Chapter 10
Mobility Data Analysis

This chapter studies algorithms for analyzing trajectory data. Existing work falls into two categories. The first one adapts machine learning algorithms originally devised for alphanumerical data to spatiotemporal data. For instance, defining a similarity or distance measure for trajectories would enable running machine learning models such as clustering, classification, and outlier mining on trajectory datasets. The second category develops specialized algorithms for trajectory analysis. Examples include the identification of trips within a long track of the moving object, and pinpointing hotspots within groups of moving object trajectories and locations frequented by numerous moving objects during certain time periods.

Section 10.1 addresses **trajectory simplification** and compression. Trajectory data can be voluminous with several thousand points in a single trajectory, hindering analysis tasks. The goal of trajectory simplification is to reduce the number of points in the trajectory while preserving the original trajectory features. Section 10.2 studies **trajectory segmentation**, which aims at splitting the trajectories into individual *trips*. Since the notion of a trip is application-dependent, we present various ways of splitting trajectories that can be adapted to most scenarios. Section 10.3 addresses **heat maps**, a common visual analytics technique for understanding the distribution of moving objects in space and time and for identifying hotspots. We continue with Sect. 10.4, which studies **trajectory similarity** with three commonly used **similarity measures**, namely, **dynamic time warp** (DTW), **Fréchet distance**, and **time warp edit distance** (TWED).

The second part of this chapter deals with clustering using the mechanisms introduced in the sections described above. In Sect. 10.5 we study classic spatial and temporal clustering, using the NYC Citi Bike dataset used in Chap. 8. We review three different kinds of clustering, namely, distance-based, density-based, and agglomerative clustering. We also review the notion of dimensionality reduction. Finally, in Sect. 10.6 we study **trajectory clustering** using the AIS dataset and applying the simplification and segmentation algorithms prior to the clustering analysis itself.

10.1 Simplification and Compression

Processing and storing mobility data is challenging, in particular due to the large volumes of such data. For instance, we explained in Sects. 5.2.1 and 9.2 that one day of AIS data in the area monitored by the Danish Maritime Authority amounts to about 2 GB. As another example, Bruxelles Mobilité, the public agency which manages transportation infrastructure in Brussels, Belgium, daily collects around 19 GB of positional data from heavy goods vehicles for toll calculation. Similarily, in the Brussels Mobility Twin,[1] a project that aims at creating an open data ecosystem for accessing and using mobility data from the Brussels-Capital Region, 1.5 GB of public transport and micromobility trajectory data are being collected every day.

To address the challenge of managing such large datasets, several simplification and compression algorithms have been developed. In this section, we introduce these techniques using a simple trajectory to give the intuition of how they work. After that, we apply these techniques to AIS ship trajectories, which typically consist of thousands of data points. The simple trajectory is created as follows:

```
CREATE TABLE RoutePoints(pt, t) AS
WITH Points(x, y, t) AS (VALUES
  (0, 0, 1), (0.5, 1.2, 2), (1, 1.7, 3), (1.5, 2.1, 4), (2, 2.5, 5), (5, 5.9, 6),
  (6, 6, 7), (8, 3.5, 8), (12, 1, 9), (14, 2, 10), (16, 3.8, 11), (18, 6, 12) )
SELECT ST_MakePoint(x, y), t FROM Points;
CREATE TABLE RouteTrajectory(line) AS
SELECT ST_MakeLine(pt ORDER BY t) FROM RoutePoints;
```

Table RoutePoints creates twelve points that represent a sample trajectory. These points include pairs of x, y coordinates, which are converted into points using the function ST_MakePoint, and a sequence number t, which is used as a timestamp. Table RouteTrajectory aggregates the route points into a line string using the function ST_MakeLine in ascending order of the sequence attribute t. The resulting trajectory is illustrated in Fig. 10.1.

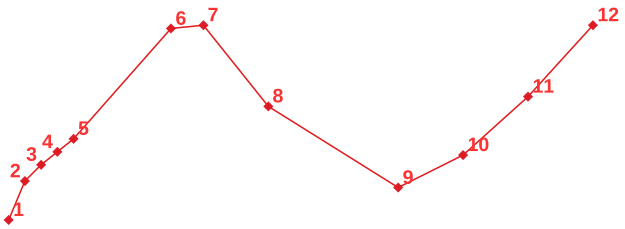


Fig. 10.1 Example trajectory used for comparing various simplification and compression methods.

[1] https://mobilitytwin.brussels/

10.1.1 Origin-Destination Simplification

The first simplification technique consists in keeping the first (origin) and last (destination) points of the trajectory as follows:

```
CREATE TABLE SimplifiedOD(line) AS
WITH Temp AS (
  SELECT pt, t, ROW_NUMBER() OVER () AS RowNo,
    COUNT(*) OVER () AS NoPoints
  FROM RoutePoints
SELECT ST_MakeLine(pt)
FROM Temp t
WHERE RowNo = 1 OR RowNo = NoPoints;
```

This query creates in table SimplifiedOD a simplified trajectory by retaining only the first and last points of the original route. In table Temp we assign a sequential row number to each point in the RoutePoints table using the ROW_NUMBER window function, while the COUNT function calculates the total number of points. The main query filters out all points except the first and the last, and uses the ST_MakeLine function from PostGIS to construct a simplified line connecting just these two points as illustrated in Fig. 10.2a.

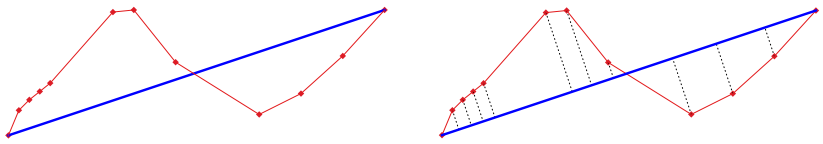


Fig. 10.2 (**a**) Original trajectory (in red) and result of the origin-destination simplification (in blue); (**b**) Distance between the original points and the simplified trajectory.

Although this simplification method may seem naive, it is widely adopted, particularly in the form of Origin-Destination (OD) data. This approach is foundational in various fields, especially in transportation planning and analysis. The basic idea is that the key information necessary for decision-making often lies in the start and end points of journeys rather than the exact path taken. By focusing on the origin and destination, planners can analyze the demand between different areas or regions, which is crucial for optimizing transportation services and infrastructure. Further, OD data are commonly represented in an OD matrix, where the origins and destinations are grouped into larger cells, such as zones or geographic areas, rather than using specific coordinates. This aggregation is heavily used by transport companies to understand passenger flows and demand patterns across various locations, allowing them to tailor services like bus routes, train schedules, or ride-sharing

networks accordingly. OD data is often collected from telecommunication companies, which aggregate the data based on mobile tower connections, or from companies like TomTom, Waze, and taxi services, which track vehicle movements. These aggregated data provides a large-scale view of mobility patterns without needing individual trip details.

While the method described above is valuable and compliant with privacy regulations such as GDPR[2] (since it aggregates data and hides personal details), it has a number of limitations. By only preserving the origin and destination, the approach loses all the details of the actual routes taken between these points. In terms of trajectory compression, this can introduce significant errors when evaluated with certain error metrics, particularly those focused on route accuracy. Figure 10.2b illustrates one such error measure, where we sum the perpendicular distances between the original points and the simplified line string. This calculation can be done as follows:

```
SELECT SUM(ST_Distance(pt, line))
FROM RoutePoints, SimplifiedOD;
-- 20.96590088691635
```

Further, the simplification omits critical data such as travel time, route choice, and deviations, leading to a potentially large error margin in applications that require fine-grained details of the movement. Thus, while effective for high-level demand analysis, it is less suitable for precise route-based analyses. Notice that this error metric is purely spatial and does not consider time. In Sect. 10.1.4 we explain how to consider time in the calculation. Also, since the original coordinates do not have units, the units of the error in this example have no meaning. However, for real geospatial trajectories, the distances are typically expressed in meters.

10.1.2 Sampling

The second idea for simplification is to keep one point out of every k consecutive points. For instance, the following query keeps one point out of two consecutive ones:

```
CREATE TABLE SampledTrajectory(line) AS
WITH Temp AS (
  SELECT pt, t, ROW_NUMBER() OVER () AS RowNo
  FROM RoutePoints )
SELECT ST_MakeLine(pt)
FROM Temp t
WHERE RowNo % 2 = 0;
```

The resulting trajectory is shown in Fig. 10.3a, while Fig. 10.3b shows the spatial deviation between the original and the sampled trajectories.

[2] https://gdpr-info.eu/

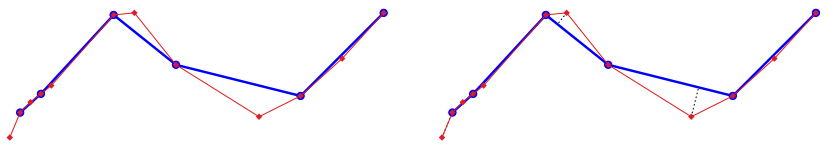


Fig. 10.3 (**a**) Original trajectory (in red) and result of the sampling (in blue). (**b**) Distance between the original points and the simplified trajectory.

This method of simplifying trajectories by sampling their points is very fast and straightforward to implement, making it an attractive option when working with large datasets. It results in trajectories that resemble the original ones in a better way compared against the Origin-Destination compression, as measured by the SQL query below.

```
SELECT SUM(ST_Distance(pt, line))
FROM RoutePoints, SampledTrajectory;
-- 3.73336399294522
```

The simplicity of this method makes it well-suited for visualization purposes, particularly when dealing with dense trajectories where observations are recorded at high frequency. In such cases, consecutive points are very close to each other and skipping points does not result in a significant loss of detail. Therefore, the technique provides a fairly accurate visual representation of the original trajectory at a low complexity cost.

An alternative sampling approach is to use temporal aggregation with the tsample and tprecision functions in MobilityDB. To explain these functions we create a spatiotemporal trajectory (i.e., a tgeompoint value) that corresponds to our example trajectory.

```
CREATE TABLE OriginalTrajectory(Trip, Trajectory) AS
WITH Points(x, y, t) AS (VALUES
  (0, 0, timestamptz '2001-01-01'), (0.5, 1.2, timestamptz '2001-01-02'),
  (1, 1.7, timestamptz '2001-01-03'), (1.5, 2.1, timestamptz '2001-01-04'),
  (2, 2.5, timestamptz '2001-01-05'), (5, 5.9, timestamptz '2001-01-06'),
  (6, 6, timestamptz '2001-01-07'), (8, 3.5, timestamptz '2001-01-08'),
  (12, 1, timestamptz '2001-01-09'), (14, 2, timestamptz '2001-01-10'),
  (16, 3.8, timestamptz '2001-01-11'), (18, 6, timestamptz '2001-01-12') ),
Route(Trip) AS (
  SELECT tgeompointSeq(array_agg(tgeompoint(ST_MakePoint(x, y), t) ORDER BY t))
  FROM Points )
SELECT Trip, trajectory(Trip)
FROM Route;
```

The tsample and tprecision functions take as arguments a time interval (e.g., 1 minute) and an optional origin of time to compute the intervals. They divide the trajectory into non-overlapping, consecutive segments based on the interval and the origin, and return one point per segment. The tsample function

selects the point at the start of each interval, which may coincide with an original trajectory point or be interpolated if necessary. In contrast, tprecision calculates a time-weighted centroid for each segment, giving a representative point for that interval. Therefore, the two functions are designed to adjust the time granularity of a moving object's trajectory, enabling transformations to coarser or finer time intervals as needed.

We apply the tsample and tprecision functions to our trajectory as follows:

```
SELECT tsample(Trip, '2 days', interp := 'linear') AS SimplifiedTsample
  tprecision(Trip, '2 days')) AS SimplifiedTprecision
FROM WorkedTrajectory;
```

We use an interval of two days, which compresses the trajectory to half its original size, as the original trajectory has one point per day.

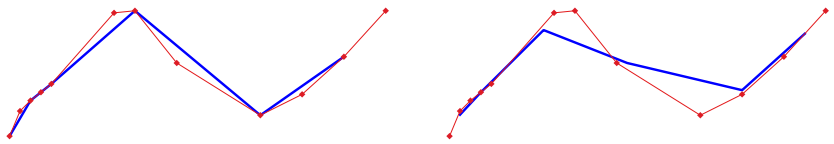


Fig. 10.4 Original trajectory (in red) and result of the compression (in blue) using (**a**) tsample, and (**b**) tprecision.

The result of tsample is illustrated in Fig. 10.4a. It resembles the earlier sampling results in Fig. 10.3a, with the difference that the sampling begins with the first point of the trajectory rather than the second. A notable feature of tsample is that it uses interpolation when the interval boundaries do not align with the original trajectory points, e.g., when we use an interval of '1.5 days'. The result of tprecision is depicted in Fig. 10.4b. Unlike tsample, the resulting points are not part of the original trajectory. Instead, they are computed using time-weighted centroid, which creates a smoothing effect. As seen in the figure, this reduces sharp fluctuations, such as the upper and lower tips of the original trajectory, resulting in a smoother representation.

It is important to note that, for compression purposes, the interval used with the tsample and tprecision functions should be larger than the original observation rate. If we use an interval that is smaller than the observation rate we can increase the number of trajectory points. Indeed, these functions transform a continuous trajectory into a time series with equi-width time intervals, making it suitable for visualization in tools like Grafana, which do not support continuous temporal data. Between the two, tprecision provides a smoothing effect by averaging points within an interval, while tsample may include noise as it simply selects the first point of each interval.

The sampling methods presented in this section have drawbacks. In the examples above, the first and/or last points of the original trajectory are not included in the result. This is typically undesirable, as the origin and destination of a trajectory represent important features. While this can be easily corrected, for example, keeping the first and last points (as done in previous section) and then applying sampling in between, it highlights a key issue: sampling does not distinguish between trajectory points of varying importance. In real-world trajectories, some points are more critical than others. These are the points where significant changes in speed or direction occur. For instance, in road networks, points at intersections or turns are vital to preserve for obtaining an accurate representation.

The main challenge in improving these methods lies in assigning weights to individual points and incorporating these weights into the compression algorithm. Since not all points in a trajectory are equally important, those where the trajectory makes sharp turns, significant speed changes, or interactions with other paths (like at junctions) should be prioritized. One possible approach could involve calculating the entropy or another measure of variability at each point, to retain the more *informative* points and discard the less significant ones. The following section illustrates a compression method that performs such weighting and selection of trajectory points.

10.1.3 Douglas-Peucker Line Simplification

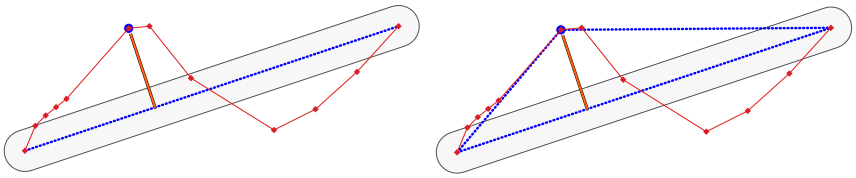
Various trajectory compression algorithms brought the idea of assign varying importance to points based on specific error minimization strategies. The Ramer-Douglas-Peucker algorithm, which is explained in this section, simplifies trajectories by minimizing the *Hausdorff error*, which measures the greatest distance between the original and the simplified paths, retaining the points with significant deviations from a straight line. It was first introduced by Urs Ramer in 1972 and later refined by David Douglas and Thomas Peucker in 1973. The refined version of this algorithm is one of the most frequently used trajectory simplification methods, and it is often named Douglas-Peucker (DP) algorithm. It is widely applied in fields such as GIS, geography, computer vision, and pattern recognition. Its popularity stems from its ease of implementation and effectiveness, as it retains critical points with significant geometric deviation while discarding redundant data.

As illustrated in Alg. 10.1, the DP algorithm is defined recursively. It takes as input a line string path $P = \langle p_1, \ldots, p_n \rangle$ and a threshold ϵ. It starts by drawing a line between the first and last points of the path and finds the point v_f that is farthest from this line (Fig. 10.5a). If the distance from v_f to the line is greater than ϵ, the algorithm recursively processes the segments between the first point and v_f, and between v_f and the last point (Fig. 10.5b). Otherwise, if the distance is within the threshold, the entire

Algorithm 10.1: Douglas-Peucker (DP) Algorithm.

Input: Line string path $P = \langle p_1, \ldots, p_n \rangle$ and threshold ϵ
Output: Simplified polygonal path
Function `DP`(P, i, j)**:**
  Find the vertex v_f between p_i and p_j farthest from line $p_i p_j$;
  $dist \leftarrow$ the distance between v_f and line $p_i p_j$;
  if $dist > \epsilon$ **then**
    `DP`(P, v_i, v_f);
    `DP`(P, v_f, v_j);
  else
    Output $v_i v_j$;

segment between the first and last points is retained and the intermediate points are discarded. This process is repeated until no points violate the distance threshold, resulting in a simplified version of the original path that approximates its shape. The complete run of the DP algorithm on the given line string is illustrated in Fig. 10.6.



(a) **(b)**

Fig. 10.5 Douglas-Peucker line simplification. (**a**) The point (blue border) with the maximal distance (orange line) to the line connecting the first and last points; (**b**) Since the distance is bigger than the threshold (gray buffer), the point is retained, and the algorithm is run recursively for the left and the right parts.

The time complexity of the DP algorithm is primarily driven by its recursive nature. For each recursive call, testing whether the line segment between two points p_i and p_j is a valid shortcut requires $O(j-i)$ time, since it checks the distance between each intermediate point and the line. In a worst-case scenario, the algorithm may need to recursively simplify almost every point, leading to recursive calls such as $DP(P, v_i, v_{i+1})$ and $DP(P, v_{i+1}, v_j)$. This results in a recurrence relation for the time complexity: $T(n) = O(n) + T(n-1)$, where n is the number of points in the path. Solving this recurrence gives a total time complexity of $O(n^2)$, meaning that in the worst case the algorithm scales quadratically with the number of points in the input path.

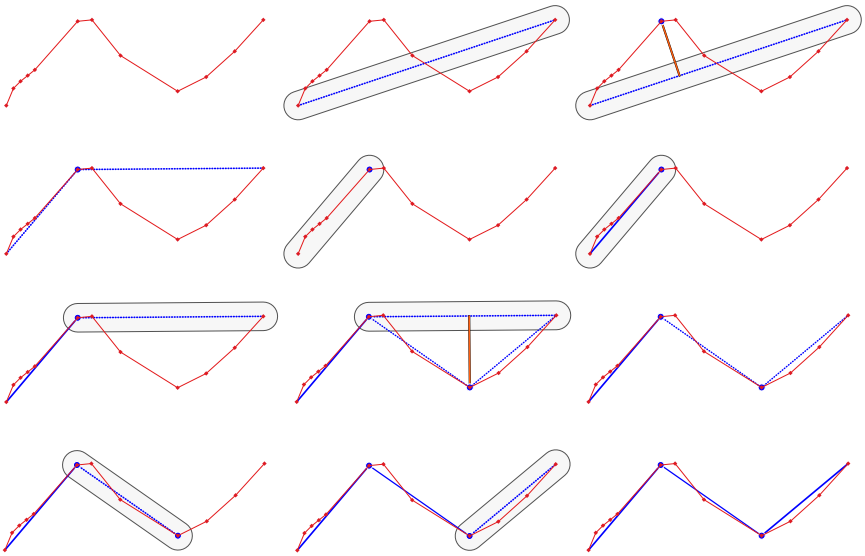


Fig. 10.6 Complete execution of the Douglas-Peucker line simplification algorithm on the trajectory in Fig. 10.1.

10.1.4 Synchronous Euclidean Distance

The Douglas-Peucker (DP) algorithm studied Sect. 10.1.3 focuses purely on the geometry of a trajectory, ignoring the time dimension. When the speed of the movement varies along various parts of the trajectory, the algorithm fails to capture these changes. An example can be seen in Fig. 10.7, where the object is moving slowly from point 6 to point 7 and then much faster between 7, 8, and 9. Since the algorithm looks at the perpendicular error of the geometries, it does not account for time in its error and thus loses this speed information in the simplification result, assuming a constant speed between timestamps 6 and 9. As a result, the time intervals between points get distorted in the simplified path. This distortion can be observed when comparing the original timestamps against the interpolated points along the simplified path. In the figure we can see the large distance between the original point 8 (in red) and the interpolated point 8 (the white circle) which corresponds to the hypothetical position of the object if it would have been moving at constant speed along the simplified trajectory. This loss of information about the speed of the movements means that DP is not appropriate for applications where time and speed are critical elements of the trajectory.

The **synchronous Euclidean distance** (SED) aims at addressing the limitations mentioned above. It considers both the spatial and the temporal dimensions and can be used as a replacement of the perpendicular distance

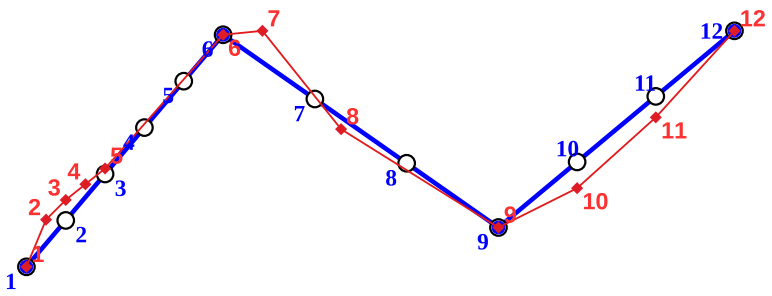


Fig. 10.7 Original trajectory (in red) and result of Douglas-Peucker simplification (in blue) using perpendicular Euclidean distance. The circles indicate the original points interpolated along the simplified trajectory assuming constant velocity.

in the DP algorithm. The idea is to interpolate points along the simplified trajectory at the corresponding timestamps, ensuring that the temporal distortion caused by the varying speeds are minimized. Consider the example in Fig. 10.8. The DP algorithm may decide to remove the point $(\mathtt{p}_1, \mathtt{t}_1)$ using the perpendicular distance to its projection $(\mathtt{q}_1, \mathtt{t}_1)$ on the line segment (vertical orange line). When using synchronous Euclidean distance, the Douglas-Peucker algorithm evaluates the distance to a different point $(\mathtt{q}_2, \mathtt{t}_1)$ (oblique orange line), which represents the interpolated position on the simplified trajectory at timestamp $\mathtt{t}_1$, calculated under the assumption of constant speed along the simplified path. As a result, this method provides a better balance between spatial and temporal fidelity, reducing errors in cases where speed and time play significant roles in the data.

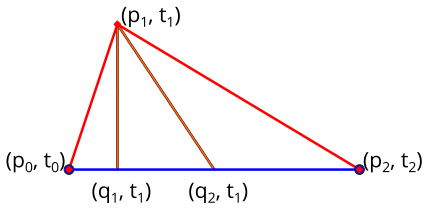


Fig. 10.8 Douglas-Peucker simplification of a trajectory (in red) with three points that occur at equi-spaced time steps $\mathtt{t}_0$, $\mathtt{t}_1$, and $\mathtt{t}_2$, showing the perpendicular distance (vertical orange line) and the synchronous Euclidean distance (oblique orange line) with the vertex point interpolated at timestamp $\mathtt{t}_1$ over the simplified line, calculated assuming constant speed along the simplified path.

The additional cost of using the synchronous Euclidean distance with the DP algorithm is the calculation of the distance at the points interpolated along the simplified trajectory based on time alignment. This additional step

makes the process slightly slower. Furthermore, using SED often results in retaining more points in the simplified trajectory than using the perpendicular distance. As a result, the overall compression ratio is reduced, providing less data reduction compared to the traditional DP method but delivering a more accurate representation when time and speed are important factors.

10.1.5 AIS Trajectory Compression

In this section, we apply the techniques presented above to the AIS use case studied in Sect. 5.2.1. We use the functions tsample and tprecision explained in Sect. 10.1.2, and the function douglasPeuckerSimplify, which implements the DP algorithm explained in Sects. 10.1.3 and 10.1.4. The latter function has two parameters: a distance parameter that corresponds to the ϵ parameter of the DP algorithm, and an optional Boolean parameter that specifies whether the synchronous Euclidean distance is used, which is set to true by default.

Figure 10.9 shows the original trajectory of a ship (in red) and three simplified trajectories (in blue) obtained with the Douglas-Peucker algorithm using a threshold of 20 m, 50 m, and 100 m, respectively, all of them using the synchronous Euclidean distance. We can see that the larger the threshold, the greater the difference between the original and the simplified trajectories. For example, the simplified trajectory in Fig. 10.9d is obtained as follows:

```
CREATE TABLE Ship_219250000_dp_100m(Trip, Trajectory) AS
SELECT douglasPeuckerSimplify(Trip, 100, true), NULL::geometry
FROM Ships WHERE MMSI = 219250000;
UPDATE Ship_219250000_dp_100m SET Trajectory = trajectory(Trip);
```

We now compare the compression rates obtained for AIS data using various methods and thresholds. We start with the Douglas-Peucker algorithm.

```
SELECT SUM(numInstants(Trip)) AS Trip,
  SUM(numInstants(douglasPeuckerSimplify(Trip, 10, false))) AS DP_10m,
  SUM(numInstants(douglasPeuckerSimplify(Trip, 20, false))) AS DP_20m,
  SUM(numInstants(douglasPeuckerSimplify(Trip, 50, false))) AS DP_50m,
  SUM(numInstants(douglasPeuckerSimplify(Trip, 100, false))) AS DP_100m,
  SUM(numInstants(douglasPeuckerSimplify(Trip, 10, true))) AS DPS_10m,
  SUM(numInstants(douglasPeuckerSimplify(Trip, 20, true))) AS DPS_20m,
  SUM(numInstants(douglasPeuckerSimplify(Trip, 50, true))) AS DPS_50m,
  SUM(numInstants(douglasPeuckerSimplify(Trip, 100, true))) AS DPS_100m
FROM Ships;
```

The douglasPeuckerSimplify function is applied to the ship trajectories with a threshold of 10 m, 20 m, 50 m, and 100 m, with either the Euclidean or the synchronous Euclidean distance (indicated by false and true, respectively). The numInstants function counts the number of instants in the trajectories, and the SUM aggregate function computes the total number of instants in all the trajectories. The result of this query is shown next.

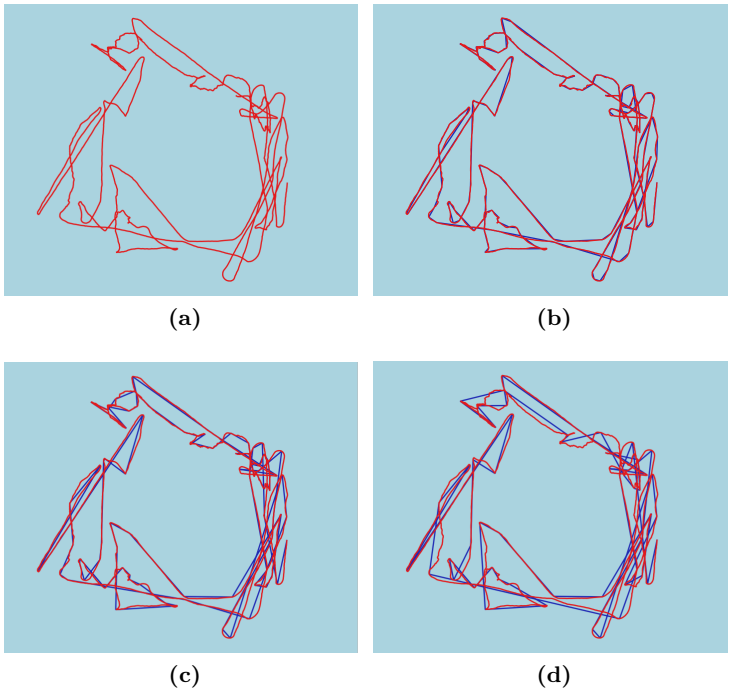


Fig. 10.9 Douglas-Peucker simplification using synchronous Euclidean distance of (**a**) the trip with MMSI 219669000 using (**b**) 20 m, (**c**) 50 m, and (**d**) 100 m.

Trip	DP_10m	DP_20m	DP_50m	DP_100m
8,743,661	196,190	116,937	69,791	48,418

DPS_10m	DPS_20m	DPS_50m	DPS_100m
641,694	326,149	138,814	76,562

We can see that the synchronous Euclidean distance produces a significantly higher number points than the Euclidean distance. This increase is smaller when the threshold is higher. Figure 10.10a shows the results graphically.

We now compare the compression obtained applying the tsample and tprecision functions explained in Sect. 10.1.2 using four different time intervals.

```
SELECT SUM(numInstants(Trip)) AS Trip,
  SUM(numInstants(tsample(Trip, '1 minute', interp:='linear'))) AS TS_1min,
  SUM(numInstants(tsample(Trip, '2 minutes', interp:='linear'))) AS TS_2min,
  SUM(numInstants(tsample(Trip, '5 minutes', interp:='linear'))) AS TS_5min,
  SUM(numInstants(tsample(Trip, '10 minutes', interp:='linear'))) AS TS_10min,
  SUM(numInstants(tprecision(Trip, '1 minute'))) AS TP_1min,
  SUM(numInstants(tprecision(Trip, '2 minutes'))) AS TP_2min,
  SUM(numInstants(tprecision(Trip, '5 minutes'))) AS TP_5min,
  SUM(numInstants(tprecision(Trip, '10 minutes'))) AS TP_10min
FROM Ships;
```

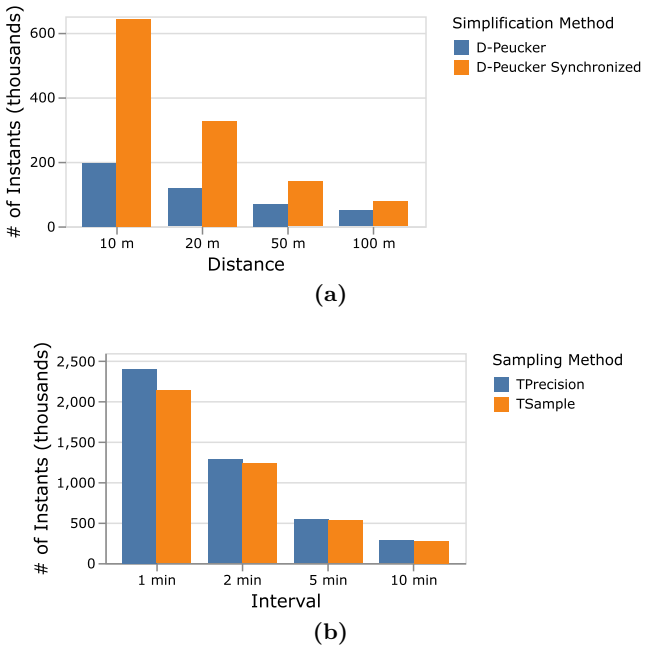


Fig. 10.10 Comparison of the compression rate over the AIS dataset: (**a**) using Douglas-Peucker simplification; (**b**) using the tsample and tprecision functions.

The result of the above query is shown next.

Trip	TS_1min	TS_2min	TS_5min	TS_10min
8,743,661	2,133,785	1,234,347	528,159	272,520

TP_1min	TP_2min	TP_5min	TP_10min
2,401,922	1,286,053	542,618	280,574

The tsample function yields slightly better compression rates, although the difference is smaller for larger intervals. Figure 10.10b shows the results graphically. Comparying with Fig. 10.10a, the compression rates are smaller than with the Douglas-Peucker algorithm, due to the time intervals used.

We analyze now the deformation between the original and the simplified trajectories. One possibility to measure this deformation is to use the similarity measures that we will study in Sect. 10.4. An alternative way is to use the temporal distance function (tdistance) to measure the distance at each instant between the original trajectory and the simplified one, resulting in a temporal float. By measuring the area under the curve of this temporal float with the function integral we obtain the overall deformation between the original and the simplified trajectories. This value is expressed in the units of the underlying coordinate system (e.g., meters) per microseconds, which is the granularity of the timestamptz type in PostgreSQL. By dividing the result of

the integral by $3,600 \times 10^9$ we obtain the result in kilo-units (e.g., kilometers) per hour. The following query implements these ideas.

```
SELECT
  round(SUM(integral(tdistance(douglasPeuckerSimplify(Trip, 10, false), Trip)) /
    (3600 * 1e9))::numeric, 2) AS DP_10m,
  round(SUM(integral(tdistance(douglasPeuckerSimplify(Trip, 20, false), Trip)) /
    (3600 * 1e9))::numeric, 2) AS DP_20m,
  round(SUM(integral(tdistance(douglasPeuckerSimplify(Trip, 50, false), Trip)) /
    (3600 * 1e9))::numeric, 2) AS DP_50m,
  round(SUM(integral(tdistance(douglasPeuckerSimplify(Trip, 100, false), Trip)) /
    (3600 * 1e9))::numeric, 2) AS DP_100m,
  round(SUM(integral(tdistance(douglasPeuckerSimplify(Trip, 10, true), Trip)) /
    (3600 * 1e9))::numeric, 2) AS DPS_10m,
  round(SUM(integral(tdistance(douglasPeuckerSimplify(Trip, 20, true), Trip)) /
    (3600 * 1e9))::numeric, 2) AS DPS_10m,
  round(SUM(integral(tdistance(douglasPeuckerSimplify(Trip, 50, true), Trip)) /
    (3600 * 1e9))::numeric, 2) AS DPS_50m,
  round(SUM(integral(tdistance(douglasPeuckerSimplify(Trip, 100, true), Trip)) /
    (3600 * 1e9))::numeric, 2) AS DPS_100m
FROM Ships;
```

The result of this query are shown next.

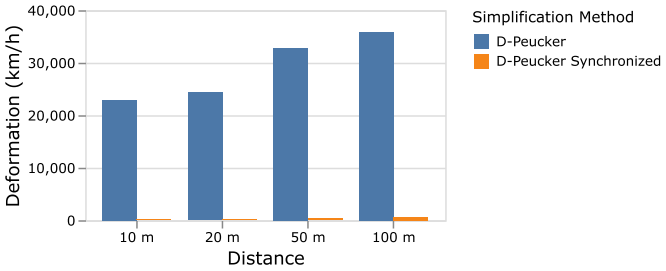
DP_10m	DP_20m	DP_50m	DP_100m
22,894.24	24,356.28	32,846.13	35,880.83

DPS_10m	DPS_20m	DPS_50m	DPS_100m
127.95	182.74	341.77	592.39

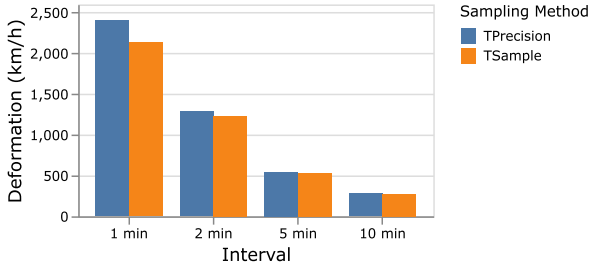
The deformation with the synchronous Euclidean distance is much smaller than with the Euclidean distance, which is also shown in Fig. 10.11a. However, as seen in Fig. 10.10a this comes at the price of a higher number of instants.

We now compare the deformation obtained with the tsample and tprecision functions in the query below.

```
SELECT
  round(SUM(integral(tdistance(tsample(Trip, '1 minute', 'linear'), Trip)) /
    (3600 * 1e9))::numeric, 2) AS TP_1min,
  round(SUM(integral(tdistance(tsample(Trip, '2 minutes', 'linear'), Trip)) /
    (3600 * 1e9))::numeric, 2) AS TP_2min,
  round(SUM(integral(tdistance(tsample(Trip, '5 minutes', 'linear'), Trip)) /
    (3600 * 1e9))::numeric, 2) AS TP_5min,
  round(SUM(integral(tdistance(tsample(Trip, '10 minutes', 'linear'), Trip)) /
    (3600 * 1e9))::numeric, 2) AS TP_10min,
  round(SUM(integral(tdistance(tprecision(Trip, '1 minute'), Trip)) /
    (3600 * 1e9))::numeric, 2) AS TP_1min,
  round(SUM(integral(tdistance(tprecision(Trip, '2 minutes'), Trip)) /
    (3600 * 1e9))::numeric, 2) AS TP_2min,
  round(SUM(integral(tdistance(tprecision(Trip, '5 minutes'), Trip)) /
    (3600 * 1e9))::numeric, 2) AS TP_5min,
  round(SUM(integral(tdistance(tprecision(Trip, '10 minutes'), Trip)) /
    (3600 * 1e9))::numeric, 2) AS TP_10min
  FROM Ships;
```



(a)



(b)

Fig. 10.11 Comparison of the deformation over the AIS dataset: (**a**) using Douglas-Peucker simplification; (**b**) using the tsample and tprecision functions.

The result of the above query is shown next.

TS_1min	TS_2min	TS_5min	TS_10min
103.61	217.66	532.25	1,212.51

TP_1min	TP_2min	TP_5min	TP_10min
2,065.97	4072.47	9,900.97	19,339.37

The deformation is bigger with tprecision since it computes the time-weighted centroid for each time bin. Figure 10.11b shows the results graphically.

10.2 Trajectory Segmentation

Segmenting moving object trajectories into individual trips is an essential task for analyzing spatiotemporal data. Moving objects, such as vehicles, ships, or even people, are often equipped with location sensors, which continuously report their positions throughout the period in which the sensors are active. In the maritime domain, the AIS transceiver of a ship reports its location every 2 to 10 seconds depending on a vessel's speed while underway, and every 3 minutes while a vessel is at anchor. Since a ship may undertake

multiple trips during the day, each with a distinct purpose, this continuous stream of location data needs to be logically segmented into individual trips to provide meaningful insights about the movement behavior of the ship. For example, when segmenting the trajectory of a cargo ship, every port visit could represent the beginning or end of a trip, depending on business logic of the shipping company. Segmenting such trajectories is necessary for analyzing shipping patterns, optimizing routes, and managing logistics.

This section illustrates three main criteria for segmenting moving object trajectories, namely, segmentation using gaps, stops, and points of interest (PoIs). Each of these criteria serves a distinct purpose depending on the context of the movement and the application. Since not every segmentation method suits all purposes, the data analyst must select the right approach for a particular use case.

10.2.1 Segmentation using Gaps

This kind of segmentation focuses on identifying significant spatial or temporal gaps in the trajectory data. For example, a trajectory may be segmented into separate trips if the distance between two consecutive recorded points exceeds a predefined threshold or if the sensor stops recording for a certain period. This method is particularly useful when movement data is sparse or when the tracking device intermittently loses signal, such as in tunnels or dense urban environments. We illustrate this technique using the AISInputTarget table from the AIS use case in Sect. 5.2.1.

First, we start by exploring whether the trajectory data includes temporal gaps, i.e., intervals during which there are no AIS records for a given ship.

```
-- Target observation period
WITH Bounds(StartTime, EndTime) AS (
  SELECT timestamp '2024-03-01 00:00:00', timestamp '2024-03-01 23:50:00' ),
-- Generate 10-minute bins over the target observation period
TimeBins(TimeBin) AS (
  SELECT generate_series(StartTime, EndTime, interval '10 minutes')
  FROM Bounds ),
-- Get for each MMSI the times of observations binned to 10-minute intervals
Observations(MMSI, ObsBin) AS (
  SELECT DISTINCT MMSI, date_bin('10 minutes', T, StartTime)
  FROM AISInputTarget, Bounds
  WHERE T >= StartTime AND T <= EndTime ),
-- MMSI of the ships
ShipIds(MMSI) AS (
  SELECT DISTINCT MMSI FROM AISInputTarget )
-- Create a matrix where each cell represents whether an observation exists (1) or not (0)
SELECT s.MMSI, b.TimeBin, COUNT(o.ObsBin) > 0 AS HasObservations
FROM ShipIds s CROSS JOIN TimeBins b LEFT JOIN Observations o ON
  s.MMSI = o.MMSI AND b.TimeBin = o.ObsBin
GROUP BY s.MMSI, b.TimeBin;
```

The query above generates a bitmap that determines whether the vessels (identified by their MMSI) have any observation during ten-minute intervals throughout a given day, namely, March 1st, 2024. Table TimeBins partitions the time line into ten-minute intervals using the generate_series function. Table Observations computes for each MMSI the timestamps of its observations truncated to the containing ten-minute interval using the date_bin function. An example of the use of this function is given next:

```
SELECT date_bin('10 minutes', '2024-03-01 11:48:23', '2024-03-01');
-- 2024-03-01 11:40:00
```

Finally, the main query performs a CROSS JOIN between the MMSI values and the generated time intervals, ensuring that each vessel is matched with every ten-minute bin of the day. A LEFT JOIN is then used to associate the vessels and time intervals with the observation data, accounting for cases where no observation exist. The expression with the COUNT function returns a Boolean value that determines whether any observation occurred within each time bin for a given vessel. The result of the query, shown in Fig. 10.12, is composed of rows containing an MMSI, a time interval, and a Boolean value that indicates whether an observation exists for the ship during the time interval. This result can be used to generate a matrix to visualize data gaps or to analyze the temporal availability of AIS data for each vessel as in Fig. 10.13. The figure was generated using Matplotlib in the ExploreAISGaps Python notebook that can be found in the companion GitHub repository.

mmsi integer	timebin timestamp without time zone	hasobservations boolean
174	2024-03-01 00:00:00	false
174	2024-03-01 00:10:00	false
174	2024-03-01 00:20:00	false
174	2024-03-01 00:30:00	false
174	2024-03-01 00:40:00	false
174	2024-03-01 00:50:00	false
174	2024-03-01 01:00:00	false
174	2024-03-01 01:10:00	false

Fig. 10.12 Portion of the result of the query that is used to build the bitmap in Fig. 10.13. The column HasObservations indicate the presence (true) or absence (false) of AIS observations during 10-minute intervals.

As this analysis shows that there are temporal gaps in the data, we next want to use these gaps for segmenting the ship trajectories into trips. In MobilityDB, the constructor functions for sequence set values account for value and/or time gaps. Thes constructors accept as first argument an array of the

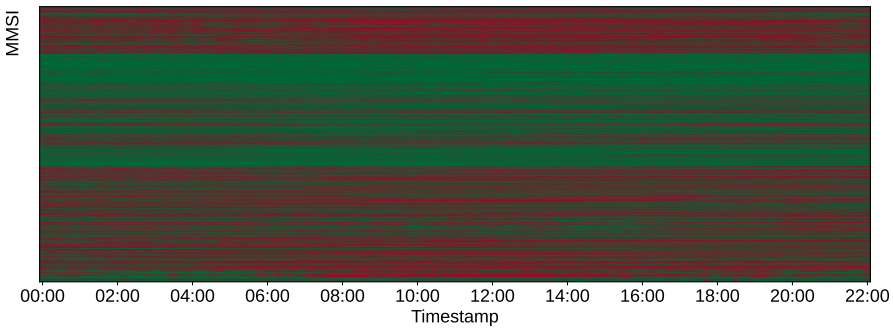


Fig. 10.13 Bitmap showing the presence (green) or absence (red) of AIS observations at 10-minute intervals (x-axis) for the ships identified by their MMSI (y-axis).

corresponding instant values, and two optional arguments stating a maximum time interval and a maximum distance such that a gap is introduced between sequences of the result whenever two consecutive input instants have a time gap or a distance greater than these values. For temporal points, the distance is specified in units of the coordinate system. We explore these constructors by revisiting the query we showed in Sect. 5.2.1 that creates the Ships table. Now, we create a similar table but accounting for spatial gaps of 1 km and temporal gaps of 30 minutes as follows.

```
CREATE TABLE ShipsWithGaps(MMSI, Trip, SOG, COG) AS
WITH Gaps(SpaceGap, TimeGap) AS (
  SELECT 1000, interval '30 minute' )
SELECT MMSI,
  tgeompointSeqSetGaps(array_agg(tgeompoint(Geom, T) ORDER BY T),
    TimeGap, SpaceGap),
  tfloatSeqSetGaps(array_agg(tfloat(SOG, T) ORDER BY T)
    FILTER (WHERE SOG IS NOT NULL), TimeGap),
  tfloatSeqSetGaps(array_agg(tfloat(COG, T) ORDER BY T)
    FILTER (WHERE COG IS NOT NULL), TimeGap)
FROM AISInputTarget, Gaps
GROUP BY MMSI, SpaceGap, TimeGap;
```

We analyze the number of gaps found in the dataset with the following query.

```
SELECT MIN(numSequences(Trip)), MAX(numSequences(Trip)),
  AVG(numSequences(Trip))
FROM ShipsWithGaps;
-- 1 | 19 | 2.2506426735218509
```

We can show the gaps for each trip as follows.

```
SELECT MMSI, timeSpan(Trip) - getTime(Trip)
FROM ShipsWithGaps
WHERE numSequences(Trip) > 1
ORDER BY numSequences(Trip);
```

The result of this query is given next.

```
/* 232035207 | {(2024-03-01 18:05:27, 2024-03-01 19:29:12)}
  232046826 | {(2024-03-01 23:00:04, 2024-03-01 23:30:13)}
  311379000 | {(2024-03-01 19:30:46, 2024-03-01 20:57:25)}
  ... */
```

The gaps are constructed by computing with function timeSpan the overall time span of a trip from the first to the last instant, from which we subtract the time over which the trip is defined obtained with function getTime.

Figure 10.14 shows the trajectories of two ships with MMSI 211608700 and 630001042, both containing gaps. The overall trajectories of the ships, as recorded in the database table Ships, are shown in black, while the individual trips derived from the query are displayed in white. The figure highlights how spatial and temporal gaps in the original trajectories had been linearly interpolated. Segmenting these trajectories into trips, as shown in the figure, may lead to more accurate results in further analysis.

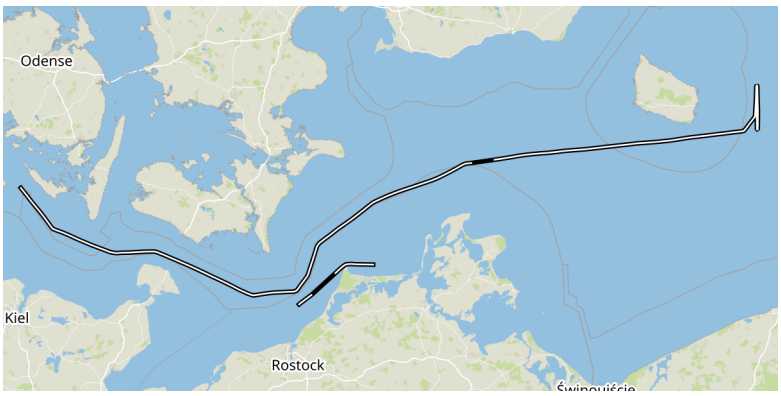


Fig. 10.14 Trajectories of two ships (thick black lines) with MMSI 630001042 and 211608700 segmented into individual trips (thinner white lines) using spatial or temporal gaps.

10.2.2 Segmentation Using Stops

This kind of segmentation aims at identifying points where the moving object remains stationary for a certain amount of time. In this approach, a stop might represent a meaningful pause in the trajectory, such as car parking at a destination or a ship anchoring at a port. These stops mark natural divisions between trips and can be used to understand the purpose of the movement, such as distinguishing between work and leisure trips. This criterion is useful when the stops define the semantics of the trips.

We must take into account that segmenting a trip at every stop can cause problems because not every stop is important. For example, a moving object like a car or a ship might make short pauses due to reasons like stopping at traffic lights, waiting briefly at a signal, or even slowing down momentarily. These are minor interruptions and should not be treated as stops that separate trips. Additionally, GPS errors can create false stops where the object appears to have paused, even though it was still moving. If we split the trip at every single stop, in case those minor pauses or errors occur, the trip would be broken into many unnecessary parts. For instance, imagine a car making ten brief stops at traffic signals over a 30-minute drive. If we consider each stop as a new trip, we end up with ten short trips instead of one complete journey, making it difficult to analyze the true path and behavior of the car. To avoid this, it is important to ignore short, nonsignificant stops, and only focus on meaningful breaks in the journey, like when the object stops for a long time at a destination.

The next query implements the ideas above, segmenting ship trajectories into distinct trips using periods of long stops to perform the segmentation.

```
CREATE TABLE TripsByStops(MMSI, TripId, TripSegment, TrajSegment) AS
WITH StopDetection(MMSI, StopSpan) AS (
  SELECT MMSI, unnest(spans(whenTrue(speed(Trip) #< 1)))
  FROM Ships ),
ShipStops(MMSI, Stops) AS (
  SELECT MMSI, spanset(array_agg(StopSpan ORDER BY StopSpan))
  FROM StopDetection
  WHERE duration(StopSpan) >= interval '60 minutes'
  GROUP BY MMSI ),
TripsWithStops(MMSI, TripSegment) AS (
  SELECT s.MMSI, unnest(sequences(minusTime(Trip, Stops)))
  FROM Ships s, ShipStops t
  WHERE s.MMSI = t.MMSI AND minusTime(Trip, Stops) IS NOT NULL )
SELECT MMSI, ROW_NUMBER() OVER(PARTITION BY MMSI), TripSegment,
  trajectory(TripSegment)
FROM TripsWithStops;
```

The StopDetection table identifies periods during which a ship's speed falls below a certain threshold, in this case, less than 1 m/s. This is done with the expression unnest(spans(whenTrue(speed(trip) #< 1))), which extracts the periods when the speed condition is true, marking those as potential stop intervals for further analysis. Table ShipStops retains only those stops that last for at least 60 minutes with function duration. The rationale here is that long stops (such as docking or anchoring) are more meaningful for trip segmentation since they represent significant pauses in the ship's journey. The selected stops are then aggregated into a collection of stop intervals (a spanset) for each ship in column Stops. These intervals are important because they allow us to define the portions of the trip representing inactive periods that are to be excluded. Table TripsWithStops segments the ship's movement into distinct trips by excluding the long stop intervals. Function minusTime is used to subtract the aggregated stop intervals from the original ship trajectory,

leaving only the segments where the ship was actively moving. The result is a tgeompointseqset. The sequences and the unnest functions are used to extract the individual sequences. Finally, in the main query the segments are assigned a TripID with the ROW_NUMBER window function.

The query below shows the number of segments obtained for the ships.

```
SELECT MMSI, MAX(TripId) AS NoSegments
FROM TripsByStops
GROUP BY MMSI;
/* 219029272 | 2
  219032932 | 2
  219006219 | 6
  265714460 | 1
  636018261 | 1
  ... */
```

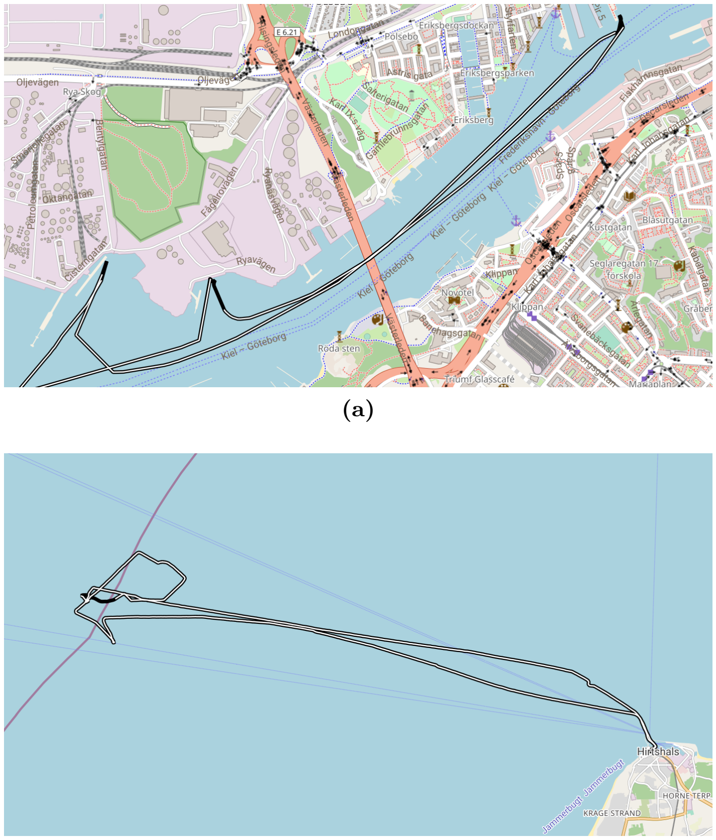
This approach allows for effective trip segmentation based on periods of inactivity, ensuring that long pauses in movement (such as docking or anchoring) do not distort the analysis of ship movements. The segmented trips can then be used for further analysis, such as calculating distances traveled, trips duration, or patterns in maritime activity.

Figure 10.15a illustrates the segmentation of the trajectory of a ship, where three stops are identified close to harbors. In certain cases, segmenting trips at these harbor stops might be useful, as it can help with specific analyses, such as monitoring docking times or harbor visits. Figure 10.15b presents another example resulting from the same segmentation query. Here, a long stop is detected during what seems to be fishing activity. Depending on the application, splitting a trip that represents a fishing activity might not be appropriate, as the activity is part of a continuous fishing process. This highlights that not every segmentation method suits all purposes, the right approach depends on the data and the application.

10.2.3 Segmentation Using Points of Interest

The segmentation using points of interest (PoI) is based on the relevance of specific locations to the moving object. A PoI is typically a place that has significance to the purpose of the trajectory, such as a workplace, school, shopping center, or transportation hub. Segmenting using PoI assumes that a trip starts and ends at source and destination PoIs. Thus, the locations of the PoIs can be used to split the moving object trajectory into segments that navigate between pairs of these PoIs.

To illustrate this method on the AIS dataset, let us consider again the ferry example in Fig. 5.17, which we have introduced in Sect. 5.2 and studied in Queries 5.7 and 5.9. While the ferry has done several trips between Rødby and Puttgarden, in the data what we got is a single ship trajectory for the whole day. It seems reasonable to segment this long trajectory into trips that



(a)

(b)

Fig. 10.15 Segmentation using stops (**a**) for the ship with MMSI 265550210, illustrating long stops detected at harbors, and (**b**) for the ship with MMSI 219005496, likely indicating fishing activity that includes a prolonged stop.

commute between the two ports. The query below creates a table called Ports, which stores geographic information for the two ports.

```
CREATE TABLE Ports(PortName, Geom) AS
SELECT * FROM (VALUES
 ('Rodby', ST_Buffer(ST_Point(651295, 6058400, 25832), 1000)),
 ('Puttgarden', ST_Buffer(ST_Point(644639, 6042308, 25832), 1000)));
```

The query defines for each port its name and its location. The ST_Point function creates a point from the provided coordinates in the SRID 25832. The point is then buffered by 1,000 meters using the ST_Buffer function, creating a circular area around the port. Next, we use these geometries for segmenting the ferry trajectory.

```
CREATE TABLE TripsByPois(MMSI, TripId, TripSegment, TrajSegment) AS
WITH PortIntersections AS (
  SELECT MMSI, ST_Union(Geom) AS Geom
  FROM Ships s, Ports p
  WHERE eIntersects(s.Trip, p.Geom)
  GROUP BY MMSI ),
TripsMinusIntersections(MMSI, Trip) AS (
  SELECT s.MMSI, unnest(sequences(minusGeometry(s.Trip, h.Geom)))
  FROM Ships s, PortIntersections h
  WHERE s.MMSI = h.MMSI )
SELECT MMSI, ROW_NUMBER() OVER (PARTITION BY MMSI), Trip,
  trajectory(Trip)
FROM TripsMinusIntersections;
```

In table PortIntersections, the eIntersects predicate finds the ships that intersect at any point in time a given port geometry. All the geometries of the intersecting ports are then aggregated into a single geometry using the ST_Union function, resulting in one aggregated geometry per ship. In table TripsMinusIntersections, the aggregated intersection geometries are subtracted from the ship's trajectory using the minusGeometry function, removing the portions of the trajectory that intersect with the ports. Recall that the port region is a circular buffer of a diameter of 1,000 meters. The remaining parts of the trajectory are split into individual trip segments using the sequences and the unnest functions. Finally, in the main query each segment is assigned a unique identifier using the ROW_NUMBER function. The result is a table that contains segmented trips for each ship, partitioned with the visits of the two ports. Figure 10.16 illustrates the result, where the white segments represent the individual trips resulting from the segmentation process.

The above query can be extended to segment trajectories using many PoIs, for example, all harbors in Denmark. We can download the ports using the Overpass Turbo API[3] from OpenStreetMap (OSM). This API allows users to query the OSM database using the Overpass query language, denoted Overpass QL. In this way, users can access detailed geographic data based on certain criteria, such as tags, locations, and object types (e.g., nodes, ways, relations). The Overpass Turbo API provides an interface for building and running these queries in a browser, while the Overpass API enables developers to send such queries programmatically as illustrated next.

```
# Library imports
import overpy
api = overpy.Overpass()
query = """
area["name"="Danmark"]->.boundaryarea;
(node(area.boundaryarea)["harbor"];);
out center;
"""
# Fetch the data from OSM using the Overpass API
result = api.query(query)
```

[3] https://overpass-turbo.eu/index.html

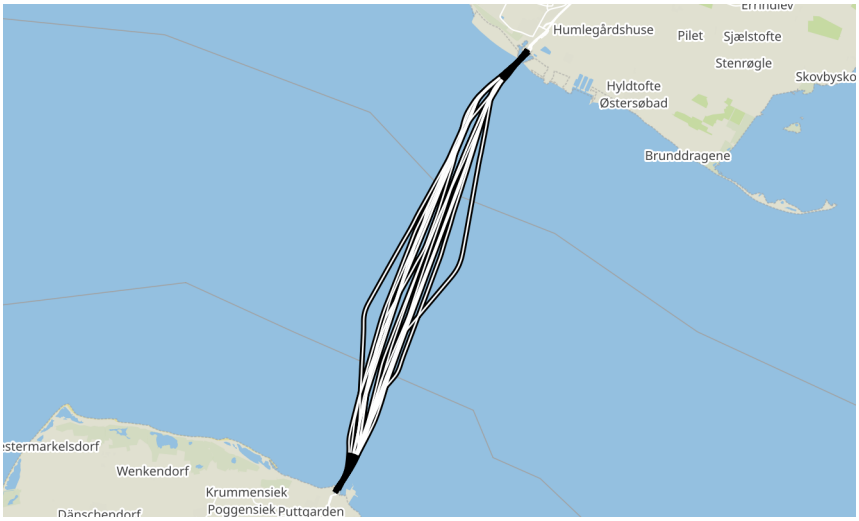


Fig. 10.16 Ferry trajectory of MMSI 211188000 segmented into individual trips navigating between the ports of Rødby and Puttgarden.

The script above uses the overpy library, which provides a Python wrapper for interacting with the Overpass API. The script initializes the Overpass API and then constructs a query to retrieve all nodes tagged as harbor within the boundary of Denmark. The results are retrieved using the api.query function and the corresponding geographic data for harbors is returned in result.

The script below allows us to visualize the results of the API call above.

```
# Library imports
import folium
import random

# Initialize a folium map centered around Denmark
danish_harbors = folium.Map(location=[56.26392, 9.501785], zoom_start=7)

buffer_radius = 6000 # Adjust the buffer size as necessary

def get_random_color():
 return "#{:06x}".format(random.randint(0, 0xFFFFFF))

# Add the harbors to the map with a circular buffer, with randomly colored circles
for node in result.nodes:
  folium.Circle(location=[node.lat, node.lon],
    radius=buffer_radius, color=get_random_color(),
    fill=True, fill_color=get_random_color(),
    fill_opacity=0.6).add_to(danish_harbors)
# Visualize
danish_harbors
```

The data structure returned by api.query is an object that organizes the results of the Overpass query into various categories: nodes, ways, and relations. For each category, the corresponding objects are returned as lists where each element represents a geographical feature. In the case of nodes, for example, each node object includes attributes such as lat (latitude), lon (longitude), and tags, where tags contains key-value pairs representing OSM tags (like harbor or other metadata). Similarly, for ways and relations, the data includes nodes or references to other objects, tags, and geographic information. This structure allows access to both the spatial data and the metadata associated with each feature, enabling further analysis or visualization in Python. The resulting visualization is shown in Fig. 10.17.

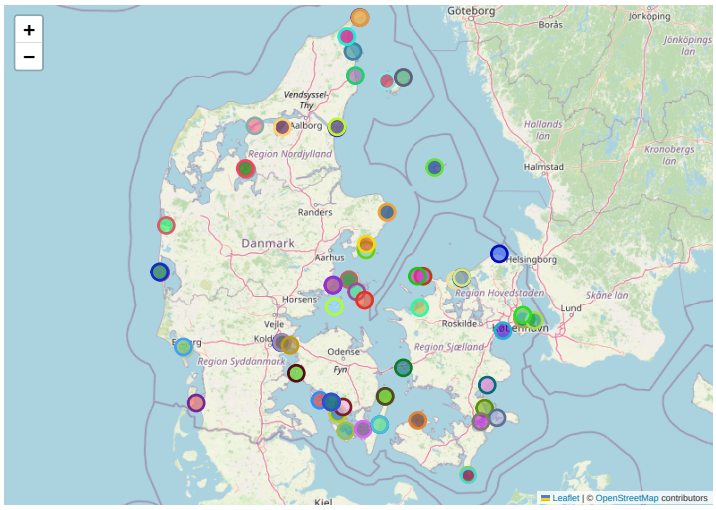


Fig. 10.17 Harbors in Denmark fetched using the Overpass Turbo API.

Next we store these harbors into the database that contains the ship trajectories. For this, we create the following table.

```
CREATE TABLE Harbors (
  Id serial PRIMARY KEY,
  osm_id bigint UNIQUE,
  Name varchar(255),
  Geom geometry);
```

The Harbors table is populated using a Python script that can be found in the companion GitHub repository. We show next relevant portion of this script.

```
# Database connection settings
connection = pg.connect(...)
# Open a cursor to perform database operations
with connection.cursor() as cursor:
```

```
# Define an insert query
insert_query = """
 INSERT INTO Harbors (osm_id, name, geom)
 VALUES (%s, %s, ST_Point(%s, %s, 4326)
 ON CONFLICT (osm_id) DO NOTHING;
"""
 # Insert nodes
 for node in result.nodes:
  lat = node.lat
  lon = node.lon
  osm_id = node.Id
  # Get name, 'Unnamed' if not available
  Name = node.tags.get('name', 'Unnamed')
  cursor.execute(insert_query, (osm_id, name, lon, lat))
connection.commit()
connection.close()
```

Note that the API in the script returns the latitude and longitude of each harbor in the result.nodes object. The INSERT statement in the query transforms these values into a point geometry with SRID 4326.

As a final step we must update the resulting Geom attribute projecting the coordinates into a CRS suitable for the Danish region and create a circular region of 1,000 meter diameter around the harbor points as follows.

```
UPDATE Harbors SET Geom = ST_Buffer(ST_Transform(Geom, 25832), 1000);
```

Now the Harbors table can be used instead of the Ports table in the query that segments using PoIs explained before.

10.3 Heat Maps

Heat maps represent numerical data by mapping the data values to color gradients. Heat maps are especially useful to discover patterns and trends in large datasets. The data values are typically converted into a color gradient, where the intensity of each color is a linear function of the underlying data value. This provides a smooth transition between colors, reflecting continuous changes in the data. Heatmaps were introduced in Chap. 4.

Heat maps are widely used for spatial data visualization, but for spatiotemporal data the time dimension is of course relevant. More often, spatiotemporal data are visualized as purely spatial data, neglecting the time aspect, which can lead to misinterpretations. In this section, we start by studying general heat map characteristics and then explain how time can be included in heat map visualizations, ensuring that temporal patterns are properly represented and not overshadowed by the spatial dimension.

10.3.1 General Characteristics of Heat Maps

A **heat map** is a two-dimensional data visualization technique that represents the magnitude of individual values within a dataset as a color gradient which can be represented by hue or intensity. We have shown examples of heat maps in Figs. 4.21 and 4.26 in Chap. 4.

Although heat maps have been used by statisticians since a long time, they have become widely used in data science applications in the last decades. In the data science workflow studied in Chap. 1, heat maps are used for exploratory data analysis. Heat maps are also used in website analysis, for example, to analyze the scrolling behavior of users, and in human-computer interaction, like in mouse, eye, and click tracking to detect where the users attention is focused. Other fields where heat maps are used include biology, for example, to visualize patterns and similarities in DNA, RNA, or gene expression, and in sports, for example, to analyze patterns in the games, like sectors in the field most frequently occupied by players.

In geographical visualization, heat maps are used to show the spatial distribution of data, representing various densities of data points on a geographical map to discover the strength of certain phenomena. Although **choropleth maps** maps, which were presented in Chapter 4, also represent the spatial variation of the intensity of a certain variable, they differ from heat maps in many ways. Choropleth maps show data grouped by geographic boundaries like countries or states, each one having one or more associated values (such as population) that are visualized, for example, by color intensity or shading. Heat maps may also visualize data over a geographic region although they show the proportion of a variable over a usually small grid size, independently of geographic boundaries.

There are two main types of heat maps. A *spatial* or *spatiotemporal heat map* displays the magnitude of a spatial (or spatiotemporal) phenomena using a color gradient, usually cast over a map. A *grid heat map* displays magnitude as color gradient in a two-dimensional matrix, with each dimension representing a category and the color representing the magnitude of some measurement on the combined value of a variable over the two categories. We gave an example in Fig. 4.10, which shows the salinity value in the Schelde river, where the dimensions represent, respectively, the hour of the day and the days spanning the measures. Further, grid heat maps are categorized into two different types of matrixes: clustered and correlogram (or correlation heat map). Figure 4.10 corresponds to the former class. A correlation heat map shows a 2D correlation matrix between two discrete dimensions using colored cells to represent data from usually a monochromatic scale.

10.3.2 Spatial and Spatiotemporal Heat Maps

We now apply the concepts introduced in the previous section to mobility data. We give examples of both, spatial and spatiotemporal grid maps using AIS data with the aim of identifying hotspots of the movement of the ships in space and time in the Denmark–Sweden region. We start by splitting the trips with respect to a spatiotemporal grid, which will be used later on for constructing the heat maps.

```
CREATE TABLE TripTiles(TileGeom, TileTime, Trip) AS
SELECT (Rec).Point, (Rec).Time, (Rec).Tpoint
FROM (
  SELECT spaceTimeSplit(Trip, 10000, interval '1 hour') AS Rec
  FROM Ships ) AS Tmp;
```

The query uses the function spaceTimeSplit, which is part of the **multidimensional tiling** mechanisms in MobilityDB (see Sect. 8.5). The spaceTimeSplit function fragments the ship trajectories into spatiotemporal tiles of 10,000 meters and one hour. This results in the TripTiles table, which contains for each tile the lower-left point in TileGeom, the lower time in TileTime, and the fragment of a trip restricted to the tile in Trip.

As we will see next, the trip fragments are not needed for heat maps based on count values. However, the fragments are needed for heat maps based on particular features of the trips while in the spatiotemporal tile, such as speed or heading. Note also that by setting the spatial and temporal grid sizes, the user controls the granularity of the analysis. Smaller tiles result in more detailed heat maps, while larger tiles provide a more general overview.

Spatial Heat Maps

Once the trajectories are split into spatiotemporal tiles, we can aggregate them. The next query computes for each spatial tile the number of times it was visited by a ship, regardless the time instant.

```
CREATE TABLE SpatialHeatMap AS
SELECT TileGeom, COUNT(*) AS ShipCount
FROM TripTiles
GROUP BY TileGeom;
```

We now build a heat map from this query. Areas with a high value of ShipCount will be shown as high-density regions in the heat map. The following Python script illustrates such visualization using psycopg,[4] a well-known PostgreSQL database adapter for Python, and the Folium library.

```
# Library imports
import psycopg as pg
import pandas as pd
import numpy as np
```

[4] https://pypi.org/project/psycopg/

```
import folium
from folium.plugins import HeatMap

# Connect to the database
connection = pg.connect(...)

# Query for the spatial heat map
query = """
  SELECT ST_X(ST_Transform(TileGeom, 4326)) AS lon,
    ST_Y(ST_Transform(TileGeom, 4326)) AS lat, ShipCount
  FROM SpatialHeatMap;
"""
df = pd.read_sql_query(query, connection)

# Close the connection
connection.close()
# Apply log transformation to better visualize the data
df['log_count'] = np.log(df['ShipCount'] + 1)

# Create a base map centered on Denmark using Folium
m = folium.Map(location=[55.6761, 12.5683], zoom_start=7, tiles="cartodb positron")

# Create a list of heat map data with latitude, longitude and log_cnt
heat_data = [[row['lat'], row['lon'], row['log_count']] for index, row in df.iterrows()]

# Create a heat map layer with red gradient and add it to the map
HeatMap( # We can try different values for these parameters to enhance the visualization
 heat_data, min_opacity=0.4, max_opacity=0.9, radius=5, blur=5,
 gradient={0.4: 'white', 0.5: 'orange', 0.7: 'red'} ).add_to(m)
# Save and display the map
m.save('spatial_heatmap.html')
m
```

The HeatMap function is used to display the data on the map, with adjustable parameters such as opacity, radius, and blur for optimizing visualization. The heat map uses a gradient from white to red, representing low to high densities. In this example, we adjust the heat gradient to a log scale. Depending on the data distribution, we could use different strategies such as equiwidth and equidepth gradients. As illustrated in Fig. 10.18, heat maps highlight spatial patterns in the data and help to identify hotspots of general activity. We can see that the main job for computing the heat map was done by the SQL query stored in the SpatialHeatMap table.

Temporal Heat Map

We now aggregate the grid data along the time dimension. The resulting table shows the number of ship movements occurring at each time interval.

```
CREATE TABLE TimeHeatMap AS
SELECT TileTime, COUNT(*) AS ShipCount
FROM TripTiles
GROUP BY TileTime;
```

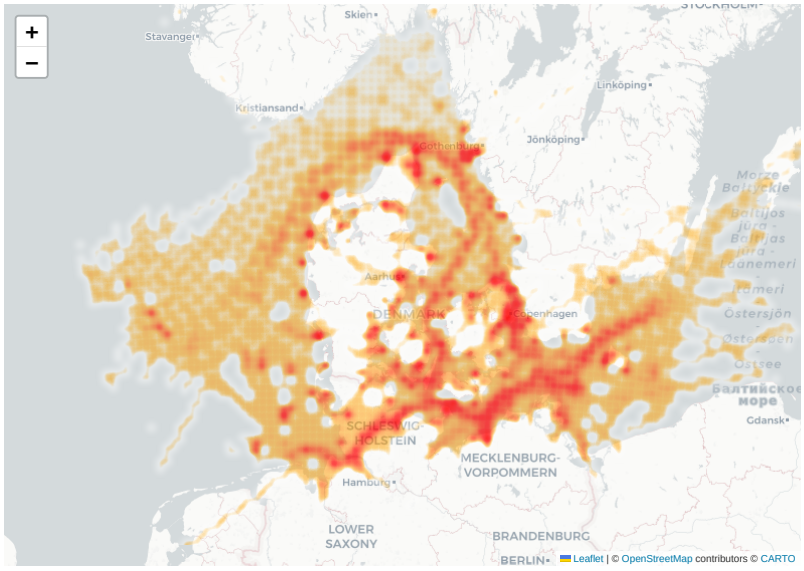


Fig. 10.18 Heat map of the spatial projection of ship trajectories

This temporal heat map is particularly useful for identifying peaks duration and the duration of low activity. We illustrate how to create such visualization using the Seaborn library.

```
# Library imports
import psycopg as pg
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Connect to the database
connection = pg.connect(...)
query = """
  SELECT TileTime, ShipCount FROM TimeHeatMap;
"""
df = pd.read_sql_query(query, connection)
connection.close()

# Format to show only hour (e.g., '15:00')
df['TileTime'] = pd.to_datetime(df['TileTime']).dt.strftime('%H:00')

# Create a heat map using Seaborn
plt.figure(figsize=(10, 6))
sns.heatmap(df.pivot_table(index='TileTime', values='ShipCount', fill_value=0),
 cmap='Blues', annot=True, fmt='.0f', cbar=False)

plt.show()
```

The script is similar to the spatial one, except that a different library is used. The result is shown in Fig. 10.19, where the y-axis shows one-hour intervals, as specified when we applied multidimensional tiling to create the table TripTiles.

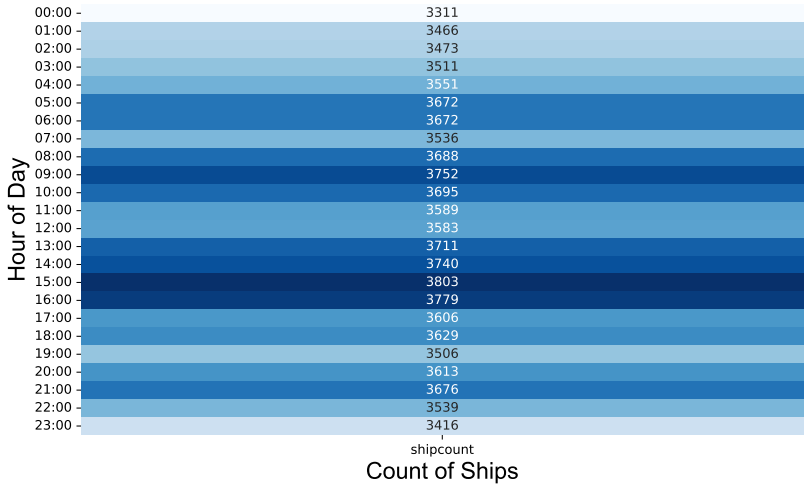


Fig. 10.19 Heat map of the temporal distribution of ships

Spatiotemporal Heat Maps

While the previous heat maps focused on either spatial or temporal aspects of the data separately, the TripTiles table enables spatiotemporal analysis by combining both dimensions. We can consider the TripTiles table as a 3D (2D + time) grid representing a spatiotemporal distribution, which enables us to extend the heat map concept to three dimensions, plotting data points across geographic space and time. However, in many occasions, 3D visualizations result in an overly cluttered representation that is difficult to interpret, making it hard to extract actionable insights. While 3D models for data visualization, such as those discussed in Chap. 4, are helpful in various contexts, applying them to heat maps introduces challenges in clarity and usability.

A more effective approach for combining spatial and temporal dimensions in heat maps is to use a time slider. This allows the analyst to *travel* through time, observing changes in spatial density dynamically. By moving the slider, users can interactively focus on specific time intervals and analyze the evolution over time of the spatial distribution, thus exploring the temporal and the spatial dimensions simultaneously. This approach helps revealing temporal trends without overwhelming the user with too much information at

once, providing an intuitive and insightful visualization. The following script implements these concepts.

```
# Library imports
import psycopg as pg
import pandas as pd
import folium
from folium.plugins import HeatMapWithTime
import numpy as np

# Connect to the database
connection = pg.connect(...)
query = """
  SELECT ST_X(ST_Transform(TileGeom, 4326)) AS lon,
    ST_Y(ST_Transform(TileGeom, 4326)) AS lat,
    COUNT(*) AS ShipCount, TileTime
  FROM TripTiles
  GROUP BY TileGeom, TileTime;
"""
# Apply log transformation to the 'count' column for better visualization
df = pd.read_sql_query(query, connection)
connection.close()

df['log_count'] = df['ShipCount'].apply(lambda x: np.log(x + 1))

# Format time as a string
df['TileTime'] = df['TileTime'].dt.strftime('%Y-%m-%d %H:%M:%S')

# Group data by time and prepare it for HeatMapWithTime
time_indexed_data = []
time_steps = sorted(df['TileTime'].unique()) # Unique time steps sorted
for time in time_steps:
 # Extract data for this specific time step
 subset = df[df['TileTime'] == time]
 heat_data = [[row['lat'], row['lon'], row['log_count']] for index, row in subset.iterrows()]
 time_indexed_data.append(heat_data)

# Create a base map centered on Denmark
m = folium.Map(location=[55.6761, 12.5683], zoom_start=7, tiles="cartodb positron")

# Add HeatMap with TimeSlider functionality
HeatMapWithTime(
 time_indexed_data, index= time_steps,
 gradient={0.2: 'white', 0.6: 'orange', 1.0: 'red'}
).add_to(m)

# Display the map
m
```

The SQL query in the script aggregates the tiles from the TripTiles table using both the spatial and the time dimensions. Once the data are retrieved, the script applies a logarithmic transformation to the ship counts, to normalize the values and enhance the heat map visualization. The core feature of the script is the use of the HeatMapWithTime plugin from the Folium library,

which creates an animated heat map enabling users to interactively explore how the spatial distribution of the ship density changes over time, as illustrated in Fig. 10.20. Each time instant in the data corresponds to a different moment in the animation, and the spatial heat map is updated as the user slides through time. This dynamic visualization allows analyzing temporal trends and patterns, such as peak times for ship traffic or identifying temporal hotspots and provides an intuitive way to understand the evolution of spatial data over time.

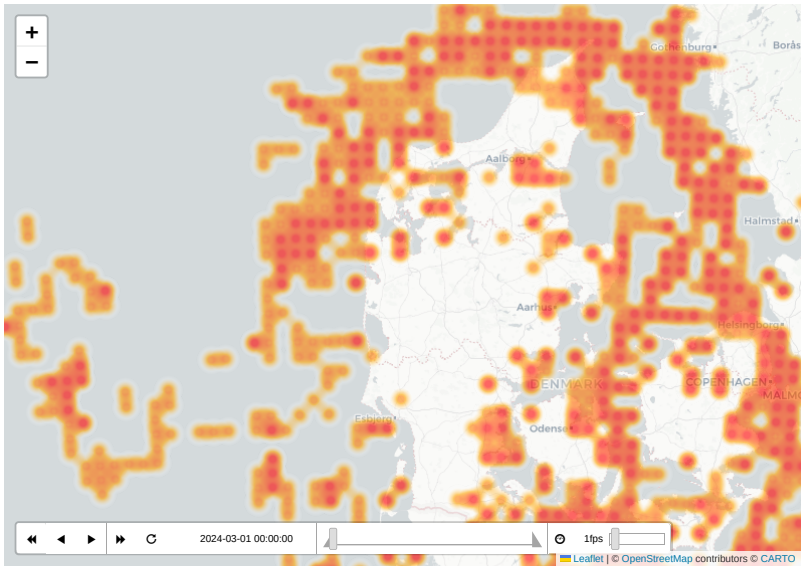


Fig. 10.20 Animated spatiotemporal heat map with a time slider.

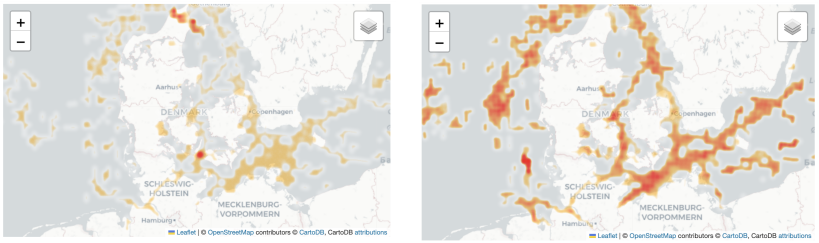
Speed Heat Maps

All the heat maps shown so far are based on the *number of trips*. Alternatively, we can build heat maps based on the *characteristics of the trips* within a tile. As an example, we compute the time-weighted average of the speed of the trips within the tiles with the following query:

```
SELECT ST_X(ST_Transform(TileGeom, 4326)) AS Lon,
 ST_Y(ST_Transform(TileGeom, 4326)) AS Lat,
 AVG(twAvg(speed(Trip))) AS AvgSpeed, TileTime
FROM TripTiles
GROUP BY TileGeom, TileTime;
```

The query uses the Trip column of the TripTiles table, which contains the fragments of the trips in the tiles. Function speed computes the temporal speed of these fragments, and function twAvg computes the time-weighted

average of the temporal speed, resulting in a single speed value for each trip in a tile. Then, function AVG aggregates the speed of all trips within each tile. Figure 10.21 compares the average speed of the ships at midnight and at 4 am, and we can clearly see lower speeds at midnight.



(a) **(b)**

Fig. 10.21 Speed heat maps (**a**) at midnight, and (**b**) at 4 am.

10.3.3 Flow Maps

We conclude this section with a different kind of heat maps. In contrast to the previous ones that focused on visualizing the data distribution in the spatiotemporal space, *flow maps* focus on the movement between points of interest (PoIs), such as between harbors. It helps visualizing the density of traffic between PoIs. As an example, the query below records ship movements between pairs of harbors.

```
CREATE TABLE Flow AS
SELECT a.Id AS HarborA, b.Id AS HarborB, MMSI
FROM Harbors a, Harbors b, Ships s
WHERE a.Id < b.Id AND eIntersects(s.Trip, a.Geom) AND eIntersects(Trip, b.Geom);
```

The query performs a self-join on the Harbors table, followed by a join with the Ships table. The HarborA and HarborB columns represent the identifiers of two distinct harbors involved in the ship traversal. The condition a.Id $<$ b.Id ensures that each pair of harbors is only considered once, avoiding duplication of pairs (i.e., (A, B) and (B, A) are treated as the same route). The eIntersects predicate is used twice in the query to check whether a ship's trajectory intersects both harbors at any point in time.

Table Flow is designed to facilitate the creation of a flow map, where each record represents a movement of a ship between two harbors. By aggregating these data, the flow map can illustrate the frequency of ship movements between various harbor pairs. Further, this query can be extended to record the location of the ship at every harbor, which would then enable spatial-only,

temporal-only, and spatiotemporal analyses as in the previous density heat maps. The query can also be extended to capture the flow direction between the two harbors.

Notice that this query is computationally expensive due to the cross join cardinality. In our dataset there are 75 harbors and 3,112 ships. Accordingly the query will perform 17,505,000 eIntersects predicate evaluations. While defining GiST or SP-GiST indexes on the Trip attribute can improve performance, the speedup may be limited because ship trajectories are often too long and their spatial bounding boxes cover large areas, reducing the effectiveness of index filtering. In general, indexing long trajectories like those in AIS data does not provide substantial performance gains. One way to solve this is to index trajectories using multiple bounding boxes, using extensions of the GiST and SP-GiST index structures like the multi-entry search trees studied in Sect. 6.8.

The following script creates a flow map using the result of the query above.

```
# Library imports
import psycopg as pg
import pandas as pd
import folium
from folium.plugins import AntPath
import random

# Connect to the database
connection = pg.connect(...)

# Query to retrieve distinct ports
query_ports = """
  SELECT Id, ST_X(ST_Transform(ST_Centroid(Geom), 4326)) AS lon,
    ST_Y(ST_Transform(ST_Centroid(Geom), 4326)) AS lat
  FROM Harbors
  WHERE Id IN (SELECT HarborA FROM Flow UNION SELECT HarborB FROM Flow);
"""

# Query to retrieve port-to-port flow
query_flows = """
  SELECT HarborA, HarborB, COUNT(DISTINCT MMSI) AS ShipCount,
    ST_X(ST_Transform(ST_Centroid(a.Geom), 4326)) AS lon_a,
    ST_Y(ST_Transform(ST_Centroid(a.Geom), 4326)) AS lat_a,
    ST_X(ST_Transform(ST_Centroid(b.Geom), 4326)) AS lon_b,
    ST_Y(ST_Transform(ST_Centroid(b.Geom), 4326)) AS lat_b
  FROM Harbors a, Harbors b, Flow f
  WHERE a.Id < b.Id AND a.Id = f.HarborA AND b.Id = f.HarborB
  GROUP BY a.Id, b.Id, lon_a, lat_a, lon_b, lat_b;
"""
# Fetch the ports and flow data
df_ports= pd.read_sql_query(query_ports, connection)
df_flows= pd.read_sql_query(query_flows, connection)

# Close the connection
connection.close()
```

```
# Create a base map
m = folium.Map(location=[56.2639, 9.5018], zoom_start=6, tiles="cartodb positron")
buffer_radius = 2000 # Buffer around harbor

# Function that generates a random color in hex format
def get_random_color():
  return "#{:06x}".format(random.randint(0, 0xFFFFFF))

# Add distinct ports as markers
for index, row in df_ports.iterrows():
 folium.Circle(
  location=[row['lat'], row['lon']],
  radius=buffer_radius, color=get_random_color(),
  fill=True,fill_color=get_random_color(), fill_opacity=0.6
 ).add_to(m)

# Add flow arrows between ports, with size based on ship count
for index, row in df_flows.iterrows():
 # Calculate arrow thickness based on the number of ships
 line_weight = max(1, row['ShipCount'] / 10.0) # Scale down the weight

 # Add an arrow between the two ports
 AntPath( # Fancy line display
  locations=[(row['lat_a'], row['lon_a']), (row['lat_b'], row['lon_b'])],
  color='blue', weight=line_weight, # Thickness represents ship flow
  opacity=0.6).add_to(m)

# Display the map
m
```

This script retrieves two datasets from the database and store them in query_ports and query_flows, respectively. The first query fetches the port coordinates (latitude and longitude) from the Harbors table, ensuring that only ports involved in ship movements (as recorded in the Flow table) are selected. The second query retrieves the number of trips between pairs of ports, along with the geographic coordinates of both ports. The ST_Transform function converts the geometries to the WGS84 coordinate system, as required by the Folium library.

After obtaining the data, the script uses Folium to create an interactive map. Ports are marked with circles, each with a random color, using their latitude and longitude coordinates. The main functionality of the script is to visualize the flow between these ports using the AntPath function, which creates animated arrows between port pairs, with the thickness of the arrows representing the number of ships traveling between them. Figure 10.22 illustrates the result of this script.

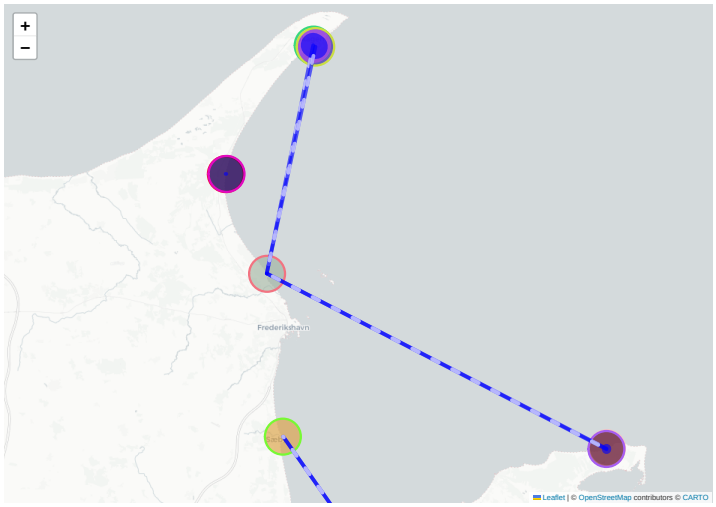


Fig. 10.22 Ship flow between harbors

10.4 Trajectory Similarity Measures

Similarity measures aim at quantifying the extent to which two trajectories resemble each other, comparing their spatial and temporal aspects. Measuring similarity is a fundamental operation for comparing two trajectories. Many analysis tasks include trajectory similarity as their core part, for instance:

- Similarity search, which finds the trajectories most similar to a given one in a collection;
- Clustering, which groups trajectories with similar properties;
- Classification, which identifies trajectories associated with a known set of properties;
- Aggregation and characterization, which identifies representative trajectories and their properties.

Some of the tasks above are covered in this book, and we will further delve into the details of clustering in Sect. 10.5 below.

In practice, measuring the similarity between trajectories is needed in many scenarios. For example, in smart cities and modern transport offerings, comparing the trajectories enables applications such as ride sharing, where there is a need to identify common spatiotemporal patterns. In surveillance systems, identifying abnormal movements through trajectory analysis can enhance security by detecting suspicious or unauthorized behavior. In sports, comparing player trajectories provides insights into performance, strategy, and training efficiency. It also helps in autonomous systems, where a robot or a drone can benefit from stored trajectories of previous missions to optimize current and future missions that involve similar planned paths.

10.4.1 Classification of Similarity Measures

Trajectory similarity measures can be characterized along three dimensions.

First, similarity measures may be *metric* or not. Given two trajectories A, B, and C, a distance measure d is metric if it satisfies the following properties:

- Identity: $d(A, B) = 0$ when $A = B$;
- Symmetry: $d(A, B) = d(B, A)$;
- Triangle inequality: $d(A, B) + d(B, C) \leq d(A, C)$.

Not all distance measures are metric: for example, travel time in transportation networks is a distance measure that is typically not symmetric. Similarly, not all similarity measures are metrics.

Second, similarity measures may be *discrete* or *continuous*. Discrete measures use only the locations at certain times, ignoring the movement in between. Continuous measures require interpolation between locations measured at a discrete set of times. Most similarity measures consider only a discrete subset of points in the trajectory, typically the measured data points.

Finally, similarity measures may be *relative* or *absolute* depending on how space and time are considered. If we consider space and/or time as absolute, similarity is computed with respect to a given space and/or time frame. Otherwise, the comparison ignores absolute positions or times. Three cases may arise here:

- Absolute time and space: when measures require that two trajectories occur in approximately the same space at the same time. This is a very restrictive situation, since typically trajectory similarity requires that measures operate in relative time, space, or both.
- Relative time: when measures prioritize space over time so that trajectories can be considered similar regardless the time where they occur. In this case, trajectories can aligned through temporal transformations.
- Relative space: when measures prioritize time over space so that trajectories can be considered similar regardless the geographical space where they occur. In this case, trajectories can aligned through spatial transformations.

Since the Euclidean distance is the basis of distance metrics used in many applications, it can also be used to provide a simple computation of similarity. Given two trajectories $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, the idea is to compute the Euclidean distance between corresponding locations, that is, between a_1 and b_1, a_2 and b_2, and so on. Starting from this simple measure, many other trajectory measures have been developed. In this section, we study three commonly used trajectory similarity measures, that is, dynamic time warping (DTW), discrete Fréchet distance (DFD), and time warp edit distance (TWED). For illustrative purposes, we will compare them with the simple trajectories shown in Fig. 10.23. In Sect. 10.4.5 we will discuss each measure with respect to the dimensions studied in this section.

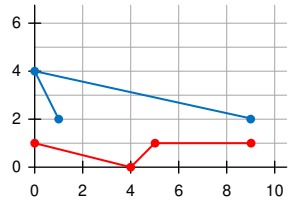


Fig. 10.23 Simple trajectories used for analyzing the various similarity measures.

10.4.2 Dynamic Time Warping

Dynamic time warping (DTW) is an algorithm for measuring similarity between two temporal sequences which may vary in speed. DTW was originally used for speech recognition, then it was generalized for time series and later, it became one of the commonly methods for measuring similarity between trajectories. The DTW recursive formula given below defines the optimal alignment between two sequences A and B by minimizing the cumulative distance between their elements.

$$DTW(i,j) = \begin{cases} 0, \text{ if } i = 0 \text{ and } j = 0, \\ \infty, \text{ if } i = 0 \text{ or } j = 0 \text{ (but not both)}, \\ d(i,j) + \min \begin{pmatrix} DTW(i-1,j-1), \\ DTW(i-1,j), \\ DTW(i,j-1) \end{pmatrix}, \text{ otherwise.} \end{cases}$$

When both sequences are empty ($i = 0$ and $j = 0$), the distance is zero since there are no elements to align. However, when one sequence is empty and the other is not ($i = 0$ or $j = 0$), the distance is set to infinity (∞) because it is impossible to align an empty sequence with a non-empty one. In other cases, the DTW distance between two points i and j is computed as the sum of the distance $d(i,j)$ between the elements being compared and the minimum of three possible alignments: (1) matching the elements from both sequences, (2) skipping the current element of sequence A, or (3) skipping the current element of sequence B. This recursive process continues, accumulating the minimum cost alignment for the entire sequences. This flexibility in choosing the alignment at every step allows DTW to handle misaligned sequences by stretching or compressing portions of the sequences to achieve an optimal alignment. Figure 10.24a shows the Euclidean distance commented above together with the DTW in Fig. 10.24b. As can be seen, DTW allows time shifts by stretching or compressing portions of the sequences to align them.

The DTW can be efficiently calculated using dynamic programming. The algorithm builds a cost matrix using the following steps:

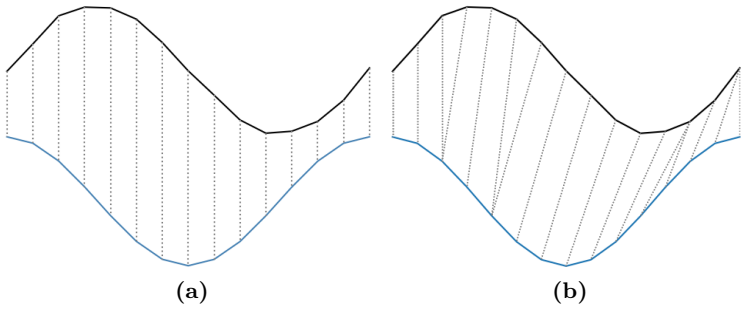


Fig. 10.24 Euclidean distance and dynamic time warping (DTW): (**a**) Euclidean distance requires sequences to be compared point by point; (**b**) DTW accommodates variations in time by allowing one sequence to match a point with several points from the other sequence.

1. *Initialization:*

   $$DTW(0,0) = 0,$$
   $$DTW(i,0) = \infty \text{ for } i > 0,$$
   $$DTW(0,j) = \infty \text{ for } j > 0$$

   The top-left corner of the matrix is initialized to 0, representing no cost for aligning two empty sequences. The first row and column are initialized to infinity (∞), since aligning a sequence with an empty sequence incurs an infinite cost.
2. *Recurrence relation:* For each element in the matrix, the cost is computed between the two elements a_i and b_j as the sum of $d(a_i, b_j)$ and the minimum among the right, diagonal, and top cells in the matrix, i.e., as indicated in the recursive expression.
3. *Termination:* The DTW distance is found in the bottom-right corner of the matrix $DTW(m,n)$, which gives the minimal cumulative cost for aligning the full sequences A and B.

The following script illustrates the computation of DTW distance between the two trajectories in Fig. 10.23, and the corresponding dynamic programming cost matrix.

```
# Library imports
import numpy as np

# Populate the DTW matrix
def populate_dtw_matrix(matrix, trajectoryA, trajectoryB):
  m, n = len(trajectoryA), len(trajectoryB)
  for i in range(1, m+1):
    for j in range(1, n+1):
      cost = euclidean_distance(trajectoryA[i-1], trajectoryB[j-1])
      matrix[i, j] = cost + min(
```

```
            matrix[i-1, j-1], # match
            matrix[i-1, j], # deletion from A
            matrix[i, j-1]) # deletion from B
    return matrix

# Define the points for the two trajectories
trajectoryA = [(0, 1), (4, 0), (5, 1), (9, 1)]
trajectoryB = [(1, 2), (0, 4), (9, 2)]

# Function that calculates the Euclidean distance between two points
def euclidean_distance(p1, p2):
    return np.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

# Initialize DTW matrix with infinity
m, n = len(trajectoryA), len(trajectoryB)
dtw_matrix = np.full((m+1, n+1), np.inf)
dtw_matrix[0, 0] = 0

# Populate the DTW matrix
dtw_matrix = populate_dtw_matrix(dtw_matrix, trajectoryA, trajectoryB)

np.set_printoptions(precision=2, suppress=True)
print(dtw_matrix)
```

The DTW dynamic programming algorithm aims at finding the optimal alignment between two sequences by minimizing the cumulative distance between their elements using a cost matrix. From a visual perspective, DTW operates by walking along the alignment path from the upper-left to the lower-right corner of a cost matrix. At each step, DTW considers three possible moves: horizontal, vertical, or diagonal. The diagonal move corresponds to matching the current pair of points from both sequences, with a cost equal to the distance between the points. The vertical move involves skipping a point in sequence A and aligning it with the current point in sequence B, again with a cost equal to the distance between the points. Similarly, the horizontal move skips a point in sequence B and aligns it with the current point in sequence A, incurring the same cost. In each case, the algorithm selects the path that minimizes the total cumulative distance. Typical choices for the distance $d(a_i, b_j)$ are the Euclidean distance, and the squared distance. In the script above, a helper function `euclidean_distance` is used to express $d(a_i, b_j)$, and may be replaced by other cost measures. The DTW matrix for the two trajectories given in Fig. 10.23 is illustrated next.

$$\begin{bmatrix} 0 & \infty & \infty & \infty \\ \infty & \mathbf{1.41} & 4.41 & 13.47 \\ \infty & \mathbf{5.02} & 7.07 & 9.8 \\ \infty & 9.14 & \mathbf{10.85} & 11.19 \\ \infty & 17.21 & 18.63 & \mathbf{11.85} \end{bmatrix}$$

The bottom-right cell shows the DTW distance value. The bold cells illustrates the optimal alignment between the points of the two trajectories, which results in the minimal total distance.

For each cell in the matrix, the computation takes constant time, as it depends only on the costs of the adjacent cells: the one to the left, top, and diagonally left-top. This is because, for each alignment, DTW evaluates three possible paths and chooses the one with the minimum cumulative cost. As a result, the overall time complexity is quadratic, or $O(mn)$, where m and n are the lengths of the two sequences being aligned. This quadratic complexity arises because the algorithm must fill an entire $m \times n$ matrix, with each entry requiring constant time to compute. Although DTW is computationally more intensive than simpler distance measures like Euclidean distance, which operates in linear time, its ability to handle misaligned sequences and time shifts justifies the increased complexity. In some practical cases, this quadratic complexity can be reduced by pruning unnecessary parts of the matrix or using approximation techniques, but the basic version of DTW remains $O(mn)$.

The following query illustrates the DTW distance in MobilityDB. The query uses the dynTimeWarpDistance function to repeat the above example. The result is the same as in the Python script, a distance value of approximately 11.85. Notice that the timestamps of the two trajectories are ignored, and they are treated as sequences of points.

```
SELECT dynTimeWarpDistance(
 tgeompoint '[Point(0 1)@2001-01-01, Point(4 0)@2001-01-03, Point(5 1)@2001-01-05,
   Point(9 1)@2001-01-07]',
 tgeompoint '[Point(1 2)@2001-01-01, Point(0 4)@2001-01-03, Point(9 2)@2001-01-05]');
-- 11.850716732682386
```

With respect to the Python script above, MobilityDB applies the following optimizations: (1) the first row and column initialized to 0 or ∞ are not stored, and (2) only two lines of the $m \times n$ matrix are needed, since for computing a row i of the matrix only the values of the row $i-1$ are required. This is important when we have trajectories with a large number of instants. For example, some statistics about the number of instants in our AIS dataset are as follows.

```
SELECT MIN(numInstants(Trip)), MAX(numInstants(Trip)), AVG(numInstants(Trip))
FROM Ships;
-- 1 | 27556 | 2809.6597043701799486
```

10.4.3 Discrete Fréchet Distance

The discrete Fréchet distance (DFD), like DTW, is a measure of similarity that takes into account both the location and the ordering of points along two curves. Fréchet distance has both discrete and continuous forms. In this

section, we focus on the discrete variant, which is referred to as Fréchet distance from now on.

The main difference between DTW and the Fréchet distance lies in the nature of the path taken to align the two trajectories. The Fréchet distance seeks the optimal continuous path between two trajectories under the constraint that the movement along the two curves must be continuous and non-reverting. In other words, both curves must be traversed in the same direction. This difference results in a slightly stricter form of alignment, where large deviations are less tolerated compared to DTW. Mathematically, the Fréchet distance between two curves A and B is the minimum length required to connect a point on A to a point on B, ensuring that both are traversed from start to end. The Fréchet distance can be visualized as the minimum length of a leash needed for a person and their dog to walk along two separate paths, where both must move continuously forward along their own path.

DFD can be seen as a version of DTW that takes the maximum distance between aligned points along the path. This is in contrast to DTW, which considers the sum of all alignment costs. The recursive definition of the Fréchet distance is thus similar to DTW, as given by:

$$F(i,j) = \begin{cases} d(a_1, b_1), & \text{if } i = 1 \text{ and } j = 1, \\ \max(d(a_i, b_j), \min(F(i-1,j), F(i,j-1), F(i-1,j-1))), & \text{otherwise.} \end{cases}$$

To compute the Fréchet distance, we only need to modify the DTW Python code above by changing the matrix population function as follows:

```
# Populate the Frechet matrix
def populate_dfd_matrix(matrix, trajectoryA, trajectoryB):
  m, n = len(trajectoryA), len(trajectoryB)
  for i in range(1, m+1):
    for j in range(1, n+1):
      cost = euclidean_distance(trajectoryA[i-1], trajectoryB[j-1])
      matrix[i, j] = max(cost, min(
        matrix[i-1, j-1], # match
        matrix[i-1, j], # deletion from A
        matrix[i, j-1])) # deletion from B
 return matrix
```

The Fréchet matrix for the trajectories in Fig. 10.23 is shown below.

$$\begin{bmatrix} 0 & \infty & \infty & \infty \\ \infty & \mathbf{1.41} & \mathbf{3.0} & 9.06 \\ \infty & 3.61 & 5.66 & \mathbf{5.39} \\ \infty & 4.12 & 5.83 & \mathbf{5.39} \\ \infty & 8.06 & 9.49 & \mathbf{5.39} \end{bmatrix}$$

The bottom-right cell shows the Fréchet distance value, which is the leash length needed to achieve this walk. The bold cells illustrate *one* possible alignment that preserves this leash length.

Repeating this example in MobilityDB is illustrated as follows:

```
SELECT frechetDistance(
 tgeompoint '[Point(0 1)@2001-01-01, Point(4 0)@2001-01-03, Point(5 1)@2001-01-05,
   Point(9 1)@2001-01-07]',
 tgeompoint '[Point(1 2)@2001-01-01, Point(0 4)@2001-01-03, Point(9 2)@2001-01-05]');
--5.385164807134504
```

With respect to the Python script above, the MobilityDB implementation applies the same optimizations as explained for the DTW above.

10.4.4 Time Warp Edit Distance

Time Warp Edit Distance (TWED) is a similarity measure for trajectories that combines spatial distances with penalties for temporal shifts. Unlike DTW and DFD, TWED introduces two additional parameters: the penalty parameter λ, which discourages large gaps between aligned points, and the elasticity parameter ν, which controls the flexibility of the alignment. TWED is particularly suitable for applications where both spatial and temporal alignment are crucial, since it penalizes not only the Euclidean distance between points (as is the case for both DTW and DFD), but also the time gaps between them. This results in a measure that is robust to differences in sampling rates and varying speeds along the trajectories.

TWED has a similar recursive structure to that of DTW and DFD. It finds the sequence of edit operations with minimal cost that transforms two time series into aligned paths. The formula for TWED is as follows:

$$TWED(i,j) = \begin{cases} 0, \text{ if } i = 0 \text{ and } j = 0, \\ \infty, \text{ if } i = 0 \text{ or } j = 0 \text{ (but not both)}, \\ d(i,j) + \min \begin{pmatrix} TWED(i-1,j-1) + \lambda, \\ TWED(i-1,j) + \nu + \lambda, \\ TWED(i,j-1) + \nu + \lambda \end{pmatrix}, \text{ otherwise.} \end{cases}$$

As can be seen, the recursive definition extends that of DTW by adding the two penalty parameters: λ the penalty for deleting or inserting a point, and ν the elasticity parameter. These penalties allow TWED to balance spatial and temporal alignment, whereas DTW only minimizes cumulative spatial alignment without regard for temporal continuity.

The Python script below implements the TWED using both the penalty and elasticity parameters to compute the alignment cost. The cost is similar to DTW, except for the matrix population, which is given next:

```
# Populate the TWED matrix
def populate_twed_matrix(matrix, trajectoryA, trajectoryB, lambda_penalty,
    nu_elasticity):
  m, n = len(trajectoryA), len(trajectoryB)
```

```
  for i in range(1, m+1):
    for j in range(1, n+1):
      cost = euclidean_distance(trajectoryA[i-1], trajectoryB[j-1])
      matrix[i, j] = cost + min(
        matrix[i-1, j-1] + lambda_penalty, # match with penalty
        matrix[i-1, j] + lambda_penalty + nu_elasticity, # deletion from A
        matrix[i, j-1] + lambda_penalty + nu_elasticity # deletion from B
      )
  return matrix
```

The TWED matrix for the trajectories in Fig. 10.23 using lambda_penalty = 0.5 and nu_elasticity = 1 is shown next,

$$\begin{bmatrix} 0 & \infty & \infty & \infty \\ \infty & \mathbf{1.91} & 6.41 & 16.97 \\ \infty & 7.02 & \mathbf{8.07} & 12.3 \\ \infty & 12.64 & 13.35 & \mathbf{12.69} \\ \infty & 22.21 & 22.63 & \mathbf{14.85} \end{bmatrix}$$

where the path that minimizes the distance is highlighted in bold.

10.4.5 Comparing Similarity Measures

We conclude this section comparing the three measures studied above, namely, dynamic time warping (DTW), discrete Fréchet distance (DFD), and time warp edit distance (TWED) with respect to the dimensions in Sect. 10.4.1.

Both the DFD and the TWED measures are also metrics, while the DTW does not obey the triangle inequality.

The three measures are discrete, while continuous versions of DTW and Fréchet distance have been defined.

The three measures support relative time, in the sense that the definition of each measure relies on the absolute spatial distance between ordered points in the trajectory, rather than the absolute time gap between points.

With respect to the relative versus absolute space dimension, the three measures align trajectories in time to minimize absolute Euclidean distances. However, depending on the application, the relative distance may be relevant. A basic approach to address this problem would be to optimize the measures under a suitable set of spatial transformations, for example translations.

With respect to the relative versus absolute time dimension, the three measures temporally align the two trajectories being compared, aggregating the cost of this alignment between each pair of points. The key differences between measures lie in the details of how this is done. For instance, DTW and DFD differ on whether they take the sum (DTW) or the maximum (DFD) of the local costs.

As a final observation, with respect to tolerance to outliers, in general, measures that use the maximum distance between matched points (such as DFD) emphasize large distances and are therefore more sensitive to outliers than measures that use some aggregation of distances, when thresholds can be useful for dealing with outliers.

10.5 Clustering

Clustering is a technique that organizes a set of objects in groups, called **clusters**, such that objects in the same group are more similar to each other than objects in different groups. In machine learning, this method belongs to the family of *unsupervised learning* algorithms, since, opposite to *supervised learning* algorithms, we do not know in advance the class an object belongs to. Many types of clustering methods exist in the data mining and machine learning literature. A common classification identifies three main types: partition-based, hierarchical, and density-based clustering.

Partition-based clustering decomposes a dataset into a set of disjoint clusters. Given a dataset of N points, a partitioning method constructs k partitions of the data, $k \leq N$, where each partition represents a cluster. Each group must contain at least one point and each point must belong to exactly one group. Many partition-based clustering algorithms try to minimize an objective function, for example, a distance function like the ones studied in the previous section. In this section, we use the *K-means* algorithm, where the parameter k is given by the user. It starts from a random partitioning of the objects and then performs several iterations to progressively refine it. During an iteration, K-means first computes a centroid for each cluster, i.e., a new representative object computed from those currently in the cluster. Then, the algorithm re-assigns each object to the centroid that is closest to it in terms of the distance function used, and the process stops when convergence up to a certain threshold is reached. We refer the reader to the literature for details, in this section we will focus on the usage of the algorithm.

Hierarchical clustering aims at building a hierarchy of clusters. There are two classes of hierarchical clustering. *Agglomerative clustering* uses a bottom-up approach where each observation starts in its own cluster, and pairs of clusters are merged while moving up in the hierarchy. *Divisive clustering* use a top-down approach where all observations start in one cluster, and splits are performed recursively while moving down along the hierarchy. In this section, we focus on agglomerative clustering, which is the most common type of hierarchical clustering. The algorithm first computes the similarity information between every pair of objects in the dataset and then uses a so-called **linkage function** to group objects into a hierarchical cluster tree, based on the distance information generated in the previous step. Thus, clusters that are in close proximity are linked together using such function.

In this way, pairs of clusters are recursively grouped based on their similarity, until all clusters have been merged into one big cluster containing all objects. The result is a tree-based representation of the objects, called **dendrogram**.

The most common linkage methods are described below.

- *Maximum or complete linkage*, where the distance between two clusters is the maximum value of all pairwise distances between the elements in two different clusters. This is also known as the nearest point algorithm.
- *Minimum or single linkage*, where the distance between two clusters is the minimum value of all pairwise distances between the elements in two different clusters. This is also known as the farthest point algorithm or the Voor Hees algorithm.
- *Mean or average linkage*, where the distance between two clusters is the average value of all pairwise distances between the elements in two different clusters.
- *Centroid linkage*, where the distance between two clusters is the distance between the centroids of every pair of clusters.
- *Ward's minimum variance linkage*, which minimizes the total within-cluster variance by merging the clusters with minimum between-cluster distance. This is also known as the incremental algorithm.

Density-based clustering aims at forming maximal, dense groups of objects, thus not limiting the cluster extension or its shape, where objects that cannot be linked to any cluster are labeled as noise and removed. A classic example of this kind of algorithms is DBSCAN, standing for Density-based Spatial Clustering of Applications with Noise. Given a set of points in some space, the algorithm groups together points that are closely packed, that is, with many nearby neighbors, and marks as outliers points that lie alone in low-density regions, that is, points whose nearest neighbors are too far away. It receives as parameters a threshold distance and a minimum number of objects in a cluster.

The choice of an appropriate clustering method must consider the characteristics of the data and the expected characteristics of the output. If we expect compact clusters, K-means can be used, especially for large datasets. The drawback of the method is that the user must have in advance, an idea of the number of desired clusters, which is not required by agglomerative algorithms, since the dendrograms synthesize the results that can be obtained for all possible values of k. However, hierarchical clustering is usually computationally expensive. Finally, density-based methods do not suffer of any of the issues mentioned above, and are also more robust to noisy data.

Clustering methods can be applied to any data type, provided that a notion of similarity or distance between objects is given. In this section, we show how the clustering methods explained above can be applied to spatial and spatiotemporal data using the similarity measures and distance functions explained in Sect. 10.4. We take as use case the NYC Citi Bike dataset used in Sect. 8.9. We start in Sect. 10.5.1 with the K-means and DBSCAN clustering

algorithms provided by PostGIS. In Sect. 10.5.2 we repeat the same analysis using Python, given its popularity within the data science community. In Sect. 10.5.3 we show an example of the application of agglomerative clustering and dendrograms for performing temporal clustering. Finally, we use K-means for spatiotemporal clustering in Sect. 10.5.4. We study trajectory clustering in Sect. 10.6.

10.5.1 Spatial Clustering with PostGIS

PostGIS has four window clustering functions, which receive geometries as parameters and return cluster numbers or NULL when a cluster cannot be assigned to an object. In this section, we focus on the most used ones, namely, K-means and DBSCAN, using the NYC Citi Bike dataset introduced in Sect. 8.9. We use the pickups and returns of bicycles from and to stations in NYC in the month of March, 2024, and the station locations at that time. Data are loaded from the Citi Bike website[5] into a CitibikeInput table with the following structure.

```
CREATE TABLE CitibikeInput(ride_id text, rideable_type text, started_at timestamptz,
  ended_at timestamptz, start_station_name text, start_station_id text,
  end_station_name text, end_station_id text, start_lat float, start_lng float,
  end_lat float, end_lng float, member_casual text, start_point geometry(Point),
  end_point geometry(Point));
```

From this table, we create two tables, described below.

```
CREATE TABLE station_information(station_id text, station_name text,
  station_lat float, station_lng float, geom geometry(Point));
CREATE TABLE station_flow(station_id text, start_time timestamptz, in_count float,
 out_count float);
```

Table station_information contains information about the docking stations, namely, the identifier, name, and geographic location. Table station_flow is populated with a script which can be found in the companion GitHub repository. The start_time attribute is obtained dividing the time span of the database (one month) into bins of thirty minutes each and taking the start bound of each bin. The bins are created as follows.

```
CREATE TABLE time_bins(start_time, stoptime) AS
SELECT h, h + interval '30 minutes'
FROM generate_series(timestamptz '2024-03-01 00:00:00',
  timestamptz '2024-03-31 23:30:00', interval '30 minutes') AS h;
```

Attributes in_count and out_count compute the total number the returns to and pickups from each station, respectively, within each time bin.

We will analyze the spatial distribution of the stations using the PostGIS functions ST_ClusterKMeans and ST_ClusterDBSCAN. We start with K-means clustering as follows.

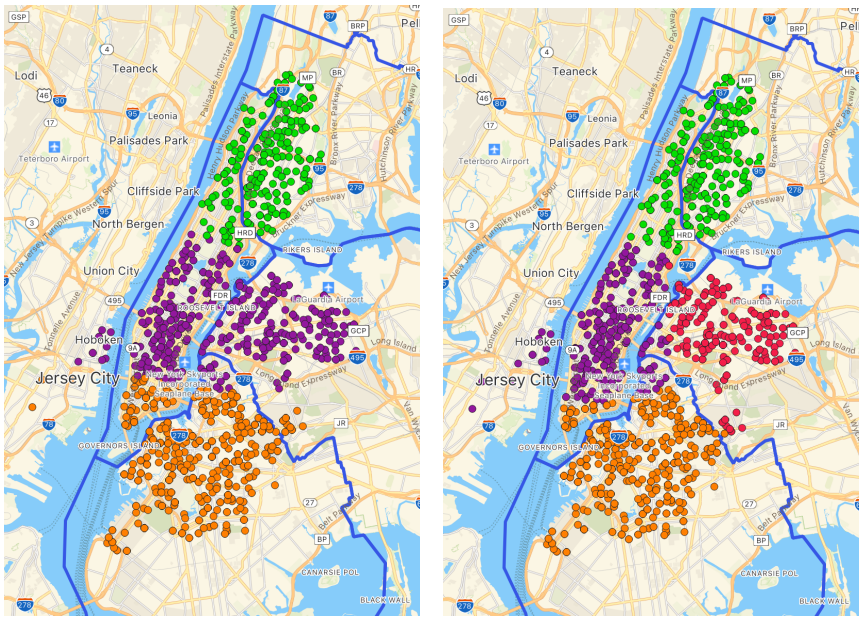
[5] https://s3.amazonaws.com/tripdata/index.html

```
CREATE TABLE station_clusters_kmeans3 AS
SELECT ST_ClusterKMeans(geom, 3) OVER () AS cluster_id, station_id, geom
FROM station_information;
```

The query above creates three clusters ($k = 3$) based on the distance between the stations. Similarly, we run the same algorithm with $k = 4$. The results are shown in Figs. 10.25a and 10.25b, respectively. There are three clusters in Fig. 10.25a indicated in green, orange, and purple. The figure also shows the boundary lines corresponding to four of the five NYC boroughs: the island of Manhattan, The Bronx in the north, Queens in the east, and Brooklyn in the southeast. A detailed look at the map reveals that the stations in the green cluster are the ones that are farther apart from each other. The stations that are closest to each other are the purple ones, which makes sense (and coincides with the results in Sect. 8.9), since this is the midtown Manhattan zone which attracts the highest tourist activity. Finally, the distance between the stations in the orange cluster (which correspond to stations in the south of Manhattan and Brooklyn), lies in-between the other two ones. We can suspect, however, that thee clusters is not enough to split the stations according to the distance between them, since the purple clusters extend outside Manhattan into Queens (and even some of them into Brooklyn). Figure 10.25b solves this problem. We can now see that the purple cluster extends a little bit to the south of Manhattan, indicating that the distance between stations is more similar to the ones in midtown. Importantly, this cluster has been split into two, and the red clusters are clearly located in the borough of Queens, which now allows us to distinguish the stations in midtown Manhattan from the ones in Queens. It becomes clearer now that stations in midtown Manhattan (and also some stations outside the borough but close to the bridges) are closer to each other than the stations in Queens.

We can also use the K-means function in PostGIS to give weight to the objects. The algorithm uses this weight when deciding which cluster an object is assigned to. This is done by adding an additional information in a dimension denoted 'M' which acts as a third or fourth coordinate of the input points. In our case, we use the number of bike pickups at each stop as the weight variable mvalue, which must always be positive. The following query performs weighted K-means clustering.

```
CREATE TABLE station_clusters_kmeans_4_weighted AS
WITH station_agg (station_id, incount, outcount) AS (
  SELECT station_id, SUM(in_count), SUM(out_count)
  FROM station_flow
  GROUP BY station_id ),
stations_information_geom AS (
  SELECT s.*, sa.incount, sa.outcount
  FROM station_information s, station_agg sa
  WHERE s.station_id = sa.station_id )
SELECT *,
  ST_ClusterKMeans(ST_Force4D(Geom, mvalue:=incount), 4) OVER () AS ClusterId
FROM stations_information_geom
WHERE incount > 0;
```



(a) **(b)**

Fig. 10.25 Spatial clustering of NYC bike stations using K-means with (**a**) $k = 3$ and (**b**) $k = 4$. The borough boundaries are indicated in blue lines.

```
CREATE TABLE station_clusters_kmeans_3_weighted AS
WITH station_agg (station_id, incount, outcount) AS (
  SELECT station_id, SUM(in_count), SUM(out_count)
  FROM station_flow
  GROUP BY station_id ),
stations_information_cnt AS (
  SELECT s.*, sa.incount, sa.outcount
  FROM station_information s, station_agg sa
  WHERE s.station_id = sa.station_id )
SELECT *,
  ST_ClusterKMeans(ST_Force4D(Geom, mvalue:=incount), 3) OVER () AS cluster_id
FROM stations_information_cnt
WHERE incount > 0;
```

The query above creates three clusters ($k = 3$) based on the distance between the stations, weighted by the total count of pickups per station during the month under analysis. Table Station_Agg computes the aggregation of pickups per station, which is used to extend the table Station_information, resulting in table Station_information_cnt. Finally, the clustering window function is executed. Similarly, we run the same algorithm with $k = 4$. The results are shown in Figs. 10.26a and 10.26b, respectively. We can clearly see the

influence of the usage pattern in the cluster distribution. In Fig. 10.26a, the green stations in Queens indicate that the use of stations is similar to the use in North Manhattan and The Bronx, also indicated in green. In Fig. 10.25a these stations belonged to the purple clusters, since only the distance between stations was considered. The same pattern can be detected comparing Figs. 10.25b and 10.26b. In the latter we can see that in the north of Brooklyn the usage pattern is similar to the one in Queens, since stations there belong to the red cluster, and were classified as orange when the usage was not accounted for.

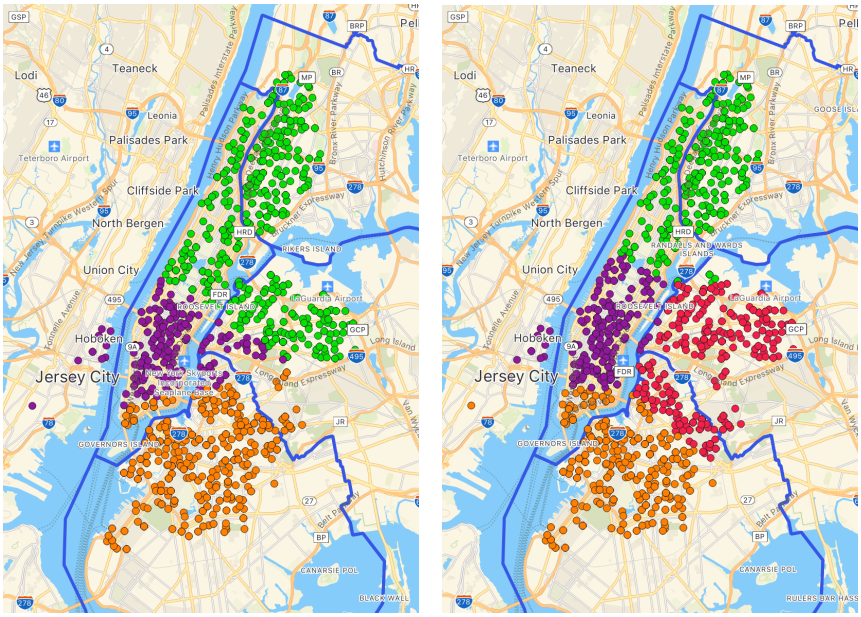


Fig. 10.26 Spatial clustering NYC bike stations using the *weighted* K-means algorithm with (**a**) $k = 3$ and (**b**) $k = 4$.

The ST_ClusterDBSCAN function implements density-based clustering as a window function that takes three parameters: an eps distance tolerance, such that geometries (in our case, stations) must be within this distance to be added to a cluster; a minpoints count, such that if a station is within the eps distance of minpoints cluster members, it is considered a core member of the cluster. A station will be added to a cluster if it is either within eps distance of at least minpoints input geometries (including itself), or a border geometry that is within eps distance of a core geometry.

We will analyze the distribution of the stations based on their closeness to identify zones where stations are sparse, showing also the impact of the eps distance in the result. We use a value 10 for minpoints and two alternative distances, 500 meters and 750 meters, as in the following example:

```
CREATE TABLE station_clusters_dbscan_500_10 AS
SELECT ST_ClusterDBSCAN(geom, 500, 10) OVER () AS cluster_id, station_id, geom
FROM station_information;
```

The result of the execution is a table with a column cluster_id which is the cluster identifier assigned to the station. Figure 10.27 shows the result graphically, illustrating the impact of the eps distance parameter. In Fig. 10.27a we can see that using a distance of 500 m we obtained fifty-five clusters. This does not give us much information about the density of stations since the number of clusters is clearly very high. On the other hand, in Fig. 10.27b we see that using eps = 750 we obtain nine clusters. This looks better and more informative than the previous distance. Compared against the result obtained using the K-means algorithm, we see that we can now distinguish different densities to the east and west of Central Park (the red and purple clusters in Fig. 10.27b, respectively). However, if is also likely that, running K-means with a higher value of k would probably produce the same result. More insight can be obtained exploring different combinations of distance and minimum number of points, which is beyond the goal of this section.

10.5.2 Spatial Clustering with Python

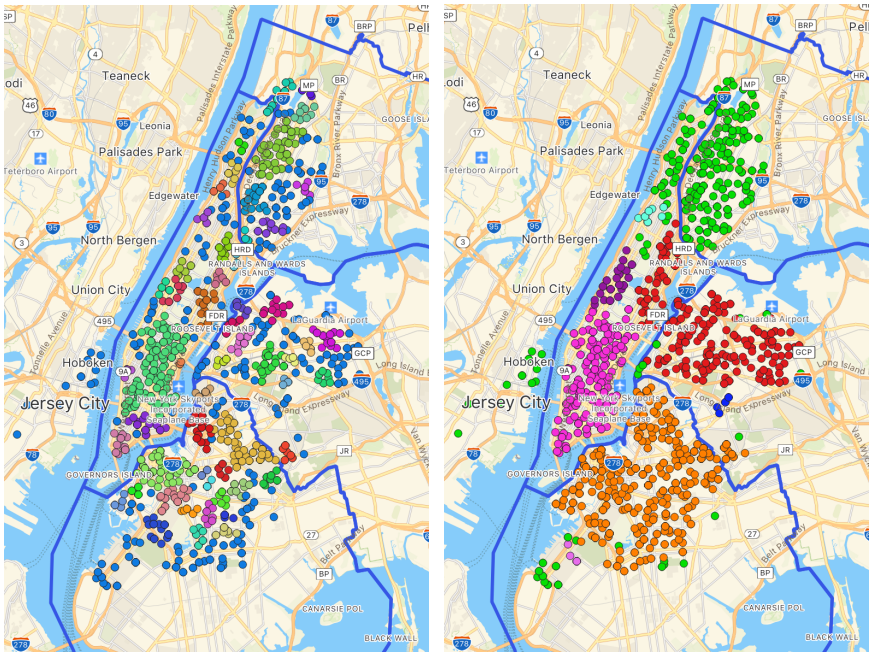
We now show how clustering analysis is applied over the NYC Citi Bike dataset using Python.[6] The analysis is mainly performed using the Python library scikit-learn.[7] The Python code described in this section is available in the companion GitHub repository. We load the data from tables station_information and station_flow into two DataFrames called dfStInf and dfStFlow, respectively. We drop the geom column from dfStInf since we will not use it in this section. For simplicity, we will only keep the first week of the data. This DataFrame, called dfFlow, is obtained as follows.

```
dfFlow = dfStFlow[(dfStFlow['start_time'].dt.month == 3) &
 (dfStFlow['start_time'].dt.day <= 7)]
```

We start with the K-means algorithm using the sklearn.cluster.KMeans function. A crucial step in this regard is the selection of a good k value. We may think that increasing the number of clusters will naturally improve the fit. However, at some point, increasing k does not provide a significant improvement, and this is called *overfitting*. To avoid this, the so-called "elbow"

[6] This section is based on the blog by Chih-Ling Hsu https://chih-ling-hsu.github.io/2018/01/02/clustering-python.

[7] https://scikit-learn.org/



(a) **(b)**

Fig. 10.27 Spatial clustering of NYC bike stations using the DBSCAN algorithm with a minimum of 10 stations and a threshold of (**a**) 500 m and (**b**) 750 m.

method is typically used. The method displays the percentage of explained variance, computed as the ratio of the between-group variance to the total variance, as a function of the number of clusters. We then pick the elbow of the curve as the number of clusters to use. The meaning is that we choose a number of clusters such that adding another cluster does not improve the model in a significant way. The underlying idea is that the first clusters add the major part of the information, but for a larger number of clusters, the added information will drop sharply, and therefore a sharp elbow will appear in the graph of explained variance versus clusters. The elbow curve is computed in Python applying K-means iteratively as described below.

The first step of the process consists in building an array of values, obtained from the values in the DataFrame dfStInf as follows.

```
X = dfStInf[['station_lat', 'station_lng']].values
```

The array X contains [latitude, longitude] tuples.

```
Ks = range(1, 10)
kmean = [KMeans(n_clusters=i).fit(X) for i in Ks]
```

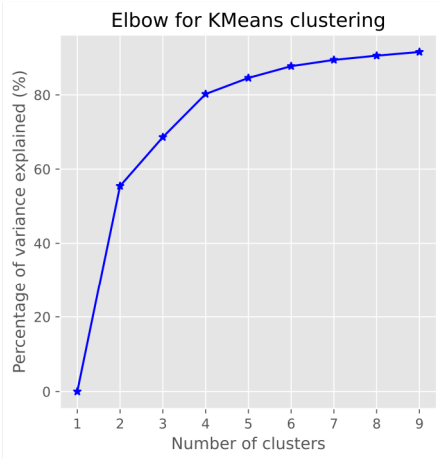


Fig. 10.28 The elbow curve for the NYC Citi Bike dataset.

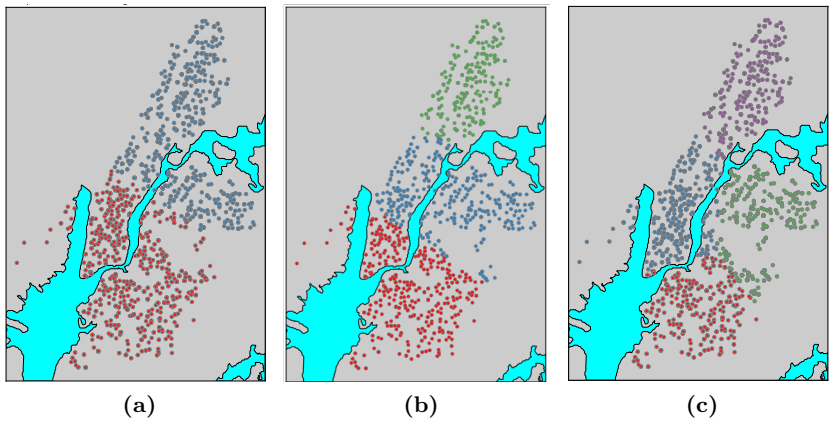


Fig. 10.29 Spatial K-means clustering with (**a**) $k = 2$, (**b**) $k = 3$, and (**c**) $k = 4$.

Figure 10.28 shows the elbow curve for our case study. Since the elbow occurs for $k = 4$, we run the K-means algorithm for $k = 2$, 3, and 4. The results are shown in Fig. 10.29. We can see that they are quite similar to the results depicted in Fig. 10.25. The difference is possibly due to the use of the projections when using PostGIS.

We next run the DBSCAN algorithm using the sklearn.cluster.DBSCAN function from the scikit-learn library. This function requires three parameters. The first two are analogous to the ones in the PostGIS function, namely, eps and min_values). The third parameter is metric, which is the metric to compute the distance between samples. We used Euclidean distance and great

circle distance (explained in Sect. 5.1.3). Figure 10.30 shows the result using great circle distance for distances of 500, 600, and 700 meters and min_values of 8 and 10. We can see that the clusters in this case are more clearly defined than in Fig. 10.27, where the Euclidean distance was used.

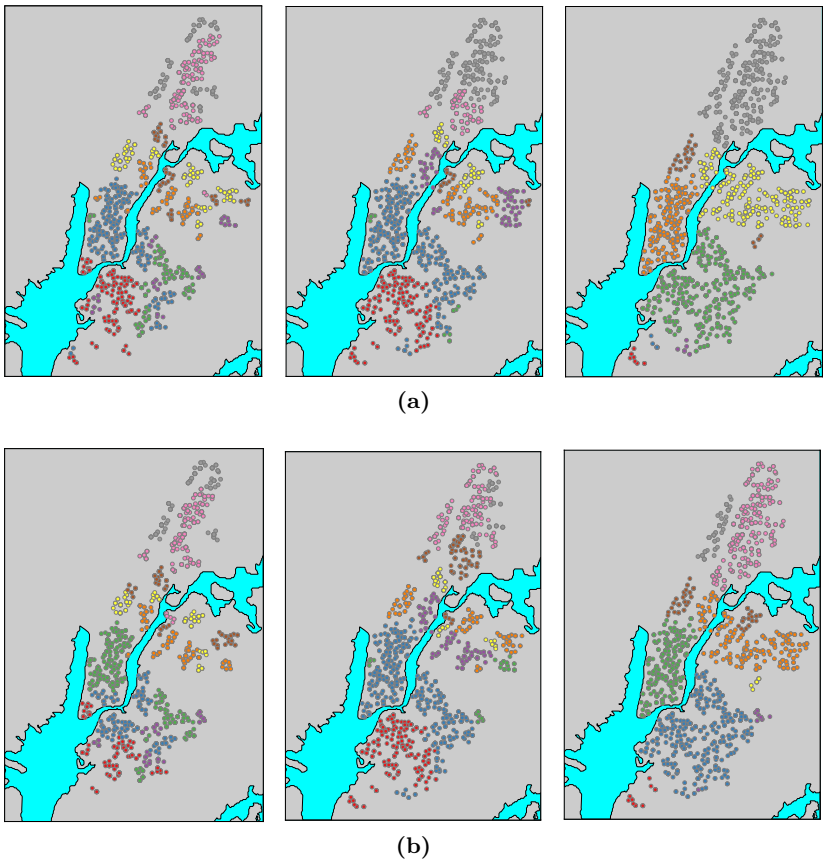


Fig. 10.30 DBSCAN clustering with great circle distance, a minimum sample of (**a**) 8 and (**b**) 10, and distances of 500 m (left), 600 m (center), and 700 m (right).

10.5.3 Temporal Clustering with Agglomerative Algorithms

We now continue using agglomerative clustering to analyze if we can build clusters according to the usage of the stations across time rather than based

on their spatial location. For this, we must consider the dfStFlow DataFrame, since it is the one containing the actual way in which stations are used to pickup and return bicycles. In this section, we use the SciPy library.[8]

We first create a working DataFrame by pivoting dfFlow in a way such that we obtain a column for each time bin, as follows:

```
df_temp = dfFlow.pivot(index='station_id', columns='start_time').reset_index()
```

This results in a DataFrame with 673 columns, one for the station_id and 672 corresponding to each observation bin, containing the count of pickups and returns for each bin. This is shown in Fig. 10.31. From this DataFrame, the feature array to be used in the clustering process is obtained after dropping the station_id column as:

```
X = df_temp.drop(["station_id"], axis=1).values
```

	station_id	in_count									...	out_count
		2024-03-01 00:00:00-05:00	2024-03-01 00:30:00-05:00	2024-03-01 01:00:00-05:00	2024-03-01 01:30:00-05:00	2024-03-01 02:00:00-05:00	2024-03-01 02:30:00-05:00	2024-03-01 03:00:00-05:00	2024-03-01 03:30:00-05:00	2024-03-01 04:00:00-05:00	...	2024-03-07 19:00:00-05:00
1	2733.03	0	0	0	0	0	0	0	0	0	...	0
2	2782.02	0	0	0	0	0	0	0	0	0	...	0
3	2821.05	0	0	0	0	1	0	0	0	0	...	1
4	2832.03	0	0	0	0	0	0	0	0	0	...	0
5	2861.02	0	0	0	0	0	0	0	0	0	...	0

Fig. 10.31 Working DataFrame for temporal clustering.

The values method obtains the array from the DataFrame. This is a feature array of length 2,143 (the number of stations) where each tuple has 672 features (the in and out counts for each bin). From this array we build the dendrogram shown in Fig. 10.32 with the following code.

```
method = 'Ward'
Z = linkage(X, method)
# Only shows the last 30 merges, only annotate in the figure distance above 300
plot_dendrogram(Z, 30, 300)
```

The function linkage performs agglomerative clustering using the Ward linkage method. The function plot_dendrogram displays the truncated graph. It can be found in the GitHub repository. We can see that only thirty merged samples are shown in the figure.

The final step of the algorithm consists in determining where to cut the dendrogram to form clusters. Normally, the algorithms available in the various Python libraries provide a function for cutting the generated tree into several groups either by specifying the desired number of groups or the cut height. In our example, we cut the hierarchical tree at three different distances

[8] https://scipy.org/

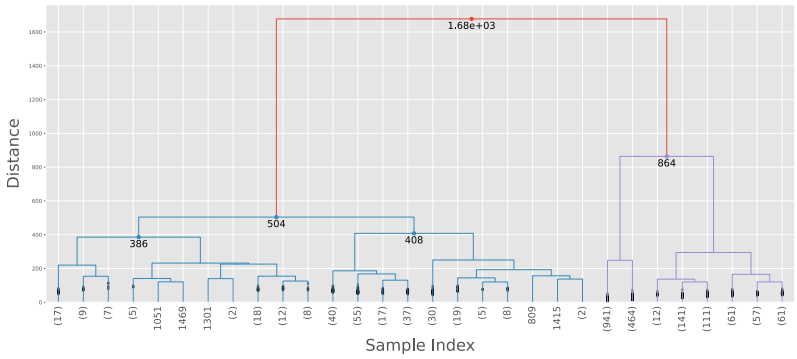


Fig. 10.32 Dendrogram for computing the temporal clustering using Ward linkage.

(800, 600, and 250) to observe the spatial distribution of the clustering result in the map. The function that computes this is:

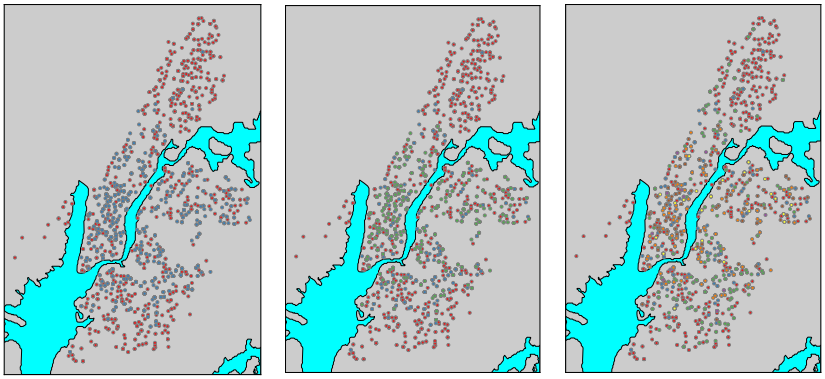
```
dist = [800, 600, 250]
...
df_All['cluster'] = fcluster(Z, dist[i], 'distance')
```

where Z is the array representing the dendrogram (computed as explained above) and dist is an array with the distances we use to cut the dendrogram. The function fcluster assigns values to each object with the cluster identifier. The variable df_All is a DataFrame containing the values in both input DataFrames, namely, dfStInf and df_temp, that is, the station information and the flow counts for each station that are joined using the function merge (details are omitted here and can be found in the scripts in the repository). Note that we need the station information to display the clusters in a map.

The result is shown in Fig. 10.33. We can see that the lesser the cutting height in the dendrogram, the higher the number of clusters obtained. In this case, a cut-off distance of 250 yields eight clusters shown in Fig. 10.33. If we choose a very high distance we end up with few clusters. The interpretation is that each color in represents a different occupation rate at different times. Cutting at 600 (center part of Fig. 10.33) shows the use of the stations, where the green cluster represents the most often used. Again, the interpretation and refinement of this analysis is beyond the scope of this section. We focus here on the presentation of various clustering techniques and methods.

Dimensionality Reduction with Principal Components

Given the large number of temporal dimensions in our problem (that is, 672), we want to see if we can reduce this number and obtain a similar result. This procedure is called **dimensionality reduction** and the technique used is called **principal component analysis** (PCA). Intuitively, the PCA technique performs a linear transformation of the original data onto a



(a) **(b)** **(c)**

Fig. 10.33 Temporal agglomerative clustering using Ward linkage. The clusters were obtained using a cutting of the dendrogram at a vector distance of (**a**) 800, (**b**) 600, and (**c**) 250.

new coordinate system with a reduced number of dimensions, the ones that capture most of the variation in the data. Figure 10.34 shows that in our example, when considering 30 components out of the 672, more than 70% of the variance of the data is explained by the first five components.

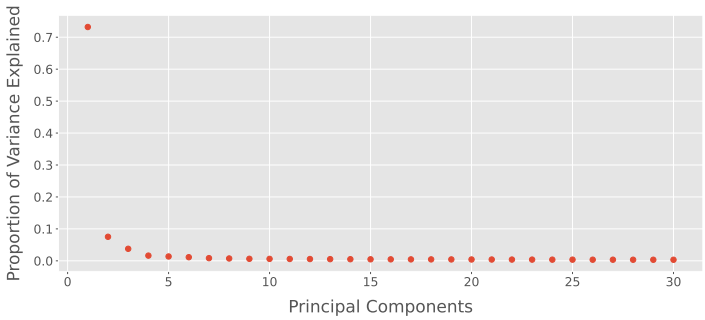


Fig. 10.34 Data variance explained by the first five components of the dataset.

We performed the same analysis as above, now reducing the number of dimensions with the function sklearn.decomposition.PCA from the scikit-learn library. For example, consider the portion of code below:

```
n_components = 30
pca = PCA(n_components=n_components)
pca.fit(X)
X_pca = pca.transform(X)
```

The input to this code is the array X that we introduced above. The output is a transformed array with 30 dimensions, called X_pca, containing 2,143 elements (stations), each one being tuples of 30 values.

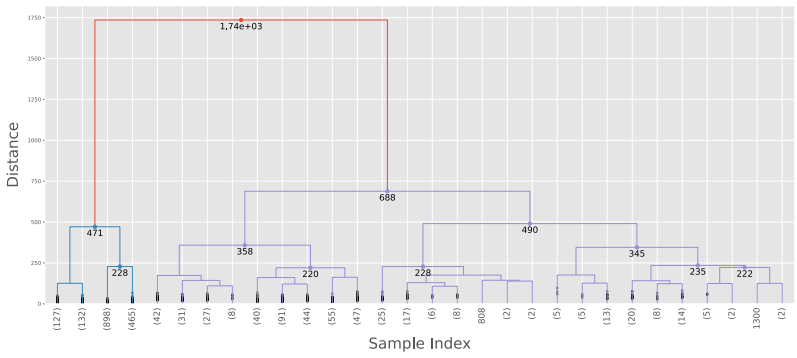
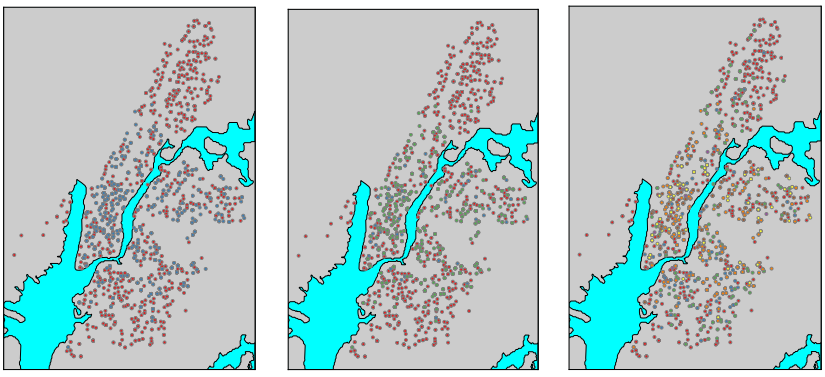


Fig. 10.35 Dendrogram obtained using Ward linkage and dimensionality reduction to 30 temporal dimensions.

Based on the results above, we run the agglomerative clustering algorithm using five components. We obtained the dendrogram in Fig. 10.35, which we can see slightly differs from the one in Fig. 10.32. From this dendrogram we build the clusters with the same values that were chosen when using all the dimensions. We can see that using only five dimensions, the results are very similar to the ones obtained using the full array, with the gain in computational efficiency, which is crucial when we deal with large datasets.



(a) **(b)** **(c)**

Fig. 10.36 Temporal agglomerative clustering with five principal components, using Ward linkage and a distance larger than (**a**) 800, (**b**) 600, and (**c**) 250.

10.5.4 Spatiotemporal Clustering using K-means

We conclude this section performing clustering over the spatial and temporal dimensions together. In this case, we will use the K-means algorithm. The number of temporal features in the dataset (that is, the number of thirty-minute intervals) is much larger than that of spatial features (672 and 2, respectively). Since in clustering all dimensions of the sample points have the same weight, we must fix this imbalance to prevent that the temporal features dominate the clustering result. We use PCA to select only a portion of all the temporal dimensions. In addition, since latitude and longitude values are very different to the values of the flow count, we need to normalize them.

In what follows, and just as an example of the procedure, we will use three combinations: (a) two spatial features and two temporal features, (b) two spatial features and fifteen temporal features, and (c) two spatial features and thirty temporal features. The procedure below combines first the spatial and temporal dimensions and then applies the K-means algorithm over these combined data.

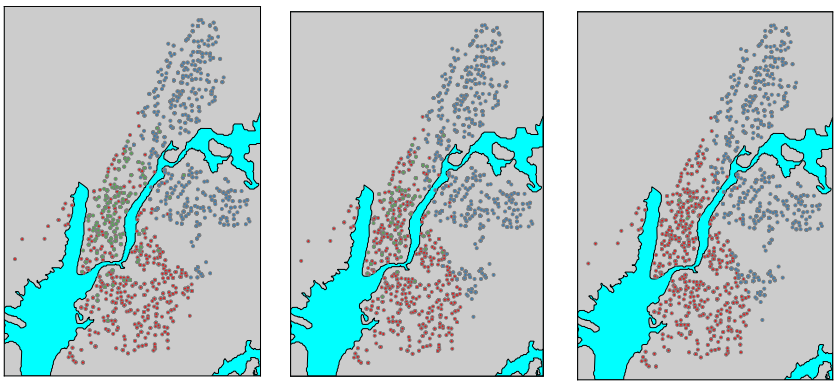
```
X = df_All.drop(["station_id", "station_name"], axis=1).values

def combine_spatial_temporal(X, n_temporal):
  pca = PCA(n_components=n_temporal)
  X_pca = pca.fit_transform(X[:, 2:])
  X = np.hstack((X[:, :2], X_pca))
  scaler = StandardScaler()
  X_std = scaler.fit_transform(X)
  return X_std
```

In the code above, we first drop the unneeded data in the DataFrame, keeping only the latitude, longitude, and the temporal dimensions with the in and out flow counts. Then, the array X is obtained as explained previously. The function combine_spatial_temporal receives this array and the number of components to compute, and returns the transformed matrix. The principal components are computed with the function pca.fit_transform and the function scaler normalizes the values in the dimensions. We then we use K-means over this combined matrix as follows.

```
k = 3
n_components = [2, 15, 30]
Xs = [combine_spatial_temporal(X, i) for i in n_components]
kmean = [KMeans(n_clusters=k).fit(data) for data in Xs]
```

The code above defines $k = 3$ and calls the function combine_spatial_temporal. Then, it uses the combined matrix to apply K-means with $k = 3$. Figure 10.37 shows the result. We can see that, compared against Fig. 10.29b (the one with $k = 3$), there is an influence of the temporal dimensions. Considering only spatial distribution divides the stations in three well-defined clusters. However, the use of the stations across time can be clearly noted, since clusters are not uniformly distributed now, given the impact of the temporal distribution.



(a) **(b)** **(c)**

Fig. 10.37 Spatiotemporal K-means results using $k = 3$ and (**a**) 2, (**b**) 15, and (**c**) 30 temporal components.

10.6 Trajectory Clustering

We conclude the chapter integrating the techniques covered so far. We analyze maritime trajectory data to uncover common shipping routes. The rationale is that, although ships do not follow strictly predefined paths, they frequently travel along similar routes due to regulatory, safety, and efficiency factors. The primary goal of our analysis scenario is to identify and cluster these recurring routes, which is achieved through two main steps. First, we segment vessel trajectories based on their visits to harbors, creating distinct trip sections that are meaningful for route analysis. To improve computational efficiency, we use trajectory simplification techniques that enable faster processing without sacrificing the essential shape of each trip. Then, we apply the DBSCAN clustering algorithm to group trips that exhibit similar paths, using Fréchet distance as the similarity measure. This combined approach finds shared shipping routes that are frequented by ships.

10.6.1 Data Preparation

Since we plan to segment trajectories based on the visits to ports, we begin by examining the Harbors table defined in Sect. 10.3. We identified some issues with this table. Importantly, we can see that some harbor geometries are redundant, either identical or positioned so close to each other that they represent effectively the same location. This redundancy can lead to unnec-

essary segmentation of the ship trajectories. Therefore, our first step is to cluster these harbor points, grouping those that are very close to each other.

To prepare for spatial clustering, we create a spatial index on the Geom column of the Harbors table as follows.

```
CREATE INDEX Harbors_Geom_idx ON Harbors USING gist(Geom);
```

This index is essential for optimizing the performance of spatial clustering, and it is particularly beneficial for large datasets. The DBSCAN algorithm has a quadratic time complexity, that is, $O(n^2)$, because for each geometry it performs a neighborhood query to find other geometries within the given distance. With a spatial index, this complexity can be significantly reduced, since the index enables neighborhood queries to run in $O(\log(n))$ time. This optimization has been incorporated into the PostGIS clustering functions discussed in Sect. 10.5.1. When a spatial index is detected, ST_ClusterDBSCAN utilizes it to improve clustering efficiency.

Next, we use the ST_ClusterDBSCAN function to cluster harbor points based on their spatial proximity. Recall that this function groups geometries that are within a given distance from one another, forming dense areas in the space. This is done as follows.

```
CREATE TABLE ClusteredHarbors AS
WITH HarborsDBSCAN AS (
  SELECT ST_ClusterDBSCAN(Geom, 500, 1) OVER () AS ClusterId, *
  FROM Harbors )
SELECT ClusterId, ST_ConvexHull(ST_Collect(Geom)) AS Geom
FROM HarborsDBSCAN
WHERE ClusterId IS NOT NULL
GROUP BY ClusterId;
```

In the ST_ClusterDBSCAN function we used an eps of 500 meters, which seems a reasonable distance for clustering nearby harbors. Notice that in Sect. 10.3, the harbor geometry was generated using a circular buffer of 1,000 meters radius around the OSM node. Therefore, the eps here is actually 1,500 meters. We also used a minpoints of 1, which means that one geometry can by itself form a cluster. Although this is an atypical choice for this parameter, it is needed since we need to merge the redundant harbors, but also keep the harbors that are not redundant as singleton clusters. In the result of the function, each harbor is assigned a cluster identifier. Finally, in the main query we consolidate each cluster into a single geometry applying the function ST_ConvexHull on the grouped geometries within each cluster. The function wraps the clustered points in the smallest convex polygon that encompasses all points in the group.

Figure 10.38 illustrates the outcome of the clustering process. The clustering results in grouping nearby harbors into cluster 18. In addition it creates two singleton clusters 19 and 25, each containing one harbor.

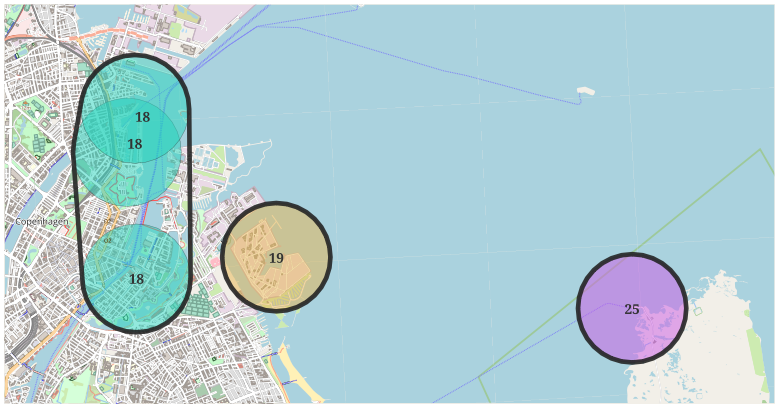


Fig. 10.38 Using ST_ClusterDBSCAN to cluster Danish harbors (colored circles). The resulting clusters are shown with black borders and labels.

10.6.2 Trajectory Simplification and Segmentation

The ClusteredHarbors table created in the previous section can be used to segment the ship trajectories. To speedup the queries below, we first simplify the ship trajectories before segmenting them by proximity to harbors. The following query uses the douglasPeuckerSimplify function with a tolerance value of 100 meters. This simplification keeps the main shape of each trip while reducing the number of points, which optimizes further processing steps.

```
CREATE TABLE SimplifiedShips AS
SELECT MMSI, douglasPeuckerSimplify(Trip, 100) AS Trip
FROM Ships;
```

Once the ship trips are simplified, we segment the trips using the harbors. This segmentation enables us to identify distinct sections of each ship trajectory based on their proximity to harbor clusters.

```
CREATE TABLE TripsByHarbors(MMSI, TripId, TripSegment, TrajSegment) AS
WITH HarborIntersections AS (
  SELECT s.MMSI, h.Geom
  FROM SimplifiedShips s, ClusteredHarbors h
  WHERE eIntersects(s.Trip, h.Geom) ),
IntersectionAggregates AS (
  SELECT MMSI, ST_Union(Geom) AS Geom
  FROM HarborIntersections
  GROUP BY MMSI ),
TripsSegments AS (
  SELECT s.MMSI, unnest(sequences(minusGeometry(Trip, Geom))) AS Trip
  FROM SimplifiedShips s, IntersectionAggregates h
  WHERE s.MMSI = h.MMSI )
SELECT MMSI, ROW_NUMBER() OVER (PARTITION BY MMSI) AS TripId, Trip,
  trajectory(Trip)
FROM TripsSegments;
```

The segmentation process, which is the same as in Sect. 10.2, is done in several steps. In table HarborIntersections, we select the ships whose trajectories intersect with any harbor. In the IntersectionAggregates table, we aggregate these intersecting geometries per ship using the ST_Union function. This operation combines all intersecting geometries for each MMSI, producing a single aggregated geometry that represents all harbor intersections for each trip. Table TripsSegments subtracts the aggregated intersections from each trip using the minusGeometry function. This operation segments the trip by removing the parts that intersect any harbor. Then, the segments are obtained with functions sequences and unnest. Finally, in the main query each resulting segment is assigned a unique TripId with the ROW_NUMBER window function. Note that this query is only concerned with ships that visit one or more harbors. Ships that only traverse the area of study without visiting harbors are not captured. In other words, this query aims to identify the local ship traffic that start, end, or alternate between harbors in Denmark.

10.6.3 Clustering Analysis

To identify similar shipping routes based on trajectory similarity, we need to run another DBSCAN clustering task, yet this time using some trajectory similarity measure. We cannot use the ST_ClusterDBSCAN function, because it computes the Euclidean distance between input geometries. Instead, we need a classification function that would accept the distance function as a parameter, e.g., passing a callback function for distance computation. The sklearn Python library allows such flexibility. The following solution is a combination of SQL and Python. We first compute the Fréchet distance between each pair of segmented trips as follows.

```
CREATE TABLE FrechetMatrix(MMSI1, TripId1, MMSI2, TripId2, Dist) AS
SELECT a.MMSI, a.TripId, b.MMSI, b.TripId,
  frechetDistance(a.TripSegment, b.TripSegment)
FROM TripsByHarbors a, TripsByHarbors b;
```

The frechetDistance function computes the Fréchet distance between each pair of trips, where each row in FrechetMatrix consists of two trips and their corresponding distance. With this distance matrix, we move to clustering the trips using the DBSCAN algorithm in sklearn to identify groups of similar trips. The script below starts by fetching the precomputed distance values:

```
# Library imports
import psycopg as pg
import pandas as pd
from sklearn.cluster import DBSCAN
import numpy as np
# Connect to the database
connection = pg.connect(...)
cursor = connection.cursor()
```

```
# Query to get the Frechet distances as a DataFrame
query = """
  SELECT MMSI1, TripId1, MMSI2, TripId2, Dist
  FROM FrechetMatrix;
"""
distance_df = pd.read_sql(query, connection)
```

Next, we prepare a numpy matrix with these distance values, in the format needed by sklearn:

```
# Convert the distance DataFrame into a distance matrix suitable for DBSCAN
unique_trips = sorted(set(distance_df["tripid1"]).union(set(distance_df["tripid2"])))
trip_index = {trip_id: i for i, trip_id in enumerate(unique_trips)}
n = len(unique_trips)
distance_matrix = np.full((n, n), np.inf)

for _, row in distance_df.iterrows():
  i, j = trip_index[row["tripid1"]], trip_index[row["tripid2"]]
  distance_matrix[i, j] = row["dist"]
  distance_matrix[j, i] = row["dist"]
```

Next, we apply DBSCAN clustering to this distance matrix.

```
# Apply DBSCAN using the distance matrix
epsilon = 20000 # Adjust based on distance units
min_samples = 2
dbscan = DBSCAN(eps=epsilon, min_samples=min_samples, metric="precomputed")
clusters = dbscan.fit_predict(distance_matrix)
```

The eps is set to 20 km and min_samples is set to 2, ensuring that any cluster contains at least two trips. The metric is set to precomputed, since the distances are already calculated and are supplied via the fit_predict function.

Once the clusters are generated, the results are stored back in the MobilityDB database. Each trip is assigned a cluster label. These cluster labels are written to the TripsByHarbors table in PostgreSQL.

```
# Map results back to trip IDs
cluster_results = pd.DataFrame({
  "tripid": unique_trips,
  "cluster": clusters
})

# Add the ClusterId column if it hasn't been added already
cursor.execute("""
  ALTER TABLE TripsByHarbors
  ADD COLUMN IF NOT EXISTS ClusterId integer;
""")
connection.commit()

# Update the cluster column in table TripsByHarbors based on the clustering results
for _, row in cluster_results.iterrows():
  tripid = row["tripid"]
  cluster = row["cluster"]
```

```
    cursor.execute("""
        UPDATE TripsByHarbors
        SET ClusterId = %s
        WHERE TripId = %s;
    """, (int(cluster), int(tripid)))

# Commit the changes and close the connection
connection.commit()
cursor.close()
connection.close()

print(cluster_results)
```

The final result is that each trip in TripsByHarbors is now assigned a cluster identifier, grouping similar routes together. Figure 10.39 illustrates part of the results. This part of the results seems to nicely achieve the goal of the analytics scenario, as it identifies several ship routes between harbors. Further, between two harbors, we see that sometimes two clusters are identified, reflecting the two directions of travel.

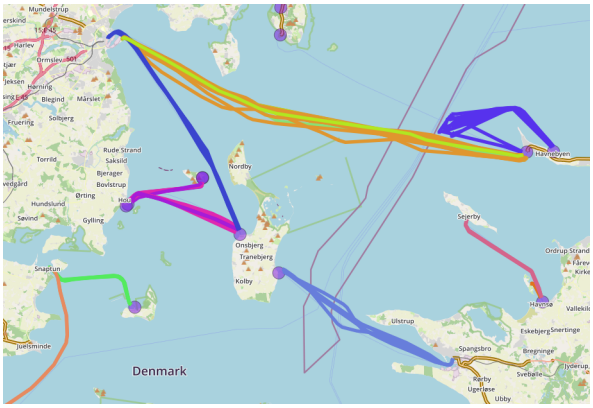


Fig. 10.39 Clustering the trip trajectories using Fréchet distance with eps = 10000 and minpoints = 2. Trip trajectories that belong to the same cluster are displayed in the same color. The circular geometries of the harbors are displayed in half transparent color.

In contrast, Fig. 10.40 illustrates a case where the clustering of harbors incorrectly merges two harbors that should remain distinct. The figure shows a ship route, possibly a ferry, between these two harbors. This route was not captured by the above analysis because of the wrong clustering of the harbors. Attempting to resolve this issue, e.g., by reducing the eps parameter, could help separating these two harbors, but it might result in the wrong splitting of other harbors. This challenge of selecting optimal parameter settings is common in machine learning. We envision other sources of inaccuracies in the above analysis pipeline, including:

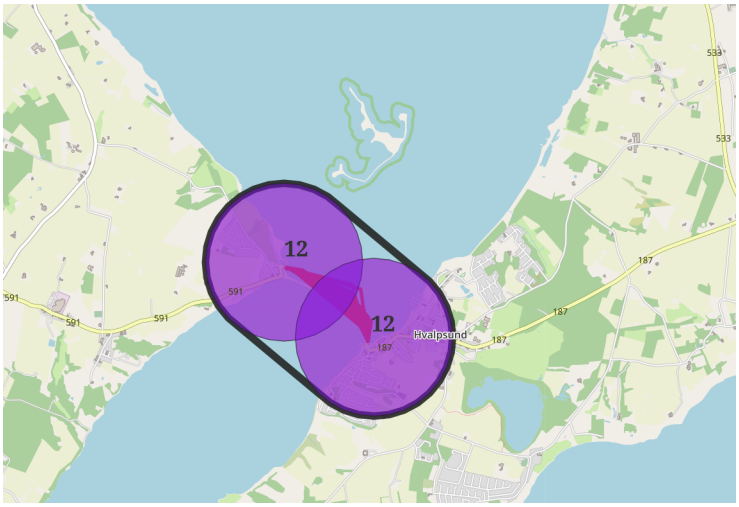


Fig. 10.40 Clustering of harbors mistakenly combining two harbors that should stay separate in cluster 12. A ship route, possibly a ferry, connecting these harbors was missed in the previous analysis due to this incorrect clustering.

- The harbors dataset of OSM need to be checked for accuracy and completeness.
- The clustering of harbors may consider attributes other than the geospatial location, e.g., harbor name.
- The dataset of this analysis consists of one-day AIS trajectories. A bigger data sample, e.g., covering various days of the week, and various seasons, would provide a more complete coverage of the frequent route network, and would make the clustering more robust to noise.
- The noise, errors, and gaps in the original ships trajectories may confuse the clustering and should be cleaned before the analysis.
- Before clustering the trip trajectories, it might be necessary to resample them in a similar sampling rate. Notice that the discrete similarity measures such as DTW, DFD, and TWED are sensitive to different sampling rates of their parameters.
- Several configurations for the trajectory clustering step need to be tested. For instance, Fig. 10.41 illustrates the result of clustering based on DTW distance with the same DBSCAN parameters as above. To obtain this figure we must compute the table containing the DTW distance matrix and use it in the computation instead of the FrechetMatrix table. The table is computed as follows:

```
CREATE TABLE DTWMatrix(MMSI1, TripId1, MMSI2, TripId2, Dist) AS
SELECT a.MMSI, a.TripId, b.MMSI, b.TripId, dynTimeWarpDistance(a.trip, b.trip)
FROM TripsByHarbours a, TripsByHarbours b;
```

This result is significantly different than Fig. 10.38, as less routes were identified. Therefore, various distance measures and various values for the DBSCAN parameters need to be tested.

- Finally, the identified routes should be validated. Clustering, being an unsupervised learning method, is difficult to validate automatically. In this dataset, other data attributes could be used to have an indication of accuracy, such as ShipType and Destination.

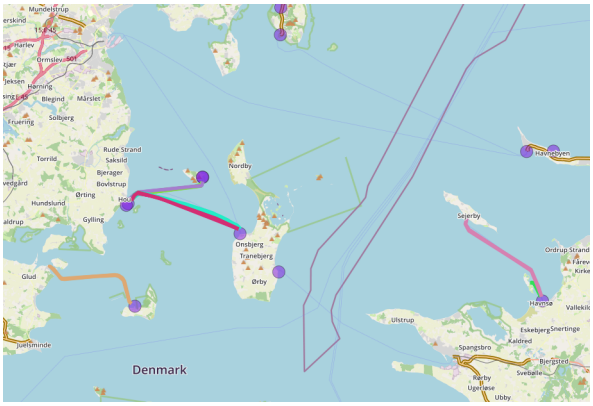


Fig. 10.41 Clustering the trip trajectories using on Dynamic Time Warp distance with eps = 10000 and minpoints = 2.

While this section is focused on demonstrating the integration of the different analysis techniques discussed in this chapter, developing it into an accurate analytics workflow would need elaborating the points above, and assessing the outcomes at every step.

10.7 Summary

This chapter studied core mobility analysis tasks, including simplification and compression, segmentation, heat map visualization, similarity measurement, and clustering. We first presented well-known trajectory simplification methods, from simple intuitive compression to the more advanced Douglas-Peucker (DP) algorithm. Further, we discussed the integration of synchronous Euclidean distance (SED) with DP for preserving both spatial and temporal fidelity. Then, we studied trajectory segmentation methods based on gaps, stops, and points of interest.

The chapter then presented comprehensive examples of different kinds of heat maps, a popular technique for exploring data distribution and movement patterns. We started with conventional spatial heat maps and then extended

them to incorporate the temporal dimension, creating spatiotemporal grids that capture variations in movement intensity over time. Advanced visualization techniques, such as flow maps, were also introduced for examining movement between points of interest.

We also studied the most commonly used similarity measures, which are essential for clustering and pattern recognition in movement data. Three similarity measures were covered, namely, dynamic time warping (DTW), discrete Fréchet distance (DFD), which, and Time Warp Edit Distance (TWED).

We concluded with a comprehensive explanation of spatial and spatiotemporal clustering, based on the similarity measures presented in the first part. A scenario combining several of the aforementioned techniques to identify recurring routes between harbors was illustrated.

10.8 Bibliographic Notes

The Ramer-Douglas-Peucker (RDP) algorithm, discussed in this chapter, was introduced by Ramer in 1972 [172] and refined by Douglas and Peucker in 1973 [63]. Building on this, the Synchronous Euclidean Distance (SED) approach incorporates both spatial and temporal dimensions into distance calculations, providing a more comprehensive simplification technique [133, 170]. Zhang et al. presented in [245] a comprehensive analysis of trajectory simplification, assessing different simplification algorithms on quality metrics. Temporal aggregation, which is the basis of the tsample and tprecision compression functions, has been studied in [85, 115, 137, 223, 236].

The segmentation of moving object trajectories into individual trips has been explored across various domains. Wu et al. [232] presented a method for segmenting ship trajectories into fishing and sailing segments, while [107] addresses segmentation by stops from noisy GPS data with time gaps, improving accuracy in identifying meaningful segments.

Wilkinson and Friendly's review of heat map history [230] offers insights into statistical visualization. A book presenting extensive coverage of trajectory visualization techniques is [16]. The M^3 model by Graser et al. [90] provides a distributed, incrementally updatable framework for exploring large-scale movement data using heat maps.

On the topic of trajectory similarity, the Dynamic Time Warping algorithm (DTW) has been introduced in 1960s [31]. Before its application in geospatial and mobility data, it has been extensively explored in various signal processing applications including speech recognition, handwriting and online signature matching, and gestures recognition. Fréchet distance (DFD), has been introduced in [14], both its discrete and continuous versions. Time Warp Edit Distance (TWED) [140] added a temporal penalty, aiming at combining spatial distances with temporal shifts in the distance calculation. Since trajectories can be very large in size, Dax et al. [58] proposes a method

to compress trajectories and use the resulting sketches for efficient distance caluculation. Comprehensive analyses of trajectory similarity measures can be found in [92, 136, 103]. The time cost of computing these trajectory distance measures can be prohibitive for big datasets. The work in [19] presents one approach for tackling the efficiency problem, by learning the distance function using neural, then efficiently and accurately estimating the distance.

As discussed in the chapter, once a similarity measure is defined for a datatypes, traditional data minging techniques become applicable for this type. The different families of clustering algorithms as well as dimensionality reduction with principal components are covered in many data mining references, e.g., [5]. With focus *similarity queries*, Silva et al. [199] addressed the evaluation and optimization of queries involving various database similarity operators, e.g., including similarity selection, join, and group-by. The example in this chapter uses clustering to identify the routes of ships. This type of analysis has been addressed in more detail in the literature, e.g., [67].

10.9 Review Questions

10.1 Explain the purpose of trajectory simplification.
10.2 Describe the Douglas-Peucker (DP) algorithm and its application.
10.3 Describe how SED modifies the error measurement in the DP algorithm and explain why this modification is important for applications where temporal accuracy is critical.
10.4 How can the concept of a *trip* varies based on application needs?
10.5 Explain how trajectory segmentation can benefit from context-aware methods, such as incorporating road networks or geographical landmarks.
10.6 In the segmentation by stops, what problem arises if all stops are treated as trip breaks? How to address this issue?
10.7 What is the role of Points of Interest (PoI) in trajectory segmentation? Describe an example scenario in which segmentation by PoI would be useful.
10.8 What is the purpose of Overpass Turbo API?
10.9 Discuss the difference between spatial and spatiotemporal heatmaps.
10.10 Why might spatial heatmaps alone be insufficient for mobility data?
10.11 Explain the purpose of the spaceTimeSplit function.
10.12 Describe the effect of using various grid sizes in a heatmap.
10.13 What are the benefits of visualizing data with a time slider?
10.14 What is dynamic time warping (DTW) and how is it calculated?
10.15 Compare the cost of computing DTW in a recursive way versus dynamic programming.
10.16 What are the main differences between DTW and discrete Fréchet distance (DFD) in measuring trajectory similarity?

10.17 How does time warp edit distance (TWED) account for temporal shifts in trajectory data?
10.18 Explain the conceptual difference between K-means and DBSCAN. When would you use each method?
10.19 What is the weight factor in the ST_ClusterKMeans function?
10.20 What is agglomerative clustering?
10.21 Define principal component analysis. What is this technique used for?
10.22 What is a linkage method? Describe various kinds of linkage methods.
10.23 How can unsupervised clustering help identifying common routes?
10.24 Explain the function of ST_ConvexHull in clustering harbors.
10.25 How can trajectory simplification improve analysis efficiency?
10.26 What role does minusGeometry play in trajectory segmentation?
10.27 Describe the importance of parameter selection for clustering.

10.10 Exercises

Exercise 10.1. Using the AIS dataset, identify the ship trajectories with the highest and lowest compression rates, and analyze factors that contribute to different compression rates. Complete the following tasks:

a. Compress the ship trajectories using Douglas-Peucker as in the chapter.
b. Compute the compression rate per trajectory as the number of trajectory instants after compression divided by the number of instants before compression.
c. Find and visualize the top 10 and the last 10 trajectories in the compression ratio.
d. Try to explain the reasons for high and low compression rates. Compare the high-compression and low-compression trajectories. What differences do you observe in their shapes, routes, and patterns?

Exercise 10.2. Segment the ship trajectories by *ports*. Notice that in this chapter we used the *harbor* features of OSM. In this exercise, you need to use the port features, which is a different set of geometries. Complete the following tasks:

a. Using the documentation of OpenStreetMap and Overpass Turbo, formulate a query to fetch Danish port features.
b. Visualize the retrieved port features, and compare to the harbor features that we fetched in the chapter.
c. Adapt the segmentation SQL query presented in the chapter to use the newly created Ports table. This query should identify segments of ship trajectories that intersect with any of the fetched Danish port locations.
d. Visualize the segmented trips, and compare with the segmentation by harbors.

Exercise 10.3. Analyze and visualize ship movements between pairs of harbors. Complete the following tasks:

a. Using the provided SQL query in this chapter, create a table called flow that captures ship movements between harbor pairs.
b. Extend the query to include a timestamp for each ship's arrival at both harborA and harborB.
c. Modify the query to capture the direction of the flow between each harbor pair. For example, add a column direction that stores whether the ship traveled from harborA to harborB or vice versa.
d. Plot a flow map, distinguishing the direction of flow and the frequency of movements.

Exercise 10.4. Analyze similar trips using distance measures. Complete the following tasks:

a. Query the TripsbyHarbors table to locate all trips associated with the ship having MMSI 219019887, and select one TripId among them.
b. Apply the DTW distance function to find the 10 trips in the TripsByHarbors table that are nearest to the reference trip.
c. Use the Discrete Fréchet Distance (DFD) to identify the 10 nearest trips to the reference trip.
d. Repeat finding the nearest 10 trips using Time Warp Edit Distance (TWED).
e. Visualize and compare the three result sets. Discuss the common and different rankings across the DTW, DFD, and TWED measures.
f. Adjust the relevant parameters of the distance functions, and repeat the experiment. Which parameters seem to influence similarity the most?

Exercise 10.5. Consider the NYC Taxi and Uber trips dataset in Ex. 8.2.

a. Download the data for March, 2024, as we did in Sect. 10.5.1 for the NYC Citi Bike dataset. Since data are provided in Parquet format, transform these data into a PostgreSQL database.
b. Perform the cleaning tasks of Ex. 8.2 to ensure the data quality of the new dataset.

Exercise 10.6. Apply *spatial clustering* using PostGIS functions to the NYC Taxi dataset in Ex. 10.5 as indicated next.

a. Run the K-means algorithm using different values of the k parameter.
b. Repeat the above using different weight factors considering the trip attributes in the dataset.
c. Run the DBSCAN algorithm using different values of the eps and minpoints parameters.

Exercise 10.7. Apply *spatial clustering* using Python libraries as explained in Sect. 10.5.2 to the NYC Taxi dataset of Ex. 10.5.

a. Apply K-means clustering with different values of k. Determine the k values using the elbow curve.
b. Apply DBSCAN clustering using at least two different distance functions (e.g., Euclidean and Great Distance).

Exercise 10.8. We now move to *temporal clustering.* Apply agglomerative clustering explained in Sect. 10.5.3 to the NYC Taxi dataset of Ex. 10.5 with different linkage methods. Build the corresponding dendrograms. In all cases use different eps and minsample parameters.

a. Use the following linkage methods over all the temporal dimensions.
   - Single linkage
   - Complete linkage.
   - Centroid linkage
   - Ward linkage.
b. Repeat the above applying dimensionality reduction with different numbers of temporal dimensions.

Exercise 10.9. Apply the K-means algorithm with different values of k, to perform *spatiotemporal* clustering as explained in Sect. 10.5.4 to the NYC Taxi dataset. Use different numbers of temporal dimensions.