SENDAS: Scalable ENrichment for mobility DAta Sets

Henrique S. Santana[1], **Fabrício A. Silva**[1]

[1]Universidade Federal de Viçosa, *campus* Florestal – Florestal, MG – Brasil

{henrique.s.santana,fabricio.asilva}@ufv.br

Abstract. *Recent years of technological advancement and popularization has grown the availability of mobility data, collected from different sources. Analyzing such data is an active research and industrial development field, with a diverse range of applications. However, trajectory data has also been following the Big Data trend, and currently existing tools lack of scalability. To cover this gap, we introduce Sendas – Scalable ENrichment for mobility DAta Sets –, a new Scala library built around the Apache Spark framework, enabling parallel and distributed execution for mobility analysis techniques. We present definitions that extend state-of-the-art notions of trajectory naming, flow calculation and mobility motif identification, and also conduct comparative performance evaluation, finding running time improvements of 4 to 6 times when compared to non-parallel execution.*

1. Introduction

Human mobility is a broad research field, comprehending multiple disciplines and being of interest to diverse applications, like urban planning, population migration analysis, environmental impact studies, recommendation systems, among others. Historical mobility data have been used by researchers and industry to understand and characterize human mobility.

While in the past detailed mobility data – also known as trajectory data – were scarce, with recent years of technological advancement and popularization for various purposes, its availability has been growing in an unprecedented scale and granularity. The use of Call Detail Records (CDRs), GPS data and social network geotagged posts are examples of data collection means that contributed to this change [Barbosa et al., 2018]. However, such data also poses challenges due to its big volume and heterogeneity.

Several tools already exist, usually as Python or R libraries, to address the extraction of primitive or derived trajectory metrics [Graser, 2019], trajectory data processing, plotting and also other specific related features. However, despite the heterogeneous aspect of trajectories is handled by those libraries, to the best of our knowledge, there is currently no practical tool available to deal with mobility data big volume as well, which is a crucial feature to its industry adoption.

Thus, in this work we introduce *Sendas* – Scalable ENrichment for mobility DAta Sets –, a Scala library which closes the aforementioned gaps, and also aims to provide ready implementations of state-of-the-art semantic enrichment algorithms and techniques for mobility analysis. *Sendas* is built on top of Apache Spark framework [Zaharia et al., 2012] and uses Apache Sedona, previously known as GeoSpark [Yu et al., 2019], to aid with geospatial queries.

The main goals of Sendas proposal can be summarized as:

- Provide scalable implementations of trajectory analysis and transformation, covering a common gap in existing tools;
- Specify an easily extensible and adaptable library for research and development application;
- Extend mobility flow definition to include temporal analysis;
- Present a more restricted definition for mobility motifs, enabling more efficient implementations.

This paper is organized as follows: Section 2 discusses related works. Section 3 presents key concepts to understanding the proposal. Implementation details and project design choices are described in Section 4. Section 5 shows an experimental and comparative evaluation of *Sendas*. Finally, concluding remarks are given in Section 6, along with future directions and challenges.

2. Related Work

Currently existing tools for handling mobility data include mainly Python and R libraries, which are popular programming languages in data science. We can highlight three libraries that most influenced Sendas development and research.

scikit-mobility [Pappalardo et al., 2021] is a Python library that provides a wide range of features, including trajectory metrics, clustering, filtering, plotting and aggregation into collective flow data, as well as generating synthetic mobility traces and privacy risk assessment. It is based on other popular Python libraries, primarily *pandas* [McKinney et al., 2010] and *GeoPandas* [Jordahl et al., 2019].

MovingPandas [Graser, 2019] is another Python library that uses *GeoPandas*, but with a greater focus on calculating trajectory metrics. It also supports visualization and integration with desktop QGIS applications. And finally *trajectories* [Moradi et al., 2018] is an R library that likewise focuses on metrics extraction and plotting, while also providing trajectories simulation and model fitting.

All of those libraries share a common problem: scalability. *GeoPandas* data structures, for instance, do not support parallel or distributed computing natively. On the other hand, Apache Spark [Zaharia et al., 2012] framework provides data abstractions that allows a programmer to easily parallelize their tasks with high-level transformations. Apache Sedona, formerly known as GeoSpark [Yu et al., 2019], extends Spark by introducing geometrical data types into Spark, and provides efficient operations like spatial joins through partitioning and indexing algorithms.

Both Spark and Sedona are implemented in Scala, an objected-oriented and functional programming language that runs on JVM (Java Virtual Machine), bringing type safety, expressiveness and performance to code. Although both provide API bindings to work with Python and R, this can become a computational overhead, as it has been shown that Spark code run on Scala can be about 10 times faster than running on Python [Gupta and Kumari, 2020; Ji and Kwon, 2020]. By choosing to develop our library in Scala, we aim for efficiency gains and consider this an opportunity to encourage the use of Scala in the mobility data analysis field.

3. Definitions

Not only mobility data can be represented in multiple ways, but there is a lack of standardized terminology among the field [Graser, 2019]. With current data collection techniques, it is possible to obtain records of individual moving objects – people, animals, vehicles. So, we take this as the base definition for trajectories: a chronologically ordered sequence of points in space, belonging to a specific object [Zheng et al., 2014; Pappalardo et al., 2021]. More generally, we can use any *spatial representation*, and, since this definition relies on a single temporal information, we call it the instant-based trajectory.

Definition 1 *The instant-based trajectory of a moving object o is defined as a sequence of tuples $IT_o = \{(t_1, s_1), ..., (t_n, s_n)\}$, where s_i is a spatial object, t_i is a timestamp, and $t_i < t_j$ if, and only if, $i < j$.*

However, in order to capture other aspects of mobility semantics, and for convenience, two other trajectory definitions are adopted. One of them comes from the concept of stay points [Montoliu et al., 2013], in which each data record represents permanence of a moving object in a location for a given time.

Definition 2 *The stay-based trajectory of a moving object o is defined as a sequence of tuples $ST_o = \{(ts_1, te_1, s_1), ..., (ts_n, te_n, s_n)\}$, where s_i is a spatial object, ts_i and te_i are respectively the start and end timestamps of permanence in s_i, $ts_i < te_i$, and $te_i < ts_j$ if, and only if, $i < j$.*

The other definition is, as Graser [2019] states, a line-based approach. In this context, we name data records as moves, since each of then represents movement of a moving object from an origin to a destination.

Definition 3 *The move-based trajectory of a moving object o is defined as a sequence of tuples $MT_o = \{(to_1, td_1, so_1, sd_1), ..., (to_n, td_n, so_n, sd_n)\}$, where so_i and sd_i are spatial objects respectively denoting origin and destination of the move, to_i is the departure timestamp from so_i, td_i is the arrival timestamp at sd_i, $to_i < td_i$, and $td_i < to_j$ if, and only if, $i < j$.*

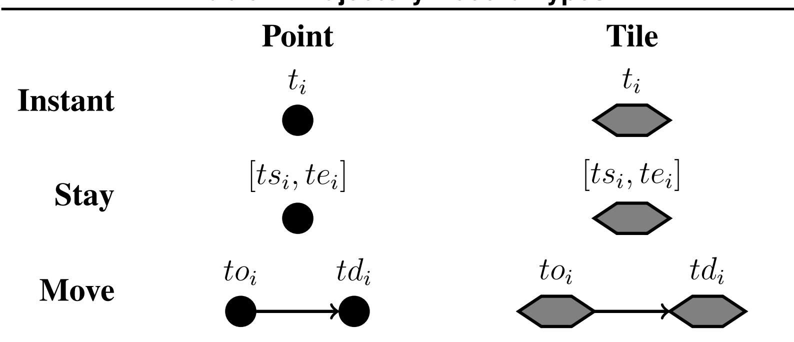
These definitions denote complementary ways to represent trajectories, and are thus called *trajectory views*, being used in the library as shown in Section 4. In every definition, each tuple constituting a trajectory is called a trajectory record.

Usually, mobility data is expressed in terms of points on a plane or on the Earth. Nonetheless, it is also useful to abstract each individual point into an aggregated notion of a region. E.g., instead of saying a person went from a given latitude and longitude to another, we could say they went from city A to city B. For that purpose, we use the concept of tessellations, which is the partitioning of space into geometries or tiles [Lee et al., 2000].

Definition 4 *A tessellation is a bijective function $T : \Theta \rightarrow \Gamma$, where Θ is a set of tile identifiers and Γ is a set of geometries. $\forall g \in \Gamma$, g supports containment operation with points $p \in \mathbb{R}^n$, $n \geq 1$.*

Tessellations can be feature-primary, dividing space according to real world characteristics, e.g. districts of a city or country borders. They can also be space-primary,

Table 1. Trajectory Record Types



dividing space according to a function or an algorithm, independently from known features, e.g. Uber H3 [Brodsky, 2018] system. There are other ways to classify tessellations, but they are all treated equally under an interface in the library.

Thus, depending on what kind of spatial objects are used in a trajectory, we can call it a point-based trajectory or a tile-based trajectory, as in Definitions 5 and 6, respectively. In order to fully characterize a set of trajectories, both the spatial representation and trajectory view should be specified, as shown in Table 1.

Definition 5 *A point-based trajectory is a trajectory whose spatial objects are points* $p_i \in \mathbb{R}^n$.

Definition 6 *Given a tessellation* $T : \Theta \to \Gamma$*, a tile-based trajectory is a trajectory whose spatial objects are tile identifiers* $\theta_i \in \Theta$.

Among important aggregated metrics for mobility analysis are origin-destination matrices, or flows, which measure the number of movements between pairs of regions. For it is often useful to analyze flow along time, we extend the notion of flow to include a temporal dimension, splitting flow into *inflow* and *outflow*, as stated in Definition 7.

Definition 7 *Flow is defined as a set of matrices* Φ*,* $\forall F_t \in \Phi$*,* F_t *is an* $O \times D$ *matrix, and* $F_{t,OD} = (f^{out}_{t,od}, f^{in}_{t,od})$*, where* $f^{out}_{t,od}$ *is the number of moving objects departing from region* o *at time slot* t *towards region* d*,* $f^{in}_{t,od}$ *is the number of moving objects arriving at region* d *at time slot* t*, coming from region* o*, and* O, D *are the number of distinct origin and destination regions, respectively.*

Finally, in order to capture repeating patterns of individual moving objects, we use the concept of mobility motifs [Schneider et al., 2013]. In analogy to complex networks, if movements are organized into a temporal network, – i.e., a temporal graph – a mobility motif is a recurring subnetwork that represents the visited locations of a moving object along a time slot. Since no moving object can ever be in two places simultaneously, it is reasonable to assume a strict order among the temporal edges, unlike other works related to temporal motifs which address possibly simultaneous events [Lei et al., 2020; Sun et al., 2019; Kovanen et al., 2011].

Definition 8 *The motif* M_{ot} *of a moving object* o *during time slot* $t = (t_{start}, t_{end})$ *is a temporal graph* $M_{ot} = (V, E)$*, where* V *is the set of visited regions and* E *is the set of strictly ordered temporal edges* $E = \{(u_i, v_i, t_i) | u_i, v_i \in V, t_{start} \leq t_i < t_{end}\}$.

4. Sendas

In order to provide an easily extensible and maintainable library, the core functionality of *Sendas* is built around the multiple definitions of trajectories, whose classes and traits [1] are defined at the root level of the package. The other features are organized into the following packages:

- `tessellation`: definition of available tessellations;
- `analysis`: trajectory metrics extraction;
- `processing`: trajectory clustering, transformation, etc.
- `flow`: aggregation of movements into flow matrices;
- `motif`: detection and labeling of mobility motifs;

Figure 1 provides an overview of *Sendas* currently available main features. Data can be loaded from various sources and also exported to different formats through Spark `DataFrame` abstraction. Internally, *Sendas* `TrajDataFrame` undergoes transformations such as `StayDetection` (Section 4.3) and `Tessellation` (Section 4.2), and is then passed to `Flow` (Section 4.4) or `MotifLabeling` (Section 4.5) features.

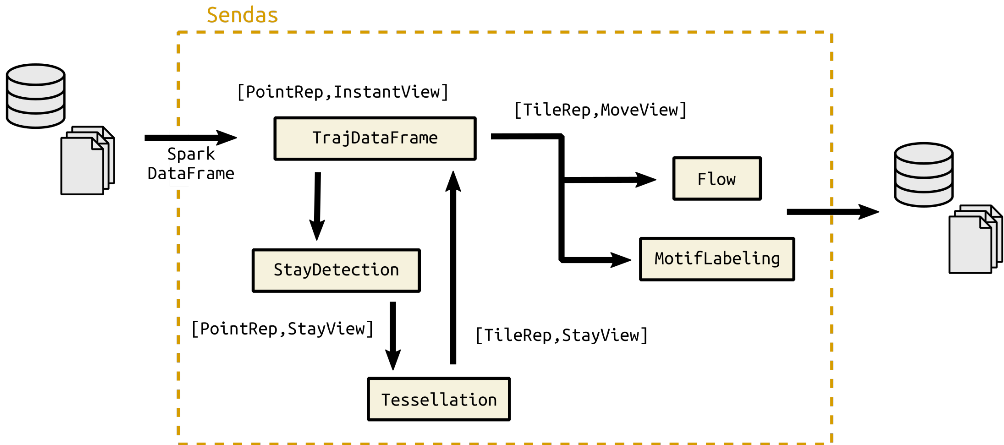


Figure 1. Overview of *Sendas* features and example data flow steps.

4.1. TrajDataFrame

At the library root package, *Sendas* core class can be found, `TrajDataFrame`. This class is a wrapper around Spark's high-level SQL API, the `DataFrame`, thus consequently benefiting from the various data input and output formats it provides. To use any *Sendas* feature, data has to be first loaded into a `DataFrame`, which has tabular format where each column is an attribute and each row is an observation. A `TrajDataFrame` is then built, as long as the provided columns conform to a specific trajectory representation.

As Figure 2 shows, to specify a trajectory representation, users must choose a spatial representation – expressed in trait `SpatialRep` – and a trajectory view – trait

[1] In Scala, a trait is almost the equivalent of a Java interface. Classes and objects can extend traits, but traits cannot be instantiated. A notable difference from Java interfaces is that traits can declare attributes as well as methods.

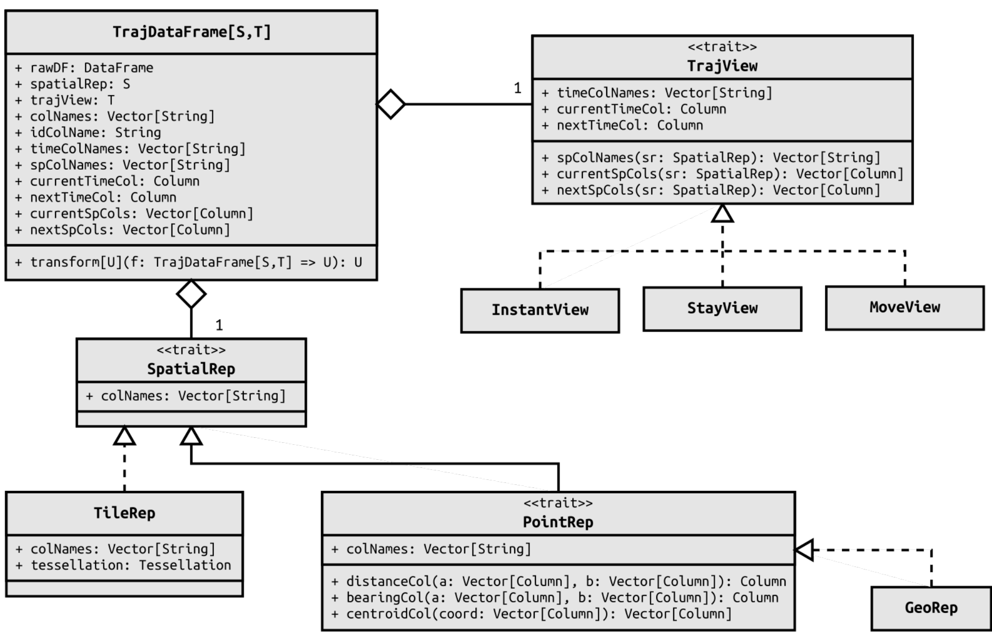


Figure 2. Class diagram related to `TrajDataFrame` class.

`TrajView`. By exposing their types through `TrajDataFrame` type parameters[2] and taking advantage of Scala type system, we allow features to specify which representation is handled as input by their functions and methods, and to also state explicitly the intended output representation.

The concrete types for `TrajView` map exactly to Definitions 1, 2 and 3, respectively: `InstantView`, `StayView` and `MoveView`. As for spatial representations, the point representation, as in Definition 5, is mapped to another trait, `PointRep`, which define a set of methods necessary to work with point-based trajectories, e.g., a distance function between two points. Currently available concrete types for `PointRep` include only `GeoRep`, which refers to the de facto standard geographical coordinate reference system ESPG:4326, expressing real world latitude and longitude in ranges $[-90, 90]$ and $[-180, 180]$, respectively. However, as long as new coordinate systems implemented in future releases extend the `PointRep` trait, `Sendas` point related features will work properly. The other concrete `SpatialRep` type is `TileRep` (Definition 6), which needs an instance of `Tessellation` to be used.

4.2. Tessellation

The main responsibility of Sendas tessellations is to assign tile identifiers to points and can also be implemented in multiple ways, as long as they implement the `Tessellation` trait. This feature must be implemented in the `toTile` method, as shown in Figure 3.

By not requiring `Tessellations` to explicitly store all of their geometries beforehand, space-primary tessellations that can calculate a tile identifier solely from point coordinates – e.g. H3 or z-order functions – will generate a corresponding polygon or shape only if requested by a call to the `tiles` method, and only for tile identifiers listed

[2]Type parameters in Scala are denoted by square brackets, as seen on the class diagrams.

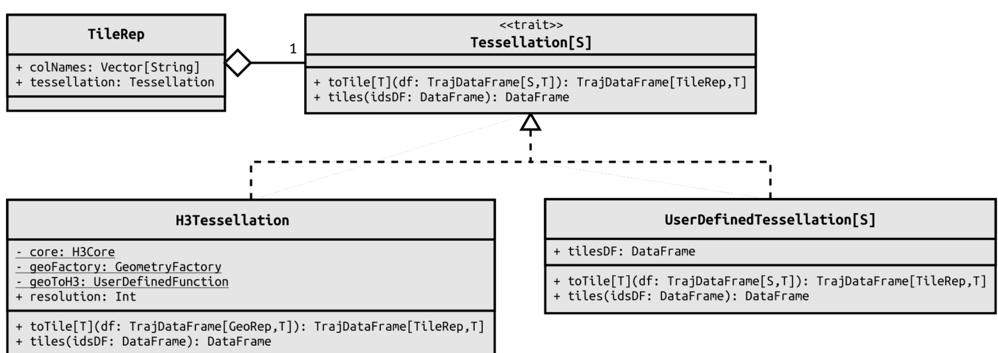


Figure 3. Class diagram related to `Tessellation` trait.

in its input parameter.

`H3Tessellation` is an example of an implemented space-primary tessellation, using Uber H3 indexing library [Brodsky, 2018]. The other currently available implementation is the generic `UserDefinedTessellation`, whose tile identifiers and geometries are directly specified in its attribute `tilesDF`. In this class, points are associated with tiles using Apache Sedona spatial join query, sped up by efficient spatial indexing structures like R-trees and quadtrees.

4.3. Trajectory analysis and processing

Exploratory data analysis, followed by filtering, clustering and applying transformations on data are usual practices in data science in general, and mobility analysis is no exception. Thus, *Sendas* provides functions for introductory extraction of trajectory derivatives [Graser, 2019] (Table 2), and a set of algorithms necessary to the main use cases presented in this paper.

Table 2. Metrics available in the `analysis` package.

Method	Description
`duration`	Calculates the time taken to move between consecutive trajectory points.
`distance`	Calculates the distance between consecutive trajectory points.
`velocity`	Calculates the average speed between consecutive trajectory points, along with the angle (bearing) formed by the points.
`gyradius`	Aggregates every point of each moving object to calculate its radius of gyration.

Among the transformation functions, it is important to highlight two clustering techniques. The first is an algorithm that groups multiple points from point-instant trajectories into point-stay trajectories, based on the work of Montoliu et al. [2013], and available through the `StayDetection` class. The other technique is a simple combination of tile-stay trajectories, which groups consecutive stays on the same tile identifier. Given two consecutive records, belonging to the same moving object o, $r_i = (o, ts_i, te_i, \theta_i)$ and $r_j = (o, ts_j, te_j, \theta_j)$, if $\theta_i = \theta_j$, r_i and r_j combined into $r_{i+j} = (o, ts_i, te_j, \theta)$. This transformation is particularly useful to avoid self loops in flow matrices or motifs.

4.4. FlowDataFrame

Aiming to capture aggregated flow measures, *Sendas* provides the `FlowDataFrame` class, which is built by a constructor method present in the `Flow` object. As shown in Figure 4, by aggregating tile-move trajectories, first an `ODFlowDataFrame` instance is created, which can later be transformed to a `TileFlowDataFrame` if needed. The former maps to Definition 7, while the latter is an even more summarized view of flow data, listing outflow and inflow for a single region in each record, respectively meaning the total number of moving objects departing from, and arriving at the same region.

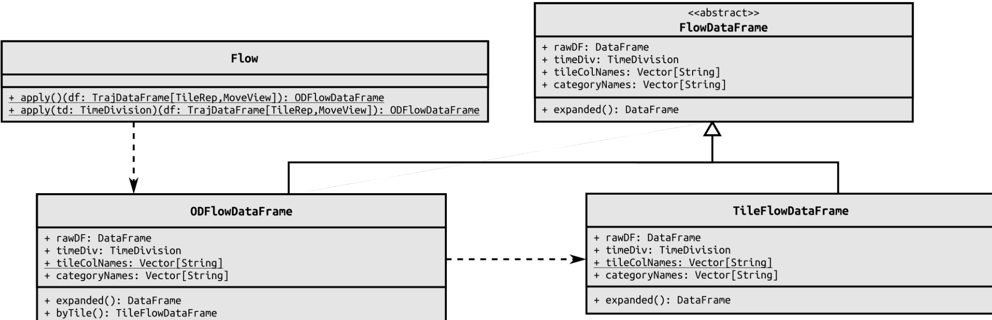


Figure 4. Class diagram related to `Flow`.

Nonetheless, both `FlowDataFrame` subclasses are capable of extending the notion of flow with a temporal dimension by having an instance of `TimeDivision` (Figure 5). It is not mandatory to group movement counts into different time slots, in which case `NoTimeDivision` is used, and the whole data set is considered as a single time slot and flow calculation is the same as other libraries provide. However, if an application needs to calculate flow separately, e.g., by day or by week, an `OffsetTimeDivision` is used.

Along with discretizing time into slots, a `TimeDivision` can also have any number of instances of `TimeCategory`. Each `TimeCategory` contains a function that assigns semantic labels to timestamps, which are later used by `FlowDataFrame` to not only group flow by time slots, but also by each different time category label. Continuing the previous daily time division example, an application could be interested in distinguishing workdays from weekend days, or separating flows that happened during morning, afternoon and night.

The `TimeCategory` function can be arbitrary, but `TimeSpan` classes can be used and chained together to define that function more conveniently: `DiscreteSpan` assigns the given label to specific values, `RangeSpan` assigns the given label to values inside a range, and `DefaultSpan` assigns the given label if no other condition is met.

4.5. MotifLabeling

The extraction of mobility motifs is also an aggregated feature from tile-move trajectories, but relative to each individual moving object. It is provided by *Sendas* through a single function contained in the `MotifLabeling` object. Besides a `TrajDataFrame`, it expects the same `window` and `offset` parameters used by `TimeDivision`, in order to separate trajectories into time slots.

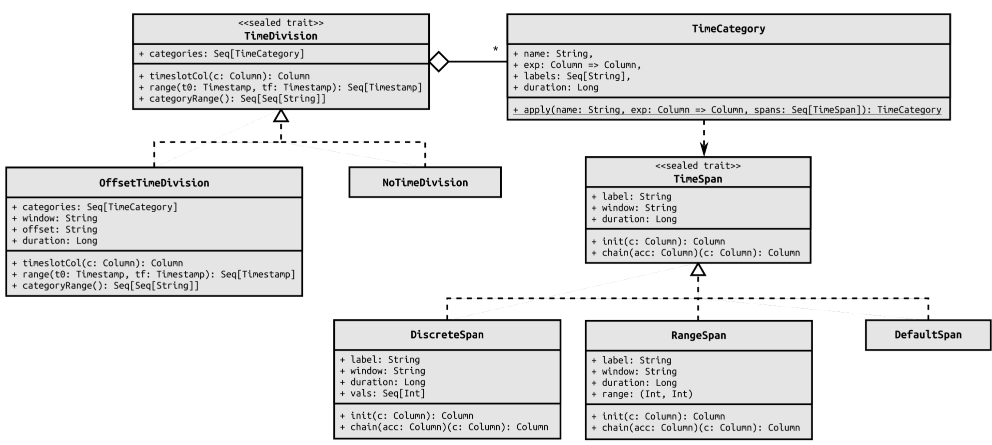


Figure 5. Class diagram related to `TimeDivision` trait.

While the definition of temporal motifs given by the works of Kovanen et al. [2011], Sun et al. [2019] and Lei et al. [2020] applies to ours, it is a more general approach which handles the case of simultaneous temporal edges. Also, following the three-steps motif identification process proposed in [Kovanen et al., 2011] – (1) find all maximal connected subgraphs E^*_{max}, (2) find all valid subgraphs $E^* \subseteq E^*_{max}$ and (3) identify the motif corresponding to E^* –, it is possible to simplify those steps by using the more restricted mobility motif Definition 8 and making certain assumptions.

The first step, as aforementioned, is accomplished by dividing movements into time slots. As for the second step, since we are concerned with capturing the mobility pattern of each object, and not patterns on the whole network formed by all movements, it is enough to assume that the entire sequence of moves performed by that object comprises its motif, for a given time slot. Finally, in order to count how many times each motif was detected for each object, or appeared in the entire data set, it is necessary to compare those graphs and check if they have the same structure, i.e., if they are isomorphic.

General graph isomorphism is an NP-complete problem, but if restrictions are imposed in their structure, polynomial algorithms can be devised [Jiang and Bunke, 1998]. Using our definition, we propose an $O(E \log E)$ technique to assign canonical labels to motifs and use those labels to identify isomorphic motifs, where E is the number of motif edges.

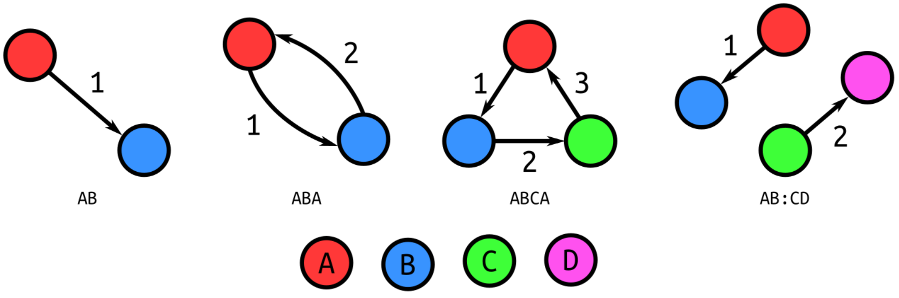


Figure 6. Examples of mobility motifs and their assigned canonical labels.

As seen on Figure 6, each vertex corresponds to a region visited by the examined moving object, and the temporal order of edges is indicated by their labels. Let $E_T = \{e_i | e_i = (u_i, v_i)\}$ be the temporally ordered sequence of edges of a temporal graph $G_T = (V, E)$, and let $\Sigma = \{\sigma_0, \sigma_1, ... | \forall \sigma_i, \sigma_j, i < j \rightarrow \sigma_i < \sigma_j\}$ be a set of unique labels, we can define a bijective function $\sigma : V \rightarrow \Sigma$ that maps each vertex of a temporal graph to a unique label from Σ, according to the order of their first appearance in E_T, i.e., the first vertex in an edge of E_T is mapped to σ_0, the second is mapped to σ_1, and so on. Formally, let $e_0 = (u_0, v_0)$ be the first edge of E_T, then $\sigma(u_0) = \sigma_0$. Also $\forall e = (u, v) \in E_T, \sigma(u) \leq \sigma(v), u = v \leftrightarrow \sigma(u) = \sigma(v)$ and $\forall e_i, e_j \in E_T, i < j \rightarrow \sigma(v_i) \leq \sigma(u_j), v_i = u_j \leftrightarrow \sigma(v_i) = \sigma(u_j)$.

The canonical label of a graph G is then defined as a list of tuples $\tau^*(G) = \{(\sigma(u_0), \sigma(v_0)), ..., (\sigma(u_E), \sigma(v_E))\}$ mapping each ordered edge of E_T with σ, but a proof is necessary to ensure $\tau^*(G)$ is indeed a canonical label. Considering that if, and only if, temporal graphs $G_A = (V_A, E_A)$ and $G_B = (V_B, E_B)$ are isomorphic, there exists a function $\varphi : E_A \rightarrow E_B$, we must show by transitivity that $\exists \varphi : E_A \rightarrow E_B \leftrightarrow \tau^*(G_A) = \tau^*(G_B)$, i.e, there exists a bijective function φ that maps every edge of G_A to an edge in G_B preserving their temporal order if, and only if, their canonical labels are equal.

First, we prove that if such function φ exists, then the canonical labels given by τ^* must be equal. The first condition of φ states that every edge in E_A must be uniquely mapped to an edge in E_B, which means that $|\tau^*(G_A)| = |\tau^*(G_B)|$. The second condition states that the temporal order of the edges is preserved after the mapping. Thus, the first vertex that appears on $E_{T,A}$ is mapped to the same label in Σ as the first vertex that appears in $E_{T,B}$, and the same applies to every subsequent vertices. Hence, $\tau^*(G_A) = \tau^*(G_B)$.

To aid in the second half of the proof, we can define a bijective function $\tau : E \rightarrow \Sigma \times \Sigma \times \mathbb{N}$ that maps each edge of the graph G to a tuple $\tau(e) = (\sigma(u), \sigma(v), i)$, where i is the index of e in the sequence E_T or $\tau^*(G)$.

Next, we prove that if $\tau^*(G_A) = \tau^*(G_B)$, then it is possible to construct a bijective function φ that meets the temporal isomorphism criteria. Such function can be simply defined as $\varphi(e_A) = \tau'_B(\tau_A(e_A))$, in which $\tau_A : E_A \rightarrow \Sigma \times \Sigma \times \mathbb{N}$ maps edges in E_A to their vertices labels and their order in $E_{T,A}$, and $\tau'_B : \Sigma \times \Sigma \times \mathbb{N} \rightarrow E_B$ is the inverse of τ_B and maps vertex labels and order back into edges of E_B. Having τ^* defined for both graphs implies that τ_A and τ'_B can be defined. Since $\tau^*(G_A) = \tau^*(G_B)$, their length must also be equal, which means every edge in E_A can be mapped to an edge in E_B. Also, since the order is also the same and is unambiguous, it is guaranteed that for any two edges e_{Ai} and e_{Aj} in E_A, if e_{Ai} appears before e_{Aj} in the canonical label, there exists two corresponding edges e_{Bi} and e_{Bj} in E_B given by φ that also respect that order. Hence, the proof is complete.

As seen in Figure 6, the chosen Σ set is the standard upper case alphabet letters in ASCII. The canonical label $\tau^*(G)$ is transformed to a string by simply concatenating each $\sigma(u)$, with the following rule. For each consecutive $(\sigma(u_i), \sigma(v_i)), (\sigma(u_{i+1}), \sigma(v_{i+1})) \in \tau^*(G)$, if $\sigma(v_i) = \sigma(u_{i+1})$, $\sigma(u_{i+1})$ is omitted, or else, a break character is inserted before it – a colon character was arbitrarily chosen as the break –, as shown by the `AB:CD` motif. Finding $\tau^*(G)$ and transforming it to a string takes linear time, i.e., it is $O(E)$, but the edges must be previously sorted, which is commonly an $O(E \log E)$ operation, thus the

total complexity is $O(E \log E)$.

The final output for this feature consists of every moving object identifier associated with each time slot and their corresponding motif canonical label. It is worth mentioning that all data preparation steps, as studied in [Schneider et al., 2013], should be done outside this module, e.g. through the stay detection algorithm.

After outlining these definitions and proving our proposal, we further improved state-of-the-art motif labeling techniques in temporal graphs by limiting the scope of the problem to the characteristics of mobility data. Such findings contribute to research and applications in movement pattern analysis by providing an explicit model to identify mobility motifs, independently of extraction methods.

5. Performance Evaluation

In order to assess the scalability and efficiency of *Sendas*, two use case scenarios were proposed to illustrate its currently main implemented features: flow calculation and motif labeling. Then, comparative experiments were conducted to evaluate how the amount of parallel execution threads used by Spark can leverage performance. In the case of flow calculation, since *scikit-mobility* provides this same feature, its execution time on the same dataset is compared to our proposal.

The input dataset is the record of mobile users along the month of November of 2019, offered by a partner company. It is comprised of 49,790 unique users and 8,528,058 rows, as point-instant trajectories, using conventional latitude and longitude coordinates.

As both evaluated features expect tile-move trajectories as input, the first step to use them is to aggregate the data set into point-stay trajectories. Using the `StayDetection` algorithm results in 1,251,003 stay points, as can be viewed in Figure 7. Next, we use an `H3Tessellation` to turn point-stays into tile-stays. To give a fair comparison between *Sendas* and *scikit-mobility* flow calculation, the resulting H3 tile identifiers were converted to their respective polygons and reused as input to `UserDefinedTessellation`. This way, both libraries would need to perform spatial join queries to assign tile identifiers to points.

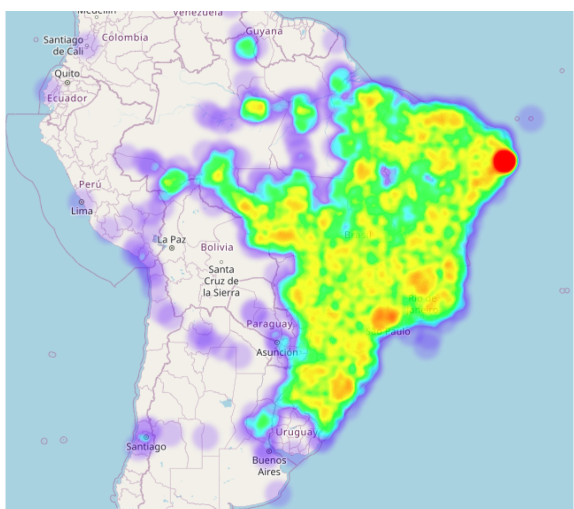


Figure 7. Distribution of resulting stay points.

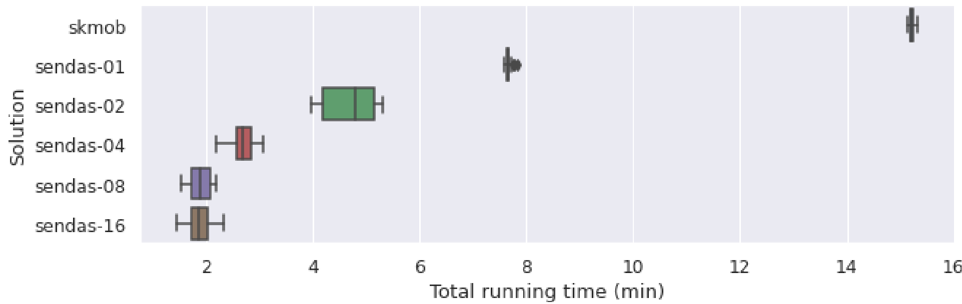


Figure 8. Running time for flow calculation in minutes.

With the appropriate data in hand, we perform the performance evaluation. All tests were performed in a single computer, with the following specifications: 2 Intel Xeon CPU X5650 (12M Cache, 2.66 GHz, 6.40 GT/s Intel QPI, 6 cores, 12 threads) processors, 24 GB DDR3 1333 MHz RAM and 512 GB of storage. For each test instance, the corresponding relevant code section was run 33 times and the total running time measure was recorded, in order to avoid running time disparities due to other processes running in the same machine.

First, for flow calculation, we applied no time division or categorization, to align with the provided features of *scikit-mobility* and ensure both libraries would produce the same output. Code sections whose running time was recorded include loading input trajectory and tessellation data into memory, using a spatial join query to assign tile identifiers to points, counting the number of movements between pairs of regions, and finally writing the results to CSV files. A total of 246,956 unique origin-destination pairs were calculated.

Figure 8 shows tests results, first for *scikit-mobility* (labeled as *skmob*), and then for *Sendas*, with labels from 1 to 16 indicating the number of execution threads used. Even when a single thread is used, *Sendas* performs better, taking up to 8 minutes to complete, while *scikit-mobility* took about 15 minutes, an almost two-fold difference. This could indicate the performance gains of using Apache Sedona spatial indexing capabilities. As the number of threads doubled, execution time approximately halved as well, except for 16 threads, which performed just as well as 8 threads, at around 2 minutes.

Finally, for motif labeling, we use `H3Tessellation` directly instead of using the stored tessellation, performing the following steps. Trajectory data is loaded into memory, then transformed to tile-stay representation, and later to tile-move. `MotifLabeling` is then executed to identify daily motifs, which are then grouped for each user, counted and sorted from the most frequent to the least. The final result is written to disk as CSV files. A total of 2746 unique motifs were identified, with 3.8 unique motifs per user on average. The five most frequent motifs were `AB` (49.7%), `ABA` (13.3%), `ABC` (12.9%), `ABCD` (3.6%) and `ABCA` (2.5%).

Now in the order of seconds, Figure 9 shows an inversely proportional relationship between the level of parallelism and execution time. While a single thread takes about 60 seconds to perform its task, 16 threads take no longer than 10 seconds to execute. This demonstrates the efficiency of *Sendas*, which was designed and implemented following Big Data techniques, in comparison to single-threaded traditional solutions.

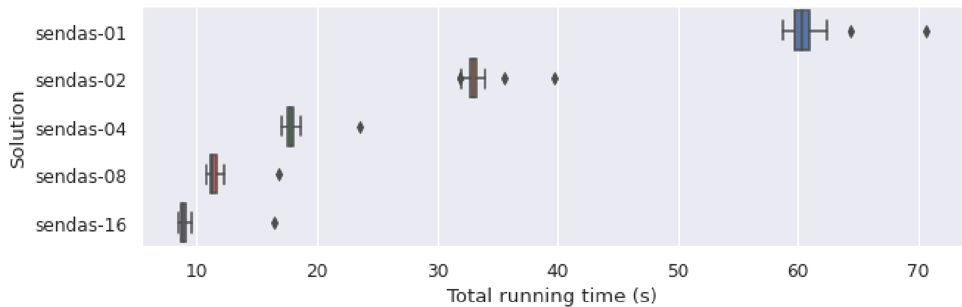


Figure 9. Running time for motif labeling in seconds.

6. Conclusions and future directions

In this work, we proposed *Sendas*, a new Scala library built on top of Apache Spark and Apache Sedona, providing scalable implementations of trajectory data analysis and transformation algorithms. The technical contribution of our work is a scalable and flexible library to process mobility datasets and extract useful knowledge from them. We assess the performance of Sendas running for different levels of parallelism, finding improvements from 4 to 6 times when compared to non-parallel execution.

The scientific contributions of this work are two-fold. We add a temporal dimension to flow calculation to turn the results more flexible. As for motif extraction, we defined more strictly what motifs mean in the mobility context, enabling an asymptotically less complex method for temporal graph isomorphism. These contributions are relevant to the research community in the area of human mobility and communications.

Other future developments in the library include expanding the available analysis and transformation methods to match those of similar packages like *scikit-mobility* and *MovingPandas*. Adding convenient plotting features for Scala practitioners, since many visualization tools have been implemented and long adopted for Python and R, is also an important improvement.

Acknowledgements

This work was supported by Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), and by the Universidade Federal de Viçosa (UFV) Cluster.

References

Hugo Barbosa, Marc Barthelemy, Gourab Ghoshal, Charlotte R. James, Maxime Lenormand, Thomas Louail, Ronaldo Menezes, José J. Ramasco, Filippo Simini, and Marcello Tomasini. Human mobility: Models and applications. *Physics Reports*, 734:1–74, 2018. ISSN 0370-1573. doi: https://doi.org/10.1016/j.physrep.2018.01.001.

Isaac Brodsky. H3: Uber's Hexagonal Hierarchical Spatial Index, 2018. URL https://github.com/uber/h3.

Anita Graser. MovingPandas: Efficient structures for movement data in Python. *GI_Forum – Journal of Geographic Information Science 2019*, 7:54–68, 2019. doi: 10.1553/giscience2019_01_s54.

Yogesh Kumar Gupta and Surbhi Kumari. A study of Big Data Analytics using Apache Spark with Python and Scala. In *2020 3rd International Conference on Intelligent*

Sustainable Systems (ICISS), pages 471–478, 2020. doi: 10.1109/ICISS49785.2020.9315863.

Keung-yeup Ji and Youngmi Kwon. Performance comparison of Python and Scala APIs in Spark distributed cluster computing system. *Journal of Korea Multimedia Society*, 23(2):241–246, 2020.

Xiaoyi Jiang and Horst Bunke. Marked subgraph isomorphism of ordered graphs. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 122–131. Springer, 1998.

Kelsey Jordahl, JV den Bossche, J Wasserman, J McBride, J Gerard, M Fleischmann, J Tratner, et al. Geopandas/geopandas: V0. 6.1. *Zenodo. doi*, 2019.

Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(11):P11005, 2011.

Y.C Lee, Z.L Li, and Y.L Li. Taxonomy of space tessellation. *ISPRS Journal of Photogrammetry and Remote Sensing*, 55(3):139–149, 2000. ISSN 0924-2716. doi: https://doi.org/10.1016/S0924-2716(00)00015-0.

Da Lei, Xuewu Chen, Long Cheng, Lin Zhang, Satish V Ukkusuri, and Frank Witlox. Inferring temporal motifs for travel pattern analysis using large scale smart card data. *Transportation Research Part C: Emerging Technologies*, 120:102810, 2020.

Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

Raul Montoliu, Jan Blom, and Daniel Gatica-Perez. Discovering places of interest in everyday life from smartphone data. *Multimedia Tools And Applications*, 62(1):29. 179–207, 2013. doi: 10.1007/s11042-011-0982-z.

Mehdi Moradi, Edzer Pebesma, and Jorge Mateu. trajectories: Classes and methods for trajectory data. *Journal of Statistical Software*, 2018.

Luca Pappalardo, Filippo Simini, Gianni Barlacchi, and Roberto Pellungrini. scikit-mobility: a Python library for the analysis, generation and risk assessment of mobility data, 2021.

Christian M Schneider, Vitaly Belik, Thomas Couronné, Zbigniew Smoreda, and Marta C González. Unravelling daily human mobility motifs. *Journal of The Royal Society Interface*, 10(84):20130246, 2013.

Xiaoli Sun, Yusong Tan, Qingbo Wu, Baozi Chen, and Changxiang Shen. Tm-miner: Tfs-based algorithm for mining temporal motifs in large temporal network. *IEEE Access*, 7:49778–49789, 2019.

Jia Yu, Zongsi Zhang, and Mohamed Sarwat. Spatial data management in Apache Spark: The GeoSpark perspective and beyond. *Geoinformatica*, 23(1):37–78, 2019. ISSN 1384-6175. doi: 10.1007/s10707-018-0330-9.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX Association. ISBN 978-931971-92-8.

Yu Zheng, Licia Capra, Ouri Wolfson, and Hai Yang. Urban computing: Concepts, methodologies, and applications. *ACM Transaction on Intelligent Systems and Technology*, 10 2014.