (i.e. number of stops at a location) and of edges (i.e. number of trips connecting two locations). We can see a few real examples of IMNs in Figure 4.11.

Here, we represent the frequency of stops at locations as nodes' size, and the frequency of trips as thickness of edges. Notice that IMNs are oriented graphs, and edges flow in a clockwise direction. For instance, in the top-left IMN the flow from '0' to '1' is slightly stronger than the flow from '1' to '0'. Finally, nodes are numbered in order of stop frequency and self-loops (i.e. trips that start and end in the same location) are also represented. We can see that IMNs are mostly characterized by a central, high-degree node (most likely corresponding to the home location of users) connected to a large number of other nodes. Often this location is connected to a second one (most likely the workplace) with high-frequency edges. This overall view of the individual mobility allows us to better understand its complexity and if/how much it is dominated by routinary

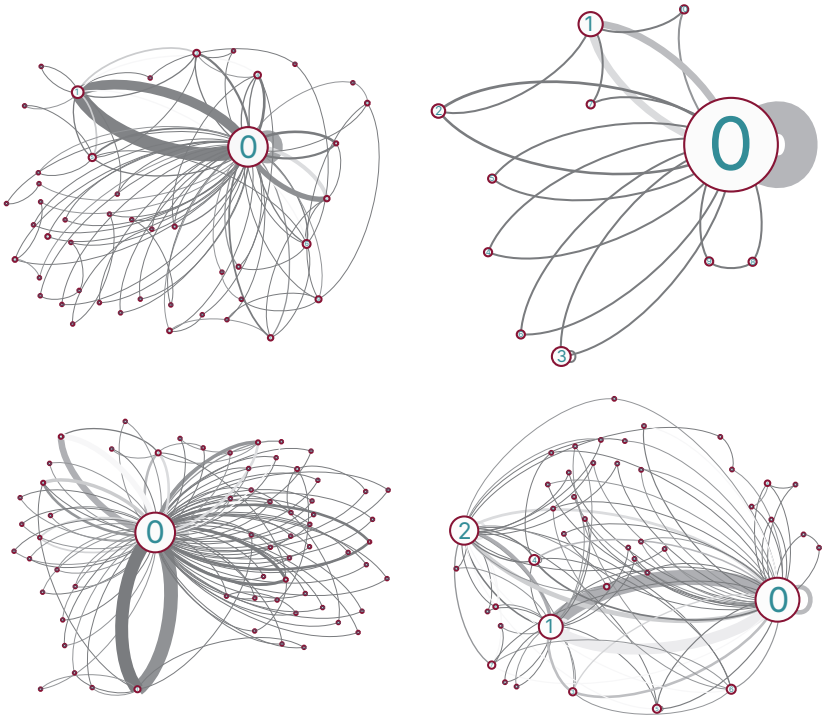


Figure 4.11 Sample IMNs. The Individual Mobility Network of four sample users along two months is shown. Size of nodes and thickness/darkness of edges represent their frequency. Nodes are numbered in order of stop frequency. [Source: paper by one of the authors.]

habits (high-frequency edges and nodes) vs. occasional events (low-frequency ones).

4.4.3 Adding semantics: Activity recognition

A step forward in understanding the mobility of an individual consists in inferring the purpose of trips and stops, namely the (most likely) activity performed there. Clearly, this is an extremely difficult task which has no general solution, and yet we can try to compute some estimates and educated guesses. A very popular approach to the problem consists in a POI-based inference, namely exploiting external information about the Points-of-Interests that are at hand from a location: simply put, if a user stops in a place where there are only restaurants it is likely that they went there for eating.

Figure 4.12 POI-based activity recognition.

The process can be applied to each stop location l of a user u, and consists of the following steps:

(i) Acquire the set R_0 of all relevant POIs in the geographical area of interest. Nowadays this can be conveniently realized thanks to open data sources like OpenStreetMap, which provide crowdsourced lists of points of interest for nearly everywhere in the world. These typically include position data, the POI category (referred to a taxonomy of categories, e.g. *bar* and *restaurant* could be both subcategories of *eating*) and sometimes opening hours.

(ii) Select the subset $R_1 \subseteq R_0$ of POIs that are at a close distance from the stop location, in most cases done by fixing a maximum distance δ that is either a global parameter (e.g. something between 100 m and 500 m) or a specific feature of user u. This δ can capture the uncertainty of position due to GPS errors (e.g. when entering a mall) or a *walking distance tolerance* for the cases where stops actually refer to a parked vehicle, not the user's position.

(iii) If opening hours are known, further filter out POIs that are closed at the time of the stop, resulting in $R_2 \subseteq R_1$. More sophisticated approaches might even consider the time needed to reach the POI from the stop point, if δ was very large. Notice that while POI-specific opening hours might be difficult to obtain, general educated guesses are usually easy to perform at the level of category type. E.g., the Bakery category can be safely excluded during night time.

(iv) For each possibly POI category type c compute its frequency $f(c)$ in R_2.

(v) Return either f (namely, the distribution of POI categories) or the most frequent category $c_{top} = \arg\max_c f(c)$.

In the example in Figure 4.12 we can see that several POIs were fetched for the area of interest, classified into 12 categories. Among them, only four POIs are within a reachable distance from the stop point (see the circle area). In this case, we obtain that $f(\text{Dentists}) = f(\text{Churches}) = 1/4$ and $f(\text{Banks}) = 1/2$. The most frequent category is then *Banks*. Results might differ is opening hours were available, since some POIs might be non-compliant with the stop time.

4.5 Notes on mobile phone data preprocessing

Mobile phone data – more exactly: CDR, XDR and the other forms of data directly collected by the mobile phone infrastructure, not through tracing apps – usually provide a much sparser and spatially coarser representation of movement trajectories. Therefore, we can expect that the preprocessing strategies discussed in the previous sections can be applied only to a limited extent:

- filtering approaches based on speed are generally applicable, though speed estimates are much coarser and thus only extreme speed errors can be detected; context-based methods are less likely to have significant impact (e.g., data points of moving cars related to phone cells that contain no roads are probably errors, yet this might be an unlikely situation in dense urban areas);
- applying point mapping and route reconstruction might be reasonable if the road network is relatively sparse, so that the set of candidate roads for each input point remains tractable;
- compression methods are likely to be not needed/usable, due to the original data sparseness;
- stop detection might be doable, yet it could be largely simplified, by just looking for phone cells that repeat within a short time (no spatial thresholds needed, since the cell granularity already plays that role);
- activity recognition might be adapted to the this context. We will discuss this in more detail in this section.

In the following we discuss two significant and representative preprocessing approaches tailored around mobile phone data: the first one is a filtering task devoted to a kind of noise specific to the data type; the second one revisits (and strongly simplifies) the home and work location detection problem.

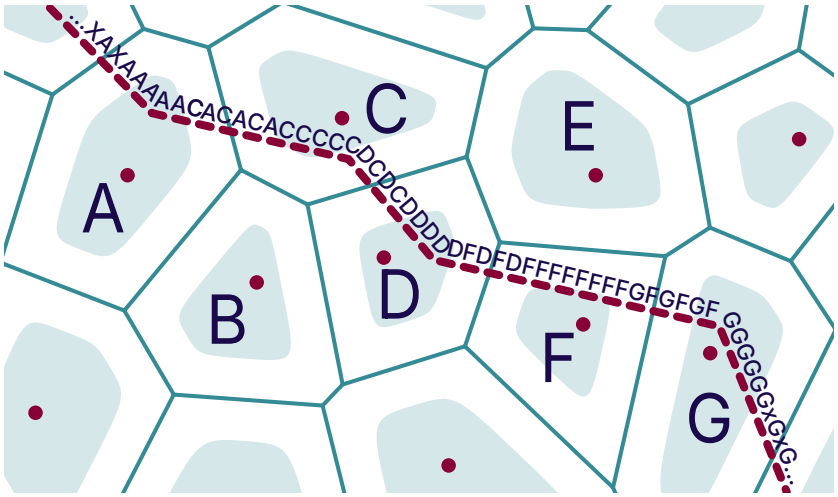


Figure 4.13 A toy example of ping-pong effect on mobile phone data. The sequence of sensed cell towers (A, B, etc.) gets discontinuous close to cell borders.

4.5.1 Ping-pong effect

Mobile phone data describe position in terms of the antenna the phone is connected to at each specific timestamp. The common assumption in analysis is then that such antenna is the closest one to the phone – in other words, the phone should always connect to the closest antenna. Actually, that is not completely true, since several factors can enter the process: if an antenna is overloaded with many connected phones it might reject other connections; obstacles between the antenna and the phone might reduce the perceived signal power, which is a key decision factor; antennas might have different capacities and power; finally, weather and other environmental changing factors can influence the antenna selection. The result is a much more complex process that can create instability of connections, especially when different antennas are at similar distance from the phone. In terms of Voroni tesselation that means we can expect instabilities close to the borders of cells, where the phone might connect to either of the two (or more) neighboring cells.

We illustrate this through the example in Figure 4.13, showing a tesselation for a set of antennas and the movement trajectory of a phone. Areas close to cell borders are depicted in white, and the small letters along the trajectory represent the connected antenna at that time – assuming a rather high sampling rate. For instance, while close to the center of cell C the phone is consistently connected to it, whereas in the middle ground between C and D the sequence

of connections is an alternation of C and D (“. . . CDCDC . . . ”). This is the so-called *ping-pong effect*, and the single antenna changes due to it are called ping-pong handovers. Relying on the raw trajectory to compute statistics and other analyses might create significant biases, since these alternating connections would be interpreted as instantaneous jumps back and forth between antennas. For instance, the apparent average speed might be orders of magnitude larger than the real one. For this reason, in some applications it is preferable to identify ping-pong effects and remove them.

Speed-heading-based detection. A very simple heuristic to find ping-pong handovers is the *speed-heading approach*, which compares each data point p_k with the previous one p_{k-1} through two parameters:

- average speed between the two points, $v_k = dist(p_k, p_{k-1})/(time_k - time_{k-1})$;
- heading change, namely how much the direction dir_k between the two points differ from the previous pair, $h_k = |dir_k - dir_{k-1}|$.

Ping-pong handovers are then estimated as the points showing a high average speed (larger than a threshold σ) and a complete U-turn change of direction, namely:

$$HO = \{p_k \mid v_k > \sigma \ \wedge \quad h_k = 180^o\}$$

Notice that the condition on direction change is sightly more general than checking a repetition of the same antenna. Indeed, usually several antennas are installed on the same tower (thus at the same coordinates), typically to cover different directions or to provide coverage redundancy, thus the antenna ID would be not sufficient to spot repeated handovers. The set HO of ping-pong handovers can then be removed from the dataset.

An example. Considering Figure 4.13, for instance, in the input sequence of connected antennas CCCCD**CDCD**DDD between C and D all the points in bold would marked as ping-pong handovers, assuming the time distance between consecutive points is small. After removal we would obtain the sequence CCCCDDDD, which contains no extra handover.

4.5.2 Anchor locations

Detecting some key locations like home with GPS data requires a relatively complex process due to the need of clustering single points into more abstract

areas. Exactly replicating this process to mobile phone data is usually not possible, and some assumptions and approximations are needed.

First, positions are already expressed as larger areas, and thus the spatial clustering phase performed on GPS points is not needed – in other words, phone cells can be directly considered as locations, since they are already at a high aggregation level.

Second, distinguishing points collected during a movement and points corresponding to stops is very difficult on mobile phone data. In particular, for sparser sources like CDRs it is practically impossible in most cases. In these situations we typically assume that all recorded points are stops, based on the rationale that movement covers just a small portion of humans' daily life and thus movement points at low sampling rates are rather rare.

Following these premises, we can implement a few simple heuristics to identify home (and, in one case, also work) locations.

Night activity-based approach for home detection. A very simple method used in literature consists in filtering data points at the source through temporal constraints, similar to the *night stops* approach we discussed for GPS data. Indeed, in this case we just fix a daily time window corresponding to night (e.g. between 10pm and the next 6am every day) and associate each phone cell the number of its data points that fit the window, the rational being that users are usually at home during night time. Thus, the cell with the highest frequency will be elected as home location. Minimum frequency thresholds are also applied, in order to avoid assigning the home label to locations with too little data to draw meaningful conclusions.

Personal Anchor Points approach. This more general heuristics aims to identify both home and work locations, and involves a few phases. We describe to some detail the main ones.

First, the analysis focuses on the cells where the user has performed calls on at least two separate days every month, considering the others as noise to remove.

Second, it selects the top two cells in terms of number of distinct days that contain calls of the user, and assumes that one of them is the home location and the other is work, though not yet deciding the exact labels.

Then, time constraints are applied to distinguish home, based on average start time (AST) of calls in each location and its deviation (std), through an handcrafted rule:

if $AST(loc) < \tau$ **and** $std(loc) < \sigma$
then WORK
else HOME

where τ represents a critical hour of the day and σ the maximum standard deviation allowed. The basic idea is that work-related calls show lower temporal variability than residential calls because work periods are more structured, while home phone usage varies widely (nights, weekends, holidays). A limitation of the approach is clearly that parameters τ and σ need to be estimated, and they can be change over different datasets under analysis. Its authors first applied it to an Estonian dataset, where the best values appeared to be τ = 5:00pm and σ = 0.175.

A case to consider is when the rule above assigns the same label to the two high-frequency locations selected. Since this is often the result of cell switching, namely phones connecting to different neighboring antennas from the same location (the same phenomenon that causes ping-pong handovers), we can check if these points are in neighboring cells. If that happens, the less frequently visited location of the two (prioritizing by days visited, then call count) is removed, and then tries again to label the third most frequent location. The same tests are applied, and the process is repeated if the label assigned is still of the same type. This repetition can go on for several iterations (e.g. the authors arrived to consider up to the fifth most frequent cell) until finding the required anchor point or when no more locations are left.

Figure 4.14 shows an example taken from the original paper, also comparing the output of the method with real home and work places. We can appreciate the effectiveness of the approach, since the estimated home and work locations are close to the real ones (with errors of 830 and 300 meters, relatively small for the city scale); but also the fact that their neighboring cells have higher frequencies than others, suggesting that in some cases the mobile phone did not connect to the closest antenna, probably preferring another antenna nearby.

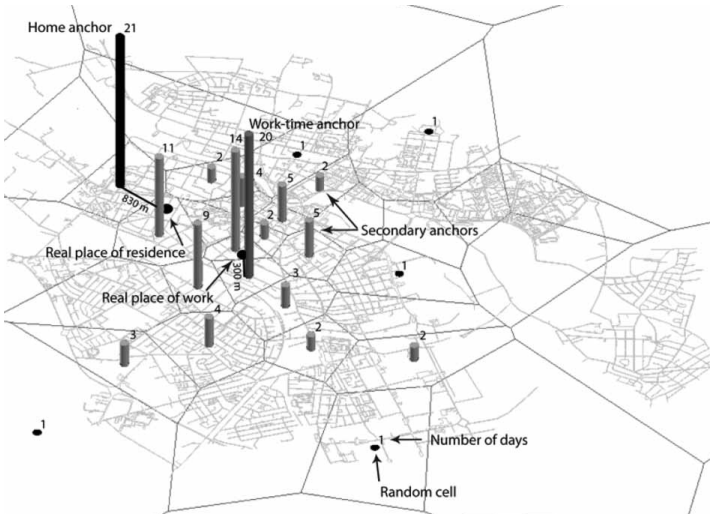


Figure 4.14 Sample user with their frequent locations and the identified home and work anchors. Figure taken from Ahas et al., J. of Urban Technology, 2010.

4.6 Homeworks and exercises

EXERCISE 4.1

How fast are users?

Here we want to replicate the cow mobility experiment mentioned in this chapter, this time on humans. To do that: (i) choose one of open mobility dataset available on internet (e.g. San Francisco taxis, Geolife, etc.); (ii) select at least 10 users/vehicles and compute their distributions of lengths; (iii) Remove 10% of points in each trajectory, either through temporal sampling (remove one point every ten) or applying a compression algorithm and playing with the parameters, and then recompute the distributions of lengths; (iv) repeat the operation above with different percentages of sampling: 20%, 30%, ..., 90%. How does the length distribution change?

EXERCISE 4.2

Estimating GPS errors

Choose a bounding rectangle covering San Francisco city. Download the road network/graph of that area. Select the GPS points of taxis available in the same area. Assign each point P to its closest road segment R. Define $pseudo\_error(P) = dist(P, R)$. Analyze the overall distribution of the pseudo-errors. Is it coherent with the official estimates of GPS errors provided by *GPS.gov*? Are pseudo-errors the same in downtown area vs. out of city?

EXERCISE 4.3

Your own "speed-aware" trajectory compression method

As we discussed in the chapter, compression methods typically do not consider time, with the effect that the output trajectories have different speed distributions compared to the original ones. Try to write your own compression algorithm, for instance a variant of Driemel's or DP, which aims to preserve speeds as much as possible. Test it on a dataset of your choice, e.g. a subset of taxis or Geolife users. Explore visually the effects of simplification on some sample trajectories, then study how the average speeds of trajectories are affected.

EXERCISE 4.4

Your own collective filtering algorithm

Typical filtering algorithms rely on the sequence of points of the individual or on static context, e.g. distance from roads. However, there might exist areas that can in theory be visited by moving objects (e.g. a road) and yet in practice they are not (e.g. the road is private or closed to traffic). Define a new filtering algorithm that works under the following assumptions:

any area or road visited by less than η distinct trajectories is actually not accessible, and thus no real point can be there. Test it on a taxi dataset.

4.7 Bibliographical notes

The interested reader can find more details about the approaches discussed in this chapter in the following papers, which were in most cases our source material.

Trajectory data cleaning is an extremely common task that however is very often treated through custom solutions by each practitioner. Beside the simple methods discussed in this chapter various others can be found in the literature, such as Kalman filters and Particle filters. A simple comparison of some of them on several datasets is provided by Garcez Duarte and Sakr (2024).

The trajectory mapping (and reconstruction) methods described in the chapter were introduced in Zhu et al. (2017) and Newson and Krumm (2009). The former is based on the classical Dijkstra' shortest path computation, which was first described in Dijkstra (2022) (from a recent collection of his writings).

Ramer-Douglas-Peucker's and Driemel–HarPeled–Wenk's algorithms for trajectory compression and simplification are provided by their respective published papers, namely Douglas and Peucker (1973) and Driemel et al. (2010). A wider survey of simplification approaches, including the previous two, is provided in Amigo et al. (2021). The half-serious discussion on cows' speed comes from the (fully-serious) paper by Laube and Purves (2011).

The stop detection heuristics have been formulated in various ways in several works, among which Bonavita et al. (2021) also discuss ways to automatically infer the temporal threshold. The modality inference approach based on the STKW statistics is instead discussed in Sila-Nowicka et al. (2016). A definition and an application of Individual Mobility Networks can be found in Rinzivillo et al. (2014), while the POI-based activity recognition approach was introduced in Furletti et al. (2013).

The two methods for home/work location detection on mobile phone data are described, respectively, in Deville et al. (2014) and Ahas et al. (2010), in both cases with a focus on CDRs. The heuristics for identifying ping-pong handovers comes from Iovan et al. (2013).

Bibliography

Ahas, Rein, Silm, Siiri, Jarv, Olle, Saluveer, Erki, and Tiru, Margus. 2010. Using Mobile Positioning Data to Model Locations Meaningful to Users of Mobile Phones. *Journal of Urban Technology*, **17**(1), 3–27.

Amigo, Daniel, Pedroche, David Sánchez, García, Jesús, and Molina, José Manuel. 2021. Review and classification of trajectory summarisation algorithms: From compression to segmentation. *International Journal of Distributed Sensor Networks*, **17**(10), 15501477211050729.

Bonavita, Agnese, Guidotti, Riccardo, and Nanni, Mirco. 2021. Individual and collective stop-based adaptive trajectory segmentation. *GeoInformatica*, **26**, 451 – 477.

Chang, Kang-Tsung. 2018. *Introduction to Geographic Information Systems*. 9 edn. Columbus, OH: McGraw-Hill Education.

Deville, Pierre, Linard, Catherine, Martin, Samuel, Gilbert, Marius, Stevens, Forrest R., Gaughan, Andrea E., Blondel, Vincent D., and Tatem, Andrew J. 2014. Dynamic population mapping using mobile phone data. *Proceedings of the National Academy of Sciences*, **111**(45), 15888–15893.

Dijkstra, E. W. 2022. *A Note on Two Problems in Connexion with Graphs*. 1 edn. New York, NY, USA: Association for Computing Machinery. Page 287–290.

Douglas, David H, and Peucker, Thomas K. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization*, **10**(2), 112–122.

Driemel, Anne, Har-Peled, Sariel, and Wenk, Carola. 2010. Approximating the Fréchet distance for realistic curves in near linear time. Page 365–374 of: *Proceedings of the Twenty-Sixth Annual Symposium on Computational Geometry*. SoCG '10. New York, NY, USA: Association for Computing Machinery.

Earle, Michael A. 2006. Sphere to Spheroid Comparisons. *Journal of Navigation*, **59**, 491–496.

Furletti, Barbara, Cintia, Paolo, Renso, Chiara, and Spinsanti, Laura. 2013. Inferring human activities from GPS tracks. In: *Proceedings of the 2nd ACM SIGKDD International Workshop on Urban Computing*. UrbComp '13. New York, NY, USA: Association for Computing Machinery.

Garcez Duarte, Mariana M, and Sakr, Mahmoud. 2024. An experimental study of

existing tools for outlier detection and cleaning in trajectories. *Geoinformatica*, **29**(1), 31–51.

Gehlke, C. E., and Biehl, K. 1934. Certain Effects of Grouping upon the Size of the Correlation Coefficient in Census Tract Material. *Journal of the American Statistical Association*, **29**(185A), 169–170.

Grünbaum, Branko, and Shephard, G. C. 1986. *Tilings and Patterns*. New York: W. H. Freeman.

Inman, James. 2012. *Navigation and Nautical Astronomy, for the Use of British Seamen*. Rarebooksclub.com.

Iovan, Corina, Olteanu-Raimond, Ana-Maria, Couronné, Thomas, and Smoreda, Zbigniew. 2013. *Moving and Calling: Mobile Phone Data Quality Measurements and Spatiotemporal Uncertainty in Human Mobility Studies*. Cham: Springer International Publishing. Pages 247–265.

Karney, Charles F. F. 2013. Algorithms for Geodesics. *Journal of Geodesy*, **87**, 43–55.

Laube, Patrick, and Purves, Ross S. 2011. How fast is a cow? Cross-Scale Analysis of Movement Data. *Transactions in GIS*, **15**(3), 401–418.

Mannila, Heikki. 2002. Local and Global Methods in Data Mining: Basic Techniques and Open Problems. Pages 57–68 of: Widmayer, Peter, Eidenbenz, Stephan, Triguero, Francisco, Morales, Rafael, Conejo, Ricardo, and Hennessy, Matthew (eds), *Automata, Languages and Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Milner, Greg. 2016. *Pinpoint: How GPS Is Changing Technology, Culture, and Our Minds*. New York: W. W. Norton & Company.

Newson, Paul, and Krumm, John. 2009. Hidden Markov map matching through noise and sparseness. Page 336–343 of: *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. GIS '09. New York, NY, USA: Association for Computing Machinery.

Openshaw, S. 1984. *The Modifiable Areal Unit Problem*. Norwich, UK: Geo Books.

Perrault, Charles. 1697. *Histoires ou contes du temps passé. Avec des moralités*. Paris: Claude Barbin. Includes the tale *Le Petit Poucet* (Tom Thumb).

Rinzivillo, Salvatore, Gabrielli, Lorenzo, Nanni, Mirco, Pappalardo, Luca, Pedreschi, Dino, and Giannotti, Fosca. 2014. The purpose of motion: Learning activities from Individual Mobility Networks. Pages 312–318 of: *2014 International Conference on Data Science and Advanced Analytics (DSAA)*.

Shylaja, B. S. 2015. From Navigation to Star Hopping: Forgotten Formulae. *Resonance*, **20**, 352–359.

Sila-Nowicka, Katarzyna, Vandrol, Jan, Oshan, Taylor, Long, Jed A., Demšar, Urška, and Fotheringham, A. Stewart. 2016. Analysis of human mobility patterns from GPS trajectories and contextual information. *International Journal of Geographical Information Science*, **30**(5), 881–906.

Sobel, Dava. 1996. *Longitude: The True Story of a Lone Genius Who Solved the Greatest Scientific Problem of His Time*. New York, NY: Penguin Putnam.

Swift, Jonathan. 2025. *Gulliver's Travels*. Cby Press. Modern edition reprint of the original 1726 publication.

Tan, P.N., Steinbach, M., and Kumar, V. 2016. *Introduction to Data Mining*. Pearson India.

Zhu, Lei, Holden, Jacob R., and Gonder, Jeffrey D. 2017. Trajectory Segmentation Map-Matching Approach for Large-Scale, High-Resolution GPS Data. *Transportation Research Record*, **2645**(1), 67–75.