Scalable Distributed Subtrajectory Clustering

Panagiotis Tampakis[1], Nikos Pelekis[2], Christos Doulkeridis[3] and Yannis Theodoridis[1]
[1]*Department of Informatics*, [2]*Department of Statistics & Insurance Science*, [3]*Department of Digital Systems*
University of Piraeus, Piraeus, Greece
{ptampak,npelekis,cdoulk,ytheod}@unipi.gr

***Abstract*—Trajectory clustering is an important operation of knowledge discovery from mobility data. Especially nowadays, the need for performing advanced analytic operations over massively produced data, such as mobility traces, in efficient and scalable ways is imperative. However, discovering clusters of complete trajectories can overlook significant patterns that exist only for a small portion of their lifespan. In this paper, we address the problem of *Distributed Subtrajectory Clustering* in an efficient and highly scalable way. The problem is challenging because the subtrajectories to be clustered are not known in advance, but they need to be discovered dynamically based on adjacent subtrajectories in space and time. Towards this objective, we split the original problem to three sub-problems, namely *Subtrajectory Join*, *Trajectory Segmentation* and *Clustering and Outlier Detection*, and deal with each one in a distributed fashion by utilizing the MapReduce programming model. The efficiency and the effectiveness of our solution is demonstrated experimentally over a synthetic and two large real datasets from the maritime and urban domains and through comparison with two state of the art subtrajectory clustering algorithms.**

***Index Terms*—Mobility data, trajectories, subtrajectory clustering, big mobility data mining, distributed clustering, mapreduce**

I. Introduction

Nowadays, the unprecedented rate of trajectory data generation, due to the proliferation of GPS-enabled devices, poses new challenges in terms of storing, querying, analyzing and extracting knowledge from big mobility data. One of these challenges is cluster analysis, which aims at identifying clusters of moving objects (thus, unveil hidden patterns of collective behavior), as well as detecting moving objects that demonstrate abnormal behaviour and can be considered as outliers.

The research so far has focused mainly in methods that aim to identify specific collective behavior patterns among moving objects, such as [5]–[8], [10], [11], [14], [23], [26]. However, this kind of approaches operate at specific predefined temporal "snapshots" of the dataset, thus ignoring the route of each moving object between these sampled points. Another line of research, tries to identify patterns that are valid for the entire lifespan of the moving objects [3], [12], [17], [20]. However, discovering clusters of complete trajectories can overlook significant patterns that might exist only for some portions of their lifespan. The following motivating example shows the merits of subtrajectory clustering.

Example 1. (Subtrajectory clustering) *Figure 1(a) illustrates six trajectories moving in the xy-plane, where each one of them has a different origin-destination pair. More specifically, these pairs are $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow D$, $B \rightarrow A$, $B \rightarrow C$ and $B \rightarrow D$. These six trajectories have the same starting time and similar speed. A typical trajectory clustering technique would fail to identify any clusters. However, the goal of a subtrajectory clustering method is to identify 4 clusters ($A \rightarrow O$ (red), $B \rightarrow O$ (blue), $O \rightarrow C$ (purple), $O \rightarrow D$ (orange)) and 2 outliers ($O \rightarrow A$ and $O \rightarrow B$ (black)), as depicted in Figures 1(b).*

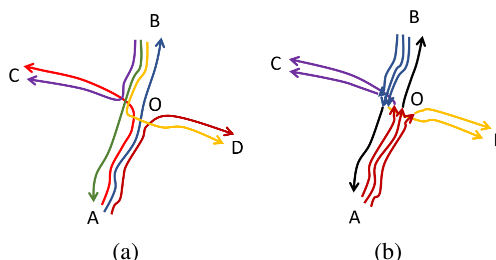

Fig. 1: (a) Six trajectories moving in the xy-plane and (b) 4 clusters (red, blue, orange and purple) and 2 outliers (black).

The problem of subtrajectory clustering is shown to be NP-Hard (cf. [1]). In addition, the objects to be clustered are not known beforehand (as in entire-trajectory – from now on – clustering algorithms), but have to be identified through a trajectory segmentation procedure. Efforts that try to deal with this problem in a centralized way do exist [1], [9], [19], however, applying these centralized algorithms over massive data in a scalable way is far from straightforward. This calls for parallel and distributed algorithms that address the scalability requirements. In this context, one challenge is how to partition the data in such a way so that each node can perform its computation independently, thus minimizing the communication cost between nodes, which is a cost that can turn out to be a serious bottleneck. Another challenge, related to partitioning, is how to achieve load balancing, in order to balance the load fairly between the different nodes. Yet another challenge is to minimize the iterations of data processing, which are typically required in clustering algorithms. Interestingly, there have been some recent efforts towards mining mobility data in a distributed way, such as mining co-movement patterns [5], identifying frequent patterns [20] or adapting already existing distributed solutions to trajectory data [3], yet no approach for distributed subtrajectory clustering exists as of now.

Motivated by these limitations, we study the *Distributed Subtrajectory Clustering (*DSC*)* problem, which has not been addressed yet in a scalable and efficient way. Moreover, salient

features of our approach include: (a) the discovery of clusters of subtrajectories, instead of whole trajectories, (b) spatio-temporal clustering, instead of spatial only, and (c) support of trajectories with variable sampling rate, length and with temporal displacement.

Our main contributions are the following:

- We formally define the problem of *Distributed Subtrajectory Clustering*, investigate its properties and discuss the main challenges.
- We propose two neighborhood-aware trajectory segmentation algorithms, which are tailored to the *DSC* problem, covering different application requirements.
- We design an efficient and scalable solution for the *DSC* problem.
- We perform an extensive experimental study, where the performance and the effectiveness of the proposed algorithms is evaluated by using two large, real trajectory datasets from different domains (urban and maritime). The merits of our solution are demonstrated with respect to two state of the art subtrajectory clustering algorithms, [19] and [9].

The rest of the paper is organized as follows. In Section II we provide an overview of the relevant literature. Subsequently, in Section III we introduce the *DSC* problem, in Section IV we present our proposed solution and in Section V, we present the results of our experimental study. We conclude the paper in Section VI.

II. Related Work

In recent years, an increased research interest has been observed in knowledge discovery out of mobility data. Towards this direction, several *co-movement pattern discovery* and *trajectory clustering* methods, which are directly related to our work, have been proposed.

Co-movement patterns. One of the first approaches for identifying such collective mobility behavior is the so-called flock pattern [8], [24]. Inspired by this, a less "strict" definition of flocks was proposed in [7] where the notion of a moving cluster was introduced. There are several related works that emerged from the above ideas, like the approaches of convoys [6], [14], swarms [11], platoons [10], traveling companion [23] and gathering pattern [26]. However, all of the aforementioned approaches are centralized and cannot scale to massive datasets. In this direction, the problem of efficient convoy discovery was studied both in centralized [14] and distributed environment by employing the MapReduce programming model [13]. An approach that defines a new generalized mobility pattern is presented in [5]. In more detail, the general co-movement pattern (GCMP), is proposed, which models various co-movement patterns in a unified way and is deployed on a modern distributed platform (i.e., Apache Spark) to tackle the scalability issue. Even though all of these approaches provide explicit definitions of several mined patterns, their main limitation is that they search for specific collective behaviors, defined by respective parameters. Nevertheless, none of the above techniques tackles the subtrajectory clustering problem.

Trajectory clustering. Most of the aforementioned approaches operate at specific predefined temporal "snapshots" of the dataset, thus ignoring the route of each moving object between these "snapshots". Another line of research, tries to discover groups of either entire or portions of trajectories considering their routes. A typical strategy in dealing with trajectory clustering is to transform trajectories to a multi-dimensional space and then apply well-known clustering algorithms such as OPTICS [2] and DBSCAN [4]. Alternatively, another approach is to define an appropriate similarity function and embed it to an extensible clustering algorithm. In this direction, there are several approaches whose goal is to group whole trajectories, including T-OPTICS [12], that incorporates a trajectory similarity function into the OPTICS [2] algorithm. CenTR-I-FCM [17], a variant of Fuzzy C-means, proposes a specialized similarity function that aims to tackle the inherent uncertainty of trajectory data. Nevertheless, trajectory clustering is a computationally intensive operation and centralized solutions cannot scale to massive datasets. In this context, [3] introduces a scalable GPU-based trajectory clustering approach which is based on OPTICS [2]. Moreover, [20] attempts to identify frequent movement patterns from the trajectories of moving objects. More specifically, they propose a MapReduce approach by employing quadtree-based hierarchical grid in order to discover complex patterns of different granularity.

Subtrajectory clustering. Nonetheless, discovering clusters of complete trajectories can overlook significant patterns that might exist only for portions of their lifespan. To deal with this, another line of research has emerged, that of *Subtrajectory Clustering*. The predominant approach here is TraClus [9], a partition-and-group framework for clustering 2D moving objects (i.e. the time dimension is ignored) that enables the discovery of common subtrajectories. The algorithm first partitions trajectories to directed segments (i.e., subtrajectories) whenever the shape of a trajectory changes significantly, by employing the minimum description length (MDL) principle. Subsequently, the resulting subtrajectories are clustered by employing a modified version of the DBSCAN algorithm, which is applicable to directed segments. Finally, for each identified cluster the algorithm calculates a "fictional" representative trajectory that best describes the corresponding cluster. A more recent approach to the problem of subtrajectory clustering, is S^2T-Clustering [19], where the goal is to partition trajectories into subtrajectories and then form groups of similar ones, while, at the same time, separate the ones that fit into no group, called outliers. It consists of two phases: a Neighborhood-aware Trajectory Segmentation (*NaTS*) phase and a Sampling, Clustering and Outlier (*SaCO*) detection phase. In *NaTS* the trajectories are split to subtrajectories by applying a voting and segmentation process that detects homogenized subtrajectories w.r.t. the density of their neighborhood. In *SaCO* the most representative subtrajectories are selected to serve as the seeds of the clusters, around which the clusters are formed (also, the outliers are isolated). A slightly different approach is presented

in QuT-Clustering [18] and [22], where the goal is, given a temporal period of interest W, to efficiently retrieve the clusters and outliers at subtrajectory level, that temporally intersect W. In order to achieve this, a hierarchical structure, called ReTraTree (for Representative Trajectory Tree) that effectively indexes a dataset for subtrajectory clustering purposes, is built and utilized. An alternative viewpoint to the problem of subtrajectory clustering is presented in [1], where the goal is to identify "common" portions between trajectories, w.r.t. some constraints and/or objectives, cluster these "common" subtrajectories and represent each cluster as a pathlet, which is a point sequence that is not necessarily a subsequence of an actual trajectory. A pathlet can be viewed as a portion of a path that is traversed by many trajectories. In order to solve this problem, the authors in [1] prove that this problem is NP-Hard and propose some approximation algorithms with theoretical guarantees, concerning the quality of the solution and the running time. Similarly, in [27] the goal is to identify corridors, which are frequent routes traversed by a significant number of moving objects. As already mentioned, all of the above subtrajectory clustering approaches are centralized and cannot scale to the size of today's trajectory data.

III. Problem Formulation

Given a set D of moving object trajectories, a trajectory $r \in D$ is a sequence of timestamped locations $\{r_1, \ldots, r_N\}$. Each $r_i = (loc_i, t_i)$ represents the i-th sampled point, $i \in 1, \ldots, N$ of trajectory r, where N denotes the length of r (i.e. the number of points it consists of). Moreover, loc_i denotes the spatial location (2D or 3D) and t_i the time coordinate of point r_i, respectively. A subtrajectory $r_{i,j}$ is a sub-sequence $\{r_i, \ldots, r_j\}$ of r which represents the movement of the object between t_i and t_j where $i < j$ and $i, j \in 1, \ldots, N$. Let $d_s(r_i, s_j)$ denote the spatial distance between two points $r_i \in r$, $s_j \in s$. In our case we adopted the Euclidean distance, however, other metric distance functions might be applied. Also, let $d_t(r_i, s_j)$ denote the temporal distance, defined as $|r_i.t - s_j.t|$. Furthermore, let Δt_r symbolize the duration of trajectory r (similarly for subtrajectories).

A. Similarity between (sub)trajectories

Subtrajectory clustering relies on the use of a similarity function between subtrajectories. Although various similarity measures have been defined in the literature, our choice of similarity function is motivated by the following (desired) requirements:

Variable sampling rate and lack of alignment. We make the realistic assumption that the trajectories do not have a fixed sampling rate and that different trajectories might not report their position at the same timestamp.

Variable trajectory length. We also assume that different trajectories might have different length (i.e. number of samples). This specification excludes euclidean-based similarity measures which deal with trajectories of equal length.

Temporal displacement. A property that a desired similarity measure for (sub)trajectory clustering should hold, is to allow trajectories that have some temporal displacement to participate to the same cluster.

Symmetry. Given a pair of (sub)trajectories r and s, an appropriate similarity measure between r and s should have the property of symmetry (i.e. $Sim(r, s)$=$Sim(s, r)$).

Efficiency. The computation of the similarity should be efficient enough in order to be able to deal with massive volumes of data, without compromising the quality of the results.

In order to meet with the aforementioned specifications we utilize the Longest Common Subsequence (LCSS) for trajectories, as defined in [25]. However, other trajectory similarity functions, which meet with the specifications set, are also applicable. More specifically, the LCSS utilizes two parameters, the parameter ϵ_t indicating the temporal range wherein the method searches to match a specific point, and the ϵ_{sp} parameter which is a distance threshold to indicate whether two points match or not. Hence, the similarity between two (sub)trajectories r and s is defined as:

$$Sim(r, s) = \frac{LCSS_{\epsilon_t, \epsilon_{sp}}(r, s)}{min(|r|, |s|)} \tag{1}$$

where $|r|$ ($|s|$) is the length of r (s respectively). Moreover, it holds that $Sim(r, s) = Sim(s, r)$.

However, LCSS returns the length of the longest common subsequence, which means that for a given point $r_i \in r$ that is matched with a specific point $s_j \in s$ the LCSS will consider the similarity between r_i and s_j as 1, regardless of their actual distance $d_s(r_i, s_j)$, which could vary from 0 to ϵ_{sp}. Put differently, LCSS considers as equally similar all the points that exist within an ϵ_{sp} range from r, which is a fact that might compromise the quality of the clustering results. Ideally, given two matching points $r_i \in r$ and $s_j \in s$, s_j (r_i, respectively) should contribute to $LCSS_{\epsilon_t, \epsilon_{sp}}(r, s)$, proportionally to the distance $d_s(r_i, s_j)$. For this reason, we propose a "weighted" LCSS similarity between trajectories, that incorporates the aforementioned distance proportionality. In more detail, for each discovered longest common subsequence the similarity is defined as:

$$Sim(r, s) = \frac{\sum_{k=1}^{min(|r|,|s|)} (1 - \frac{d_s(r_k, s_k)}{\epsilon_{sp}})}{min(|r|, |s|)} \tag{2}$$

where (r_k, s_k) is a pair of matched points.

B. A Closer Look to the Subtrajectory Clustering Problem

Our approach to subtrajectory clustering splits the problem in three steps. The first step is to retrieve for each trajectory $r \in D$, all the moving objects, with their respective portion of movement, that moved close enough in space and time with r, for at least some time duration. Actually, this first step is a well-defined problem in the literature of mobility data management, known as *subtrajectory join*, and more specifically the case of self-join. In detail, the subtrajectory join will return for each pair of (sub)trajectories, all the common subsequencies that have at least some time duration, which

are actually candidates for the longest common subsequence. Formally:

Problem 1. ***(Subtrajectory Join)*** *Given a temporal tolerance ϵ_t, a spatial threshold ϵ_{sp} and a time duration δt, retrieve all pairs of subtrajectories $(r', s') \in D$ such that: (a) for each pair $\Delta t_{r'}, \Delta t_{s'} \geq \delta t$, (b) $\forall r_i \in r'$ there exists at least one $s_j \in s'$ so that $d_s(r_i, s_j) \leq \epsilon_{sp}$ and $d_t(r_i, s_j) \leq \epsilon_t$, and (c) $\forall s_j \in s'$ there exist at least one $r_i \in r'$ so that $d_s(s_j, r_i) \leq \epsilon_{sp}$ and $d_t(s_j, r_i) \leq \epsilon_t$.*

The second step takes as input the result of the first step, which is actually a trajectory r and its neighboring trajectories and aims at segmenting each $r \in D$ into a set of subtrajectories. The way that a trajectory is segmented into subtrajectories is neighbourhood-aware, meaning that a trajectory will be segmented every time its neighbourhood changes significantly. Returning to Example 1, trajectory $A \rightarrow D$ should be segmented to $A \rightarrow O$ and $O \rightarrow D$, since at O the cardinality and the composition of its neighbourhood changes significantly. The problem of trajectory segmentation can now be formulated as follows.

Problem 2. ***(Trajectory Segmentation)*** *Given a trajectory r, identify the set of timestamps CP (cutting points), where the density (or alternatively the composition) of the neighborhood of r changes significantly. Then according to CP, r is partitioned to a set of subtrajectories $\{r'_1, \ldots, r'_M\}$, where $M = |CP| + 1$ is the number of subtrajectories for a given trajectory r, such that $r = \bigcup_{k=1}^{M} r'_k$ and $k \in [1, M]$.*

Given the output of Problem 1, applying a trajectory segmentation algorithm for the trajectories D will result in a new set of subtrajectories D'. The third step takes as input D' and the goal is to create clusters (whose cardinality is unknown) of similar subtrajectories and at the same time identify subtrajectories that are significantly dissimilar from the others (outliers). More specifically, let $C = \{C_1, \ldots, C_K\}$ denote the clustering, where K is the number of clusters, and for every pair of clusters C_i and C_j, with $i, j \in [1, K]$, it holds that $C_i \cap C_j = \emptyset$. Now, let us assume that each cluster $C_i \in C$ is represented by one subtrajectory $R_i \in C_i$, called *Representative*. Furthermore, let R denote the set of all representatives. Actually, the problem of clustering is to discover clusters of objects such that the intra-cluster similarity is maximized and the inter-cluster similarity is minimized. Therefore, if we ensure that the similarity between the representatives is zero, then the problem of subtrajectory clustering can be formulated as an optimization problem as follows.

Problem 3. ***(Subtrajectory Clustering and Outlier Detection)*** *Given a set of subtrajectories D', partition D' into a set of clusters C and a set of outliers O, where $D' = C \cup O$, in such a way so that the Sum of Similarity between Cluster members and cluster Representatives (*SSCR*) is maximized:*

$$SSCR = \sum_{\forall R_i \in R} \sum_{\forall r'_j \in C_i} Sim(R_i, r'_j) \tag{3}$$

However, trying to solve Problem 3 by maximizing Equation (3) is not trivial, since the problem to segment trajectories to subtrajectories, select the set of representatives R and its cardinality $|R|$ that maximizes Equation (3), has combinatorial complexity.

In this paper, we address the challenging problem of subtrajectory clustering in a distributed setting, where the dataset D is distributed across different nodes, and centralized processing is prohibitively expensive.

Problem 4. ***(Distributed Subtrajectory Clustering)*** *Given a distributed set of trajectories, $D = \cup_{i=1}^{P} D_i$, where P is the number of partitions of D, perform the subtrajectory clustering task in a parallel manner.*

Actually, Problem 4 can be broken down to solving Problems 1, 2 and 3 (in that order) in a parallel/distributed way. In the following, we adopt this approach and outline a solution that is based on MapReduce.

IV. Problem Solution

A. Overview

An overview of our approach is presented in Algorithm 1.

Algorithm 1 $DSC(D)$

1: **Input:** D
2: **Output:** set C of clusters, set O of outliers
3: **Preprocessing:** *Repartition* D;
4: **for each** partition $D_i \in \cup_{i=1}^{P} D_i$ **do**
5: **perform** *Point-level Join*;
6: **group by** *Trajectory*;
7: **for each** Trajectory $r \in D$ **do**
8: **perform** *Subtrajectory Join*; *– Sect. IV-B*
9: **perform** *Trajectory Segmentation*; *– Sect. IV-C*
10: **group by** D_i;
11: **for each** subtrajectory $r' \in D_i$ **do**
12: **calculate** $Sim(r', s') \; \forall s' \in D_i$; *– Sect. IV-C*
13: **perform** $Clustering$; *– Sect. IV-D*
14: **perform** *Refine Results*;
15: **return** C and O;

Initially, we *Repartition* the data into P equi-sized, temporally-sorted temporal partitions (files), which are going to be used as input for the join algorithm in order to perform the subtrajectory join in a distributed way (line 3). Note that this is actually a preprocessing step that only needs to take place once for each dataset D. However, it is essential as it enables load balancing, by addressing the issue of temporal skewness in the input data. Subsequently, for each partition $D_i \in \cup_{i=1}^{P} D_i$ and for each trajectory we discover parts of other trajectories that moved close enough in space an time (line 5). Successively, we group by trajectory in order to perform the subtrajectory join (line 8). At this phase, since our data is already grouped by trajectory, we also perform trajectory segmentation in order to split each trajectory to subtrajectories (line 9). In turn, we utilize the temporal partitions created during the *Repartition* phase and re-group the data by temporal partition. For each $D_i \in \cup_{i=1}^{P} D_i$ we calculate the

similarity between subtrajectories and perform the clustering procedure (line 12). At this point we should mention that if a subtrajectory intersects the borders of multiple partitions, then it is replicated in all of them. This will result in having duplicate and possibly contradicting results. For this reason, as a final step, we treat this case by utilizing the *Refine Results* procedure (line 14). Finally, a set C of clusters and a set O of outliers are produced.

B. Distributed Subtrajectory Join

As already mentioned, the first step is to perform the subtrajectory join in a distributed way. For this reason, we exploit the work presented in [21], called *DTJ*, which introduces an efficient and highly scalable approach to deal with Problem 1, by means of MapReduce. More specifically, *DTJ* is comprised of a *Repartitioning* phase and a *Query* phase. The *Repartitioning* phase is a preprocessing step that takes place only once and it is independent of the actual parameters of the problem, namely ϵ_{sp}, ϵ_t, and δt. The idea is to construct an equi-depth histogram based on the temporal dimension, where each of the M bins contain the same number of points and the borders of each bin correspond to a temporal interval $[t_i, t_j)$. The histogram is constructed by taking a sample of the input data Then, the input data is partitioned to processing tasks based on the temporal intervals of the histogram bins. This guarantees temporal locality in each partition, as well as equi-sized partitions, thus balancing the load fairly.

In the *Query* phase, the actual join processing takes place. It consists of two steps, the *Join* (line 5) and the *Refine* (line 8) step, which are implemented as a *Map* and a *Reduce* function respectively. The output of this MapReduce job is for each trajectory $r \in D$ all the moving objects, with their respective portion of movement, that moved close enough in space and time for at least some time duration. In more detail, the output of *DTJ* is per trajectory (i.e reference trajectory) and the tuples are of the form $< refTrajPoint_i, \{MatchingPoints\} >$, where $refTrajPoint_i$ is the i-th point of the reference trajectory, with $i \in [1, N]$ and $MatchingPoints$ is a list of points of other trajectories that have been identified as join results by the *DTJ* query. In Figure 2, the *DTJ* query corresponds to Job 1 until the Refine() procedure.

The complexity of the *Join* algorithm is $O(|D| log_2 Q)$, with Q being the average number of points per spatial index partition and $Q << |D|$. The complexity of the *Refine* algorithm is $O(T \cdot SW \cdot dt \cdot l)$, where T is the average number of points per trajectory, SW is the average number of points contained in a $\delta t + 2\epsilon_t$ window, dt the average number of points contained in a δt window and l is average the size of the $MatchingPoints$ list. For more technical details about the algorithms involved in *DTJ*, their complexity and an extensive experimental study, we refer to [21].

C. Distributed Trajectory Segmentation

The *Trajectory Segmentation* algorithm (TSA) takes as input a single trajectory, along with information about its neighborhood, and partitions it to a set of subtrajectories. In this paper, we propose two alternative segmentation algorithms. The first algorithm, coined TSA_1, identifies the beginning of a new subtrajectory whenever the density of its neighborhood changes significantly. Such a segmentation algorithm is reminiscent of the flock definition [8], where the identified groups need to be composed of at least m objects. For this purpose, we use the concept of *voting* as a measure of density of the surrounding area of a trajectory. For a given point r_i and any trajectory s, the voting $V(r_i)$ is defined as:

$$V(r_i) = \sum_{\forall s \in D} \frac{d_s(r_i, s_k)}{\epsilon_{sp}} \tag{4}$$

where, s_k is the matching point of s with r_i, as emitted by the subtrajectory join procedure. For a trajectory r that consists of N points $\{r_1, \ldots, r_N\}$, we compute its normalized voting vector $\overline{V}(r)$ as follows:

$$\overline{V}(r)[] = \{\frac{V(r_1)}{\max_{i=1}^{N} V(r_i)}, \ldots, \frac{V(r_N)}{\max_{i=1}^{N} V(r_i)}\} \tag{5}$$

Finally, the voting of a trajectory (or subtrajectory) is defined as:

$$V(r) = \frac{1}{N} \sum_{i=1}^{N} V(r_i) \tag{6}$$

The second segmentation algorithm, coined TSA_2, identifies the beginning of a new subtrajectory whenever the composition of its neighborhood changes substantially. This segmentation algorithm is reminiscent of the moving cluster definition [7], where the identified groups need to share a sufficient number of common objects. Such an algorithm does not take as input the $\overline{V}(r)[]$ but instead, for each point $r_i \in r$, it takes as input a list $L(r_i)[]$ of the trajectory ids that have been produced as output by the *DTJ* procedure.

The following example explains intuitively the difference between the two segmentation algorithms.

Example 2. *Consider the example of Figure 3(a) that illustrates five trajectories: $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow D$, $C \rightarrow B$ and $D \rightarrow B$. Figures 3(b) and (c) depict the result of TSA_1 and TSA_2, respectively. In more detail, we can observe that both TSA_1 and TSA_2 segmented trajectory $A \rightarrow D$ to subtrajectories $A \rightarrow O$ and $O \rightarrow D$, due to the fact that after O, both the density and the composition of the neighborhood changes. The same holds for trajectories $A \rightarrow C$, $C \rightarrow B$ and $D \rightarrow B$, which are segmented to subtrajectories $A \rightarrow O$, $O \rightarrow C$, $C \rightarrow O$, $O \rightarrow B$, $D \rightarrow O$ and $O \rightarrow B$. However, when it comes to trajectory $A \rightarrow B$, we can observe that while TSA_2 segments it to subtrajectories $A \rightarrow O$ and $O \rightarrow B$, TSA_1 does not perform any segmentation. This is due to the fact that, after O, even though the density of the neighborhood remains the same (i.e. 3 moving objects), the composition of the neighborhood changes completely.*

Both segmentation algorithms share a common methodology, which employs two consecutive sliding windows W_1 and W_2 of size w (i.e. w samples) to estimate the point $r_i \in CP$ (cutting point) where the "difference" between

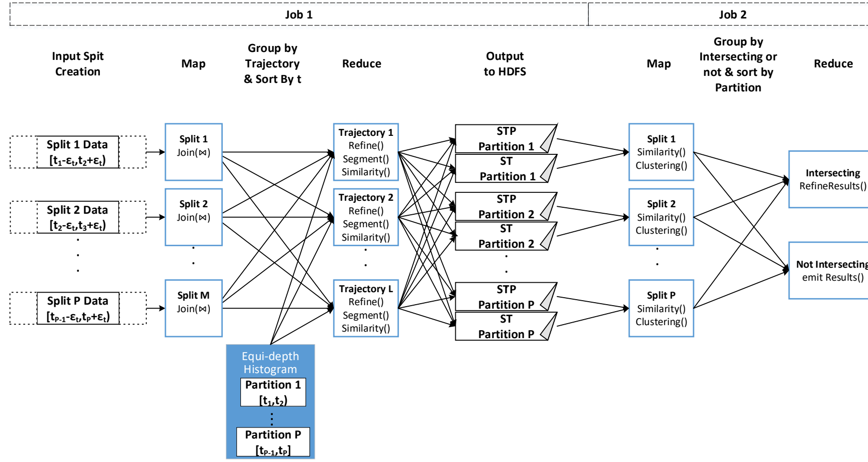

Fig. 2: The *DSC* algorithm. (Job 1) *DTJ* and *Trajectory Segmentation* and (Job 2) *Clustering* and *Refine Results*.

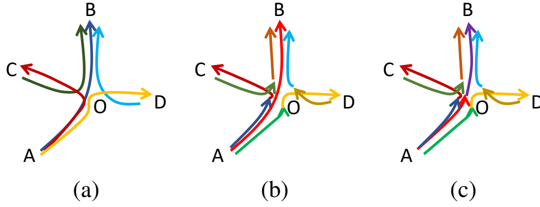

Fig. 3: (a) Five trajectories $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow D$, $C \rightarrow B$ and $D \rightarrow B$, (b) TSA_1 segmentation, (c) TSA_2 segmentation

the two windows is maximized. This methodology has been successfully applied in the past on signal segmentation [15], [16].

Trajectory segmentation. Since the output of the *DTJ* algorithm is per trajectory, it is straightforward to give it as input to *TSA* which operates at the level of a trajectory. Moreover, the segmentation is performed in an embarrassingly parallel way, due to the fact that each trajectory can be processed by a different reduce task independently from others, as depicted in Figure 2. In more detail, for a given trajectory $r \in D$, TSA_1 first calculates the normalized voting vector $\overline{V}(r)[]$ and then performs the segmentation by utilizing it. Apart from $\overline{V}(r)[]$, the input of the *TSA* algorithm is two additional parameters: w and τ. The output is a vector $CP[]$, which keeps the starting position of each subtrajectory of r.

Algorithm 2 $TSA_1(\overline{V}(r)[], w, \tau)$

1: **Input:** $\overline{V}(r)[], w, \tau$
2: **Output:** $CP[]$
3: $1 \rightarrow CP[]$;
4: **for** n = w+1 ...N-w-1 **do**
5: $\quad m_1 = \frac{1}{w} \sum_{i=n-w}^{n-1} \overline{V}(r)[i]$;
6: $\quad m_2 = \frac{1}{w} \sum_{i=n}^{n+w-1} \overline{V}(r)[i]$;
7: $\quad d[n] = |m_1 - m_2|$;
8: $\quad d_{max} = \max_{i=w+1}^{N-w-1} d[i]$;
9: $\quad$ **if** $d[n] > \tau \wedge d[n] >= d_{max}$ **then**
10: $\quad\quad n \rightarrow CP[]$;

In more detail, as presented in Algorithm 2, two consecutive sliding windows of size w are created over $\overline{V}(r)[]$, named W_1 and W_2 (line 4). These sliding windows move forward in time until $\overline{V}(r)[]$ is traversed. Here, N is the number of points of trajectory $r \in D$. Then, for each window, the average normalized voting is computed (lines 5-6) and their absolute difference is stored in $d[]$, which is an array that stores all the differences between the sliding windows (line 7). Subsequently, we examine whether the current difference $d[n]$ is larger than the maximum difference d_{max} and we update d_{max} accordingly (line 8). Finally, if the difference $d[n]$ is higher than a threshold τ and is locally maximized, then, at that point, we segment the trajectory and we store the starting position of the new subtrajectory to $CP[]$ (lines 9-10).

On the other hand, the input of TSA_2 is a list of lists $L(r)[]$ for each $r \in D$. Similarly, two consecutive sliding windows W_1 and W_2 of size w are created. Then, for each window, the union of lists is computed and stored in l_1 and l_2, respectively. Successively, the Jaccard dissimilarity between l_1 and l_2 is computed and is stored to $d[]$, which is an array that stores all the similarities between the sliding windows. From then on, the algorithm is identical to TSA_1.

The complexity of both TSA_1 and TSA_2 algorithms is $O(l \cdot |T|)$, where l is average the size of the "matching" list and $|T|$ is the average number of points per trajectory.

Similar subtrajectories. The next step is to calculate the similarity between all the pairs of subtrajectories, using Equation 2. This cannot be done completely after the segmentation at the *Reducer* phase of Job 1, illustrated in Figure 2, because at that point each reduce function has information only about the segmentation of the reference trajectory to subtrajectories. For this reason, at this point we cannot calculate the denominator of Equation 2. However, for each subtrajectory $r' \in r$, where r is the reference trajectory, we can calculate the similarity between the matching points (enumerator of Equation 2).

In more detail the output of each reduce function (Job 1 Figure 2) is a relation, called *STP*, which holds a set of key-value pairs of the form $< (r'.ID, s.ID), \{(s_f.t, Sim(s_f, r'))$

$\ldots\ (s_l.t, Sim(s_l, r'))\} >$, where s_f, s_l are the temporal first and last point, respectively, of trajectory s that "matches" with subtrajectory r'. Moreover, in a separate relation, coined *ST*, we hold some extra information for each subtrajectory. More specifically, the tuples of ST are key-value pairs, where the key is the subtrajectory identifier $< ID >$ and the value is of the form $< t_s, t_e, V, Card >$, where t_s (t_e) is the starting time (ending time, respectively) of the subtrajectory, V is the voting and $Card$ is the number of points which constitute the specific subtrajectory. Due to the fact that these two relations can be pretty large, we need to partition them into smaller files. In order to achieve this, we broadcast the load balanced temporal partitions that were created during the *Repartitioning* phase of *DTJ*. As illustrated in Figure 2, each reducer loads these partitions and assigns each subtrajectory (tuple of *ST* and *STP*) to all the partitions with which it temporally intersects. Subsequently, the tuples are grouped by temporal partition and each group is fed to a Mapper.

At this point, each *Mapper* has now all the information needed to calculate the similarity between all the pairs of subtrajectories (Equation 2), for each temporal partition separately. The similarity between subtrajectories is output in a new relation, called *SP*. Each tuple of this relation holds information about a subtrajectory r' and its similarity with all the other subtrajectories, whenever this similarity is larger than zero. More specifically, *SP* contains a set of key-value pairs where the key is the ID of the subtrajectory ($r'.ID$) and the value is a list $AdjLst$ containing elements of the form $(s'.ID, Sim)$, where s' is a subtrajectory for which it holds that $Sim(r', s') > 0$.

D. Distributed Clustering

Clustering. After having calculated the similarity between all pairs of subtrajectories for each temporal partition, we can proceed to the actual clustering and outlier detection procedure. The intuition behind the proposed solution to Problem 3 is to select as cluster representatives, highly voted subtrajectories (Equation 6) that have zero similarity with the already selected representatives $R_i \in R$, thus addressing the inter-cluster distance minimization. Then, we assign each subtrajectory r'_k to the R_i (and hence C_i) with which it has the maximum similarity $Sim(r'_k, R_i)$.

The input of the clustering algorithm is SP, ST and parameters k and α and the output is the set of clusters C and the set of outliers O. More specifically, k is a threshold for setting a lower bound on the voting of a representative. This prevents the algorithm from identifying clusters with small support. Parameter α is a similarity threshold used to assign subtrajectories to cluster representatives. It ensures that a subtrajectory assigned to a cluster has sufficient similarity with the representative of the cluster. This actually poses a lower bound to the average distance between the representatives and the cluster members and, consequently, guarantees a minimum quality in the identified clusters (intra-cluster distance).

To begin with, we want to traverse the subtrajectories by their voting, in descending order (i.e. highly voted subtrajectories first). In order to achieve this, we need to sort ST by V (line 3). Subsequently, for each subtrajectory $st \in ST$ we examine whether it is already assigned to cluster (line 5). If st is not assigned to any cluster and the voting of st is less than k, then we add st to the outliers set (line 21). Otherwise, we create a new cluster and consider st as the representative (lines 6-7). Successively, we consult relation SP and retrieve the adjacency list of st (line 8). Then, for each element l that belongs to the adjacency list of st, we examine if it is assigned to any cluster. If not, we investigate whether the similarity between l and st is greater or equal than the similarity threshold α. If not, we add l to the outlier set O, otherwise we assign it to the cluster led by st and remove it from the outliers O, in case $l \in O$ (lines 9-13). If l is assigned to a cluster, we examine whether the similarity of l with st is greater than the similarity with the representative of the cluster that l is currently assigned. If this is the case, then we remove l from the current cluster and assign it to the cluster led by st (lines 17-19). Finally, we concatenate C with R (line 22) so as to return, except from the outlier set O, both cluster members and representatives.

Algorithm 3 $Clustering(SP, ST, k, \alpha)$

```
1:  Input: SP, ST, k, α
2:  Output: set C of clusters, set O of outliers
3:  sort ST by V in descending order;
4:  for each element st ∈ ST do
5:    if st ∉ R then
6:      if st.V ≥ k then
7:        st → R;
8:        for each element l ∈ st.AdjLst do
9:          if l ∉ C then
10:           if Sim(l, st) ≥ α then
11:             l → C(st);
12:             if l ∈ O then
13:               O = O − l;
14:           else
15:             O = O ∪ l
16:         else
17:           if Sim(l, st) > Sim(l, R(l)) then
18:             C(R(l)) = C(R(l)) − l;
19:             l → C(st);
20:     else
21:       O = O ∪ st;
22: C = C ∪ R
```

Refinement of Results. At this point we successfully accomplished to deal with Problem 3 for each temporal partition. However, this might result in having duplicates due to the fact that each subtrajectory that temporally intersects multiple partitions is replicated to each one of them. The actual problem that lies here is not the duplicate elimination problem itself but the fact that the result for such a subtrajectory might be contradicting in different partitions. In more detail, for each partition, the clustering procedure will decide whether a subtrajectory is a *Representative* (R), a *Cluster Member* (C) or an *Outlier* (O). Hence, for each intersecting subtrajectory

q and for each pair of consecutive partitions (i, j) with which q intersects, q can have the following pairs of states: (a) O-O, (b) R-R, (c) C-C, (d) R-C (C-R), (e) R-O (O-R) and (f) C-O (O-C).

In order to implement the above procedure we need to have all the information concerning the intersecting subtrajectories (C and O) for all the Partitions sorted in time. To do this, we group the trajectories according to whether they are intersecting or not. As illustrated in Figure 2, the non-intersecting are emitted, since they are not affected, while the intersecting subtrajectories get sorted by partition. Hence, a *Reducer* will receive all the required information to make the appropriate decisions. In more detail, we sweep through the temporal dimension and for each pair of consecutive partitions we make the appropriate decisions.

More specifically, in case of (a), q is marked as outlier in both partitions, hence, we only need to eliminate duplicates. In case of (b), the two clusters are "merged", since all of the subtrajectories that belong to them are similar "enough" with q, which is the representative of both clusters. In case of (c), let us assume that q belongs to cluster $C_i(R(q))$ in Partition i and $C_{i+1}(R(q))$ in Partition $i+1$. Then, q is assigned to the cluster with which it has the largest similarity with its representative. In case of (d), q remains to be a cluster representative and is removed from the cluster C in which it is a member. Finally, in case of (e) and (f), q is removed from O.

The complexity of the $Clustering$ algorithm is $O(|ST| \cdot log|ST| + |ST| \cdot |L|)$, with $|ST|$ being the number of subtrajectories, $|L|$ the average size of the adjacency list $AdjLst$ and $|ST| \cdot log|ST|$ is the sorting cost. Here, we should mention that $|ST| << |D|$. Furthermore, ST and SP are implemented as HashMaps, hence key search has an $O(1)$ time complexity. The complexity of the $RefineResults$ algorithm is $O(M \cdot |P| \cdot |I|)$, where M is the number of temporal partitions, $|P|$ is the average number of intersecting subtrajectories per partition and I is the average size of the intersection. We should mention, here, that the intersection between two consecutive partitions is performed in linear time by utilizing HashSets sets.

V. Experimental Study

The experiments were conducted in a 49 node Hadoop 2.7.2 cluster, provided by *okeanos*[1]. The master node consists of 8 CPU cores, 8 GB of RAM and 60 GB of HDD while each slave node is comprised of 4 CPU cores, 4 GB of RAM and 60 GB of HDD. Our configuration enables us to lanch up to 192 tasks simultaneously. For our experimental study, we employed two real datasets from two different domains (urban and the maritime). In more detail, the first one, named SIS[2], is a 27GB proprietary insurance dataset of moving objects around Rome and Tuscany area, that contains approximately 2.2×10^7 trajectories that correspond to 7.2×10^8 points. The second one, coined Brest[3], is a 650MB publicly available AIS dataset of vessels moving in the wider Brest area, consisting approximately of 3.65×10^5 trajectories that correspond 17×10^6 points.

TABLE I: Parameters and default values (in bold)

Parameter	Values				
	(i)	(ii)	(iii)	(iv)	(v)
ϵ_{sp} (%)	10%	15%	**20%**	25%	30%
ϵ_t (%), δt (%)	0%	25%	**50%**	75%	100%
w	10	15	**20**	25	30
τ	0.2	0.4	**0.6**	0.8	1
α (in σ), k (in σ)	-2	-1	**0**	1	2

Our experimental methodology is as follows: Initially, in Section V-B we compare our solution with two state of the art subtrajectory clustering methods (TraClus [9] and S^{2}T-Clustering [19]). Subsequently, in Section V-C, we study the scalability of our solution by varying (a) the dataset size, and (b) the number of cluster nodes. Finally, in Section V-D, we perform a sensitivity analysis in order to evaluate the effect of setting different values to the parameters, in terms of execution time and quality. Table I shows the experimental setting, where we vary the following parameters: ϵ_{sp}, ϵ_t, δt, w, τ, α and k and measure their effect in the performance and the effectiveness of our algorithms. We should mention that the default segmentation algorithm in our experimental study is TSA_1.

A. Parameter Setting

Setting the different parameters for different datasets can turn out to be an arbitrary procedure, which, in turn, can jeopardise the quality of the clustering results. For this reason, we provide some simple rules for setting the parameters relatively to the dataset being clustered, that do not compromise the quality of the results. In more detail, ϵ_{sp} can be set as a percentage of the dataset diameter. This, however, can be problematic when dealing with datasets having large spatial variation in their density (e.g. ports in the maritime domain). For this reason, we utilized the partitioning provided by the spatial index (QuadTree) of *DTJ* and calculated ϵ_{sp} for each point, as a percentage of the diameter of the cell of the QuadTree to which it belongs. Moreover, ϵ_t and δt are calculated relatively to the average duration between two consecutive trajectory samples ($\approx$ 1200 sec for SIS and $\approx$ 950 sec for AIS Brest).

Concerning parameter w, small values on w can affect the robustness of the estimation, thus resulting to over-segmentation, while, large values of w can result to overlooking some cutting points due to the large window size. It has been observed that for $w \approx 20$ the robustness of the estimation is not affected and the size of the window is small enough so as not to overlook any cutting points. Concerning parameter τ, our experiments show that the best result in terms of quality is achieved for $\tau \approx 0.4$ Finally, the values of α and k can be set "around" the mean value of the similarity and the voting of the temporal partition, respectively, in terms of standard deviation. For more details about the effect, in terms of quality, of setting

[1]IAAS for the Greek Academic Community https://okeanos.grnet.gr/home/
[2]This private dataset was kindly provided by Gruppo Sistematica SpA
[3]https://zenodo.org/record/1167595\#.XKHTyaRRVPa

different values to the parameters of our solution, please refer to Section V-D

B. Comparison with related work

We compare *DSC* with two state of the art subtrajectory clustering algorithms, *S^2T-Clustering* and *TraClus*. The metric that we employ in order to evaluate the quality of the outcome of the clustering procedure is the well-known *RMSE* metric, which is actually a measure of intra-cluster distance between the representatives and the cluster members. Hence the larger the *RMSE*, the higher the intra-cluster distance and consequently the lower the quality of the clustering. In order to perform this experiment, we utilized the 20% of each dataset which was further partitioned in 4 portions (25%, 50%, 75%, 100%). This choice was necessary because the centralized implementations of *S^2T-Clustering* and *TraClus* could not scale with the full size of the datasets.

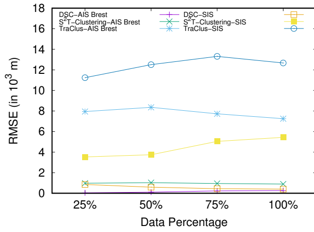

Fig. 4: Comparison of the RMSE metric between *DSC*, *S^2T-Clustering* and *TraClus*

As illustrated in Figure 4, *DSC* outperforms, in terms of *RMSE*, both *TraClus* and *S^2T-Clustering*. In more detail, *TraClus* presents the largest *RMSE* which is somehow anticipated, since the specific algorithm utilizes a density-based approach to cluster subtrajectories, which in turn, through cluster expansion, can lead to spatially extended clusters. On the other hand, *S^2T-Clustering* presents smaller *RMSE* than *TraClus*, due to the fact that it adopts a distance-based approach and discovers more compact clusters. However, *DSC* results in smaller *RMSE* than *S^2T-Clustering*, mostly due to the fact that in the latter, two trajectories might end-up in the same cluster even if they have small "matching portions". However, in *DSC* this "matching portions" should have a minimum (δt) duration.

C. Performance and Scalability

Initially, we vary the size of our datasets and measure the execution time of our algorithms. We show the impact of the individual steps: *Join*, *RSE*, *Clustering* and *RefineResult* using stacked bars. To study the effect of dataset size, we created 4 portions (20%, 40%, 60%, 80%) of the original datasets. *RSE* stands for the *Refine* and *Segmentation* procedure (Figure 2, Job 1, Reduce phase). As illustrated in Figures 5(a) and (b), as the size of the dataset increases, *DSC* appears to scale linearly. Subsequently, we keep the size of the datasets fixed (at 100%) and vary the number of nodes. As the number of nodes increases and the dataset size remains the same, it is expected that the execution time will decrease. Indeed, as depicted in Figures 5(c) and (d), as the number of nodes increases, *DSC* presents linear speedup. This linear behaviour, is somehow anticipated due to the fact that the *DSC* approach is dominated by *DTJ*, in terms of execution time, which presents linear speedup, as shown in [21].

Investigating further the performance, we can observe that the execution time of the whole procedure is dominated by the Join step (Figure 2, Job 1, Map phase), followed by *RSE*. Finally, the *Clustering* and the *RefineResults* step (Figure 2, Job 2) present very good performance, since the computationally intensive part of the similarity matrix calculation has already been done as part of the previous steps.

D. Sensitivity Analysis

In this section, we vary each parameter presented in Table I, while keeping the rest of them in their default value (bold), and we measure their effect in the execution time and the quality of the clustering results, in terms of *RMSE*. Figures 6(a) and (b) show that the parameters that appear to have a significant impact on execution time are ϵ_t and ϵ_{sp}. This is justified from the fact that these parameters actually affect significantly the complexity of the Join step (Figure 2, Job 1, Map phase), which is the dominant cost of *DSC*. Another parameter that seems to have a perceivable effect on the execution time, is δt, which in fact "filters" the results of *DTJ*, thus fewer data reach the next steps.

Regarding the quality of the clustering results, as illustrated in Figure 6(c) and (d), all the parameters seem to have an effect over it. In more detail, the larger the values of ϵ_t and ϵ_{sp}, the larger the *RMSE*. This behaviour is expected since we allow objects that are further away from a representative to participate to the same cluster. In contrast, as δt increases, the *RMSE* decreases, which is also anticipated since it sets a lower bound to the longest common subsequence. Furthermore, all the parameters that control the segmentation have the same effect on the *RMSE*, i.e. the smaller (in length) the subtrajectories, the smaller the *RMSE*. This shows that breaking trajectories to subtrajectories has a positive effect on the quality of the clustering and justifies the motivation of our work. Moreover, as α increases the *RMSE* decreases, since for small values of α, less similar objects are allowed to participate in a cluster. Finally, the larger the k the smaller the *RMSE*, since it disallows the identification of clusters with small support.

VI. Conclusions

In this paper, we addressed the *DSC* problem by building upon a scalable subtrajectory join query operator in order to tackle the problem in an efficient manner. Subsequently, we proposed two alternative trajectory segmentation algorithms. Finally, we proposed a distributed clustering algorithm where the clusters are identified in a parallel manner and get refined as a final step. Our experimental study was performed on two large real datasets of trajectories from the urban and the maritime domain. As for future work, we plan to extend our solution with properties of density-based clustering algorithms.

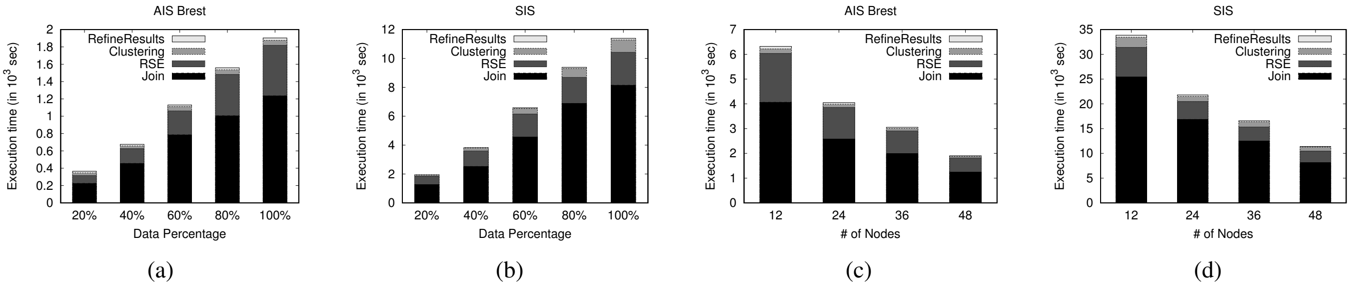

Fig. 5: Scalability by varying the size of (a) AIS Brest and (b) SIS and the number of nodes over (c)AIS Brest and (d) SIS

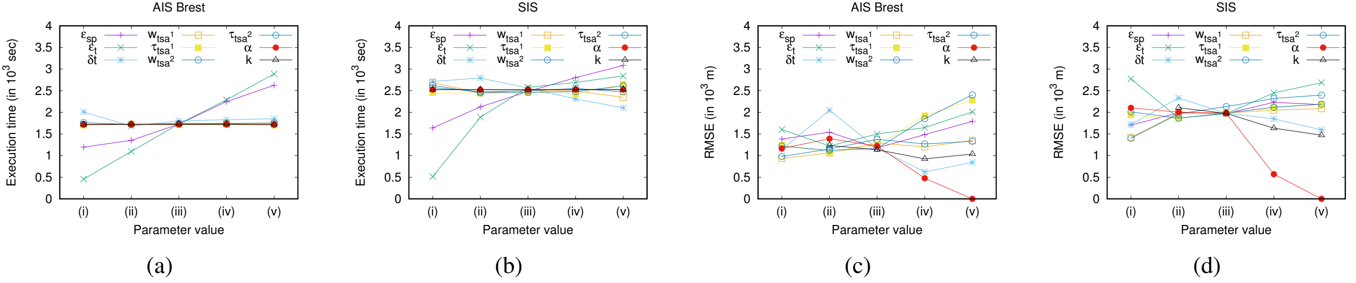

Fig. 6: Sensitivity in terms of execution time of (a) AIS Brest and (b) SIS and in terms of *RMSE* of (c) AIS Brest and (d) SIS

VII. Acknowledgements

This work was partially supported by projects Track&Know (grant agreement No 780754), MASTER (Marie Sklowdoska-Curie agreement N. 777695) and i4Sea (project code:T1EDK-3268), which have received funding from the EU Horizon 2020 R&I Programme and by the European Regional Development Fund of the EU and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH-CREATE-INNOVATE.

References

[1] P. K. Agarwal, K. Fox, K. Munagala, A. Nath, J. Pan, and E. Taylor. Subtrajectory clustering: Models and algorithms. In *PODS*, pages 75–87, 2018.

[2] M. Ankerst, M. M. Breunig, H. Kriegel, and J. Sander. OPTICS: ordering points to identify the clustering structure. In *SIGMOD*, pages 49–60, 1999.

[3] Z. Deng, Y. Hu, M. Zhu, X. Huang, and B. Du. A scalable and fast OPTICS for clustering trajectory big data. *Cluster Computing*, 18(2):549–562, 2015.

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

[5] Q. Fan, D. Zhang, H. Wu, and K. Tan. A general and parallel platform for mining co-movement patterns over large-scale trajectories. *PVLDB*, 10(4):313–324, 2016.

[6] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen. Discovery of convoys in trajectory databases. *PVLDB*, 1(1):1068–1080, 2008.

[7] P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatio-temporal data. In *SSTD*, pages 364–381, 2005.

[8] P. Laube, S. Imfeld, and R. Weibel. Discovering relative motion patterns in groups of moving point objects. *IJGIS*, 19(6):639–668, 2005.

[9] J. Lee, J. Han, and K. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD*, pages 593–604, 2007.

[10] Y. Li, J. Bailey, and L. Kulik. Efficient mining of platoon patterns in trajectory databases. *Data Knowl. Eng.*, 100:167–187, 2015.

[11] Z. Li, B. Ding, J. Han, and R. Kays. Swarm: Mining relaxed temporal moving object clusters. *PVLDB*, 3(1):723–734, 2010.

[12] M. Nanni and D. Pedreschi. Time-focused clustering of trajectories of moving objects. *J. Intell. Inf. Syst.*, 27(3):267–289, 2006.

[13] F. Orakzai, T. Calders, and T. B. Pedersen. Distributed convoy pattern mining. In *IEEE MDM*, pages 122–131, 2016.

[14] F. Orakzai, T. Calders, and T. B. Pedersen. k/2-hop: Fast mining of convoy patterns with effective pruning. *PVLDB*, 12(9):948–960, 2019.

[15] C. Panagiotakis, E. Kokinou, and F. Vallianatos. Automatic p-phase picking based on local-maxima distribution. *IEEE Trans. Geoscience and Remote Sensing*, 46(8):2280–2287, 2008.

[16] C. Panagiotakis and G. Tziritas. A speech/music discriminator based on RMS and zero-crossings. *IEEE Trans. Multimedia*, 7(1):155–166, 2005.

[17] N. Pelekis, I. Kopanakis, E. E. Kotsifakos, E. Frentzos, and Y. Theodoridis. Clustering uncertain trajectories. *Knowl. Inf. Syst.*, 28(1):117–147, 2011.

[18] N. Pelekis, P. Tampakis, M. Vodas, C. Doulkeridis, and Y. Theodoridis. On temporal-constrained sub-trajectory cluster analysis. *Data Min. Knowl. Discov.*, 31(5):1294–1330, 2017.

[19] N. Pelekis, P. Tampakis, M. Vodas, C. Panagiotakis, and Y. Theodoridis. In-dbms sampling-based sub-trajectory clustering. In *EDBT*, pages 632–643, 2017.

[20] K. Seki, R. Jinno, and K. Uehara. Parallel distributed trajectory pattern mining using hierarchical grid with mapreduce. *IJGHPC*, 5(4):79–96, 2013.

[21] P. Tampakis, C. Doulkeridis, N. Pelekis, and Y. Theodoridis. Distributed subtrajectory join on massive datasets. *CoRR*, abs/1903.07748, 2019.

[22] P. Tampakis, N. Pelekis, N. V. Andrienko, G. L. Andrienko, G. Fuchs, and Y. Theodoridis. Time-aware sub-trajectory clustering in hermes@postgresql. In *ICDE*, pages 1581–1584, 2018.

[23] L. A. Tang, Y. Zheng, J. Yuan, J. Han, A. Leung, C. Hung, and W. Peng. On discovery of traveling companions from streaming trajectories. In *ICDE*, pages 186–197, 2012.

[24] M. R. Vieira, P. Bakalov, and V. J. Tsotras. On-line discovery of flock patterns in spatio-temporal data. In *ACM SIGSPATIAL*, pages 286–295, 2009.

[25] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684, 2002.

[26] K. Zheng, Y. Zheng, N. J. Yuan, and S. Shang. On discovery of gathering patterns from trajectories. In *ICDE*, pages 242–253, 2013.

[27] N. Zygouras and D. Gunopulos. Corridor learning using individual trajectories. In *IEEE MDM*, pages 155–160, 2018.