Sequential Pattern Mining from Trajectory Data

Elio Masciari[1], Gao Shi[2], and Carlo Zaniolo[2]

[1] ICAR-CNR – Institute of Italian National Research Council
masciari@icar.cnr.it
[2] UCLA – University of California Los Angeles
{zaniolo, gaoshi}@cs.ucla.edu

Abstract. In this paper, we study the problem of mining for frequent trajectories, which is crucial in many application scenarios, such as vehicle traffic management, hand-off in cellular networks, supply chain management. We approach this problem as that of mining for frequent sequential patterns. Our approach consists of a partitioning strategy for incoming streams of trajectories in order to reduce the trajectory size and represent trajectories as strings. We mine frequent trajectories using a sliding windows approach combined with a counting algorithm that allows us to promptly update the frequency of patterns. In order to make counting really efficient, we represent frequent trajectories by prime numbers, whereby the Chinese reminder theorem can then be used to expedite the computation.

1 Introduction

In this paper, we address the problem of extracting frequent patterns from trajectory data streams. Due to its many applications and technical challenges, the problem of extracting frequent patterns has received a great deal of attention since the time it was originally introduced for transactional data [1, 4]. For trajectory data the problem was studied in [3, 12]. The challenge posed by data stream systems and data stream mining is that, in many applications, data must be processed continuously, either because of real time requirements or simply because the stream is too massive for a store-now & process-later approach. However, mining of data streams brings many challenges not encountered in database mining, because of the real-time response requirement and the presence of bursty arrivals and concept shifts (i.e., changes in the statistical properties of data). In order to cope with such challenges, the continuous stream is often divided into windows, thus reducing the size of the data that need to be stored and mined. This allows detecting concept drifts/shifts by monitoring changes between subsequent windows. Even so, frequent pattern mining over such large windows remains a computationally challenging problem requiring algorithms that are faster and lighter than those used on stored data. Thus, algorithms that make multiple scans of the data should be avoided in favor of single-scan, incremental algorithms. In particular, the technique of partitioning large windows into slides (a.k.a. panes) to support incremental computations has proved very valuable in DSMS [11] and will be exploited in our approach. We will also make use of the following key observation: in real world applications there is an obvious difference between the problem of (i) finding new association rules, and (ii) verifying the continuous validity of existing rules. In order to tame the size curse of point-based trajectory representation, we propose to partition trajectories using a suitable regioning strategy. Indeed, since trajectory data carry information with a detail not often necessary in many application scenarios, we can split the search space in regions having the suitable granularity and represent them as simple strings. The sequence of regions (strings) define the trajectory traveled by a given object. Regioning is a common assumption in trajectory data mining [9, 3] and in our case it is even more suitable since our goal is to extract typical routes for moving objects as needed to answer queries such as: *which are the most used routes between Los Angeles and San Diego?* thus extracting a pattern showing every point in a single route is useless.

The partitioning step allow us to represent a trajectory as string where each substring encodes a region, thus, our proposal for incremental mining of frequent trajectories is based on an efficient algorithm for frequent string mining. As a matter of fact, the extracted patterns can be profitably used in systems devoted to traffic management, human mobility analysis and so on. Although a real-time introduction of new association rules is neither sensible nor feasible, the on-line verification of old rules is highly desirable for two reasons. The first is that we need to determine immediately when old rules no longer holds to stop them from pestering users with improper recommendations. The second is that every window can be divided in small panes on which the search for new frequent patters execute fast. Every pattern so discovered can then be verified quickly. Therefore, in this paper we propose a fast algorithm, called *verifier* henceforth, for verifying the frequency of previously frequent trajectories over newly arriving windows. To this end, we use sliding windows, whereby a large window is partitioned into smaller panes[11] and a response is returned promptly at the end of each slide (rather than at the end of each large window). This also leads to a more efficient computation since the frequency of the trajectories in the whole window can be computed incrementally by counting trajectories in the new incoming (and old expiring) panes.

Our approach in a nutshell. As trajectories flow we partition the incoming stream in windows, each window being partitioned in slides. In order to reduce the size of the input trajectories we pre-process each incoming trajectory in order to obtain a smaller representation of it as a sequence of regions. We point out that this operation is well suited in our framework since we are not interested in point-level movements but in trajectories shapes instead. The regioning strategy we exploit uses PCA to better identify directions along which we should perform a more accurate partition disregarding regions not on the principal directions. The rationale for this assumption is that we search for frequent trajectories so it is unlikely that regions far away from principal directions will contribute to frequent patterns (in the following we will use frequent patterns and frequent trajectories as synonym). The sequence of regions so far obtained can be represented as a string for which we can exploit a suitable version of well known frequent string mining algorithms that works efficiently both in terms of space and time consumption. We initially mine the first window and store the frequent trajectories mined using a tree structure. As windows flow (and thus slides for each window) we continuously update frequency of existing patterns while searching for new ones, This step require an efficient method for counting (a.k.a verification). Since trajectories data are ordered we need to take into account this feature. We implement a novel verifier that exploits prime numbers properties in order to encode trajectories as numbers and keeping order information, this will allow a very fast verification since searching for the presence of a trajectory will result in simple (inexpensive) mathematical operations.

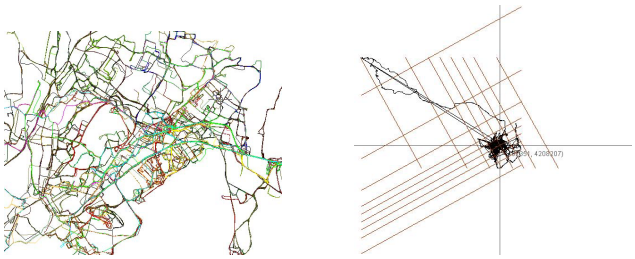
Remark. In this paper we exploit techniques that were initially introduced in some earlier works [14, 13, 15]. We point out that in this work we improved those approaches in order to make them suitable for sequential pattern mining. Moreover, data mining approaches validity relies in their experimental assessment, in this respect the experiments we performed confirmed the validity of the proposed approach.

2 Trajectory size reduction

For transactional data a tuple is a collection of features, instead, a trajectory is an ordered set (i.e., a sequence) of timestamped points. We assume a standard format for input trajectories, as defined next. Let P and T denote the set of all possible (spatial) positions and all timestamps, respectively. A trajectory Tr of length n is defined as a finite sequence $s_1, \cdots, s_n$, where $n \geq 1$ and each s_i is a pair (p_i, t_i) where $p_i \in P$ and $t_i \in T$. We assume that P and T are discrete domains, however this assumption does not affect the validity of our approach. For continuous locations, a viable approach is to

partition the space into regions in order to map the initial locations into discrete regions labeled with a timestamped symbol. The problem of finding a suitable partitioning for both the search space and the actual trajectory is a core problem when dealing with spatial data. Every technique proposed so far, somehow deals with regioning and several approaches have been proposed such as partitioning of the search space in several regions of interest (RoI)[3] and trajectory partitioning (e.g. [10]) by using polylines. In this section, we describe the application of Principal Component Analysis (PCA)[6] in order to obtain a better partitioning. Indeed, PCA finds *preferred* directions for data being analyzed. We refer as *preferred* directions the (possibly imaginary) axes where the majority of the trajectories lie. Once we detect the preferred directions we perform a partition of the search space along these directions. Due to space limitations, instead of giving a detailed description of the mathematical steps implemented in our prototype we will present an illustrating (real life) example, that will show the main features of the approach.

Example 1. Consider the set of trajectories depicted in Fig. 1(a) regarding bus movements in the Athens metropolitan area. There are several trajectories close to the origin of the axes (that is located at city downtown) so it is difficult to identify the most interesting areas for analysis.



(a) Original Data (b) Regions defined by exploiting IPCA

Fig. 1. Trajectory Pre-Elaboration steps

In order to properly assign regions we need to set a suitable level of granularity by defining the initial size s of each region, i.e. their diameter. In order to keep an intuitive semantic for regions of interest we partition the search space into square regions along the directions set by the eigenvalues returned by IPCA. Since the region granularity will affect further analysis being performed the choice of region size s is guided by *DBScan* an unsupervised density based clustering algorithm. The output of the regioning step is depicted in Figure 1(b).

Definition 1 (Dense Regions). *Let T be a set of trajectories, and X_I and Y_I the axes defined by IPCA, $C = \{C_1, C_2, \cdots, C_n\}$ a set of regions obtained with density based algorithm (DBScan) laying on X_I and Y_I, the regions defined by C_i's boundaries are* Dense.

3 Frequent Trajectories Mining

The regioning schema presented in previous section allows a compact representation of trajectories by the sequences of regions crossed by each trajectory, i.e. as a set of strings, where each substring encodes a region. It is straightforward to see that this representation transform the problem of searching frequent information in a (huge) set of multidimensional points into the problem of searching frequent (sub)strings in a set of strings representing trajectories. We point out that our goal is to mine frequent trajectories tackling the "where is" problem, i.e. we are interested in movements made by

objects disregarding time information (such as velocity). Moreover, since the number of trajectories that could be monitored in real-life scenarios is really huge we need to work on successive portion of the incoming stream of data called *windows*. Let $T = \{T_1, \cdots, T_n\}$ be the set of regioned trajectories to be mined belonging to the current window; T contains several trajectories where each trajectory is a sequence of regions. Let $S = \{S_1, \cdots, S_n\}$ denotes the set of all possible (sub)trajectories of T. The *frequency* of a (sub)trajectory S_i is the number of trajectories in T that contain S_i, and is denoted as $Count(S_i, T)$. The *support* of S_i, $sup(S_i, T)$, is defined as its frequency divided by the total number of trajectories in T. Therefore, $0 \leq sup(S_i, T) \leq 1$ for each S_i. The goal of frequent trajectories mining is to find all such S_i, whose support is greater than (or equal to) some given minimum support threshold α. The set of frequent trajectories in T is denoted as $\mathcal{F}_\alpha(T)$. We consider in this paper frequent trajectories mining over a data stream, thus T is defined as a sliding window over the continuous stream. Each window either contains the same number of trajectories (count based or physical window), or contains all trajectories arrived in the same period of time (time-based or logical window). T moves forward by a certain amount by adding the new slide (δ^+) and dropping the expired one (δ^-). Therefore, the successive instances of T are shown as $W_1, W_2, \cdots$. The number of trajectories that are added to (and removed from) each window is called its *slide size*. In this paper, for the purpose of simplicity, we assume that all slides have the same size, and also each window consists of the same number of slides. Thus, $n = |W|/|\mathcal{S}|$ is the number of slides (a.k.a. panes) in each window, where $|W|$ denotes the window size and $|\mathcal{S}|$ denotes the size of the slides.

Mining trajectories in W. As we obtain the string representation of trajectories, we focus on the string mining problem. In particular, given a set of input strings, we want to extract the (unknown) strings that obey certain frequency constraints. The frequent string mining problem can be formalized as follows. *Given a set T of input strings a given frequency threshold α , find the set S_F s.t. $\forall s \in S_F, count(s, T) > \alpha$*

Many proposal have been made to tackle this problem [7, 2]. We exploit in this paper the approach presented in [7]. The algorithm works by searching for frequent strings in different databases of strings, in our paper we do not have different databases, we have different windows instead. We first briefly recall the basic notions needed for the algorithm, more details can be found in [7, 2].

The suffix array SA of a string s is an array of integers in the range [1.. n], which describes the lexicographic order of the n suffixes of s. The suffix array can be computed in linear time [7]. In addition to the suffix array, we define the inverse suffix array SA^{-1}, which is defined by $SA^{-1}[SA[i]] = i \; \forall 1 \leq i \leq n$. The LCP table is an array of integers which is defined relative to the suffix array of a string s. It stores the length of the longest common prefix of two adjacent suffixes in the lexicographically ordered list of suffixes. The LCP table can be calculated in $O(n)$ from the suffix array and the inverse suffix array. The ω-interval is the longest common prefix of the suffixes is s. The algorithm is reported in Figure 2 and its features can be summarized as follows.

Function $extractStrings$ arrange the the input strings in the window W_i in a string S^{aux} consisting of the concatenation of the strings in W_i, using # as a separation symbol and $ as termination symbol. Functions *buildSuffixes* and *buildPrefixes* computes respectively the suffixes and prefixes of S^{aux} and store them using SA and LCP variables. Function *computeRelevantStrings* first compute the number of times that a string s occurs in W_i and then subtract so called correction terms which take care of multiple occurrences within the same string of W_i as defined in [7]. The output frequent strings are arranged in a tree structure that will be exploited for incremental mining purposes as will be explained in next section.

Incremental Mining of Frequent Trajectories. As the trajectories stream flows we need to incremental update the frequent trajectories pattern so far computed (that are inserted in a *Trajectory Tree* (TT)). Our algorithm always maintains a union of the frequent trajectories of all slides in the current window W in TT, which is guaranteed

Method: MineFrequentStrings
Input: A window slide S of the input trajectories;
Output: A set of frequent strings S_F.
Vars:
A string S_{aux};
A suffix array SA;
A prefix array LCP.
1: $S^{aux} = extractStrings(S)$;
2: $SA = buildSuffixes(S^{aux})$;
3: $LCP = buildPrefixes(S^{aux})$;
4: $S_F = computeRelevantStings(W_0, SA, LCP)$
5: **return** S_F;

Fig. 2. The frequent string mining algorithm

to be a superset of the frequent pattern over W. Upon arrival of a new slide and expiration of an old one, we update the true count of each pattern in TT, by considering its frequency in both the expired slide and the new slide. To assure that TT contains all patterns that are frequent in at least one of the slides of the current window, we must also mine the new slide and add its frequent patterns to TT. The difficulty is that when a new pattern is added to TT for the first time, its true frequency in the whole window is not known, since this pattern wasn't frequent in the previous $n-1$ slides. To address this problem, we uses an auxiliary array (aux) for each new pattern in the new slide. The aux array stores the frequency of a pattern in each window starting at a particular slide in the current window. In other words, the auxiliary array stores frequency of a pattern for each window, for which the frequency is not known. The key point is that this counting can either be done eagerly (i.e., immediately) or lazily. Under the laziest approach, we wait until a slide expires and then compute the frequency of such new patterns over this slide and update the aux arrays accordingly. This saves many additional passes through the window. The pseudo code for the the algorithm is given in Figure 3. At the end of each slide, it outputs all patterns in TT whose frequency at that time is $\geq \alpha \dot{n} |S|$. However we may miss a few patterns due to lack of knowledge at the time of output, but we will report them as delayed when other slides expire. The algorithm starts when the first slide has been mined and its frequent trajectories are stored in TT.

Herein, function $updateFrequencies$ updates the frequencies of each pattern in TT if it is present in S. As the new frequent patterns are mined (and stored in TT'), we need to annotate the current slide for each pattern as follows: if a given pattern t already existed in TT we annotate S as the last slide in which t is frequent, otherwise (t is a new pattern) we annotate S as the first slide in which t is frequent and create auxiliary array for t and start monitoring it. When a slide expires (denote it S_{exp}) we need to update the frequencies and the auxiliary arrays of patterns belonging to TT if they were present in S_{exp}. Finally, we delete auxiliary array if pattern t has existed since arrival of S and delete t, if t is no longer frequent in any of the current slides.

Algorithm Properties

Correctness This follows immediately from the fact that a pattern t belongs to $\mathcal{F}_\alpha(W)$ (the frequent patterns is a window), only if it also belongs to $\cap_i \mathcal{F}_\alpha(S_i)$ (the union of frequent patterns for each slide i in the window). Thus, every frequent pattern in W must show up after mining of at least one of the slides and then we add it to TT.

Max Delay The maximum delay allowed by our algorithm is $(n-1)$ slides. Indeed, after expiration of $(n-1)$ slides, we will have a complete history of the frequency of all frequent patterns of $W$and can report them. Moreover, the case in which a pattern is reported after $(n-1)$ slides of time, is rare. For this to happen, patterns support in

Method: IncrementalMaintenance
Input: A trajectory stream T.
Output: A trajectory pattern tree T_T.
Vars:
A window slide S of the input trajectories;
An auxiliary array aux;
A trajectory tree TT'

```
1: For Each New Slide S_new
2:    updateFrequencies(TT, S);
3:    TT' = MineFrequentStrings(S_new);
4:    For Each trajectory t ∈ TT ∩ TT'
5:       annotateLast(S_new, t);
6:    For Each trajectory t ∈ TT' \ TT
7:       update(TT, t);
8:       annotateFirst(S_new, t, t.aux);
9: For Each Expiring Slide S_exp
10:   For Each trajectory t ∈ TT
11:      conditionalUpdateFrequencies(S_exp, t);
12:      conditionalUpdate(t.aux);
13:      if t has existed since arrival of S
14:         delete(t.aux);
15:      if t no longer frequent in any of the current slides
16:         delete(t);
```

Fig. 3. The incremental miner algorithm

all previous $n-1$ slides must be less than α but very close to it, say $\alpha \cdot |S| - 1$, and suddenly its occurrence goes up in the next slide to say β, causing the total frequency over the whole window to be greater than the support threshold. Formally, this requires that $(n-1) \cdot (\alpha \cdot |S| - 1) + \beta \geq \alpha n |S|$, which implies $\beta = n + \alpha \cdot |S| - 1$. This is not impossible, but in real-world such events are very rare, especially when n is a large number (i.e., a large window spanning many slides).

Time Complexity The main steps of the algorithm are the counting phase of all patterns of TT in the new slide and the expired one (i.e. delta maintenance), and the computation (and insertion in TT) of new frequent patterns. Let us denote the average time for the counting step as $f(|S|, |TT|)$. f is a function of the size of TT ($|TT|$) and slide size ($|S|$). The time for computing new patterns (denote it $M(|S|, \alpha)$ is a function of the slide size and the threshold value α. The total running time to process each window will be $2 \cdot f(|S|, |TT|) + M(|S|, \alpha)$ up to some negligible terms (the factor 2 take into account the verification step for new and expiring slides). It is worth noticing that, the only part of this running time that depends on window size ($|W|$) is $|TT|$. Finally, the number of frequent patterns is significantly smaller than the $n \cdot \mathcal{F}_\alpha(S_i)$ since most frequent patterns are common between slides.

Space Consumption The memory required for running our algorithm, consists of an binary tree containing the new slide, and the pattern tree containing the frequent patterns so far computed (which is significantly smaller than the slides size). The only concern which remains is that we need an auxiliary array (of size $n-1$) for each pattern which has been added to TT recently (i.e., within the last n slides). After that period we no longer need an auxiliary array for that pattern and we release its memory. Therefore, the worst case happens when all patterns need such an array resulting in $4 \cdot n \cdot |TT|$ bytes of extra memory (assuming we use 4-byte integers for storing the frequency); this is not prohibitive since the number of patterns is not supposed to be very large.

A very fast verifier for trajectories. In the following, we first define the verifier notion and propose our novel verifier for trajectories data.

Definition 2. *Let T be a trajectories database, P be a given set of arbitrary patterns and min_{freq} a given minimum frequency. A function f is called a verifier if it takes T, P and min_{freq} as input and for each pattern $p \in P$ returns one of the following results: a) p's true frequency in T if it has occurred at least min_{freq} times or otherwise; b) reports that it has occurred less than min_{freq} times (frequency not required in this case).*

It is important to notice the subtle difference between verification and simple counting. In the special case of $min_{freq} = 0$ a verifier simply counts the frequency of all $p \in P$, but in general if $min_{freq} > 0$, the verifier can skip any pattern whose frequency will be less than min freq. This early pruning can be done by the Apriori property or by visiting more than $|T| - min_{freq}$ trajectories. Also, note that verification is different (and weaker) from mining. In mining the goal is to find all those patterns whose frequency is at least min_{freq}, but verification simply verifies counts for a given set of patterns, i.e. verification does not discover additional patterns. Therefore, we can consider verification as a concept more general than counting, and different from (weaker than) mining. The challenge is to find a verification algorithm, which is faster than both mining and counting algorithms, since the algorithm for extracting frequent trajectories will benefit from this efficiency. In our case the verifier needs to take into account the sequential nature of trajectories so we need to count really fast while keeping the right order for the regions being verified. To this end we exploit an encoding scheme for regioned trajectories based on some peculiar features of prime numbers.

4 Encoding Paths for Efficient Counting and Querying

A great problem with trajectory sequential pattern mining is to control the exponential explosion of candidate trajectory paths to be modeled because keeping information about ordering is crucial. Indeed, our regioning step heavily reduce the dataset size so the number of regions we have to deal with is of hundreds of regions instead of thousands of points. Since our approach is stream oriented we also need to be fast while counting trajectories and (sub)paths. To this end, prime numbers exhibit really nice features that for our goal can be summarized in the following two theorems. They have also been exploited for similar purposes for RFID tag encodings [8], but in that work the authors did not provide a solution for paths containing cycles as we do in our framework.

Theorem 1 (The Unique Factorization Theorem). *Any natural number greater than 1 is uniquely expressed by the product of prime numbers.*

As an example consider the trajectory $T_1 = ABC$ crossing three regions A,B,C. We can assign to regions A, B and C respectively the prime numbers 3,5,7 and the position of A will be the first ($pos(A) = 1$), the position of B will be the second ($pos(B) = 2$), and the position of C will be the third ($pos(C) = 3$). Thus the resulting value for T_1 (in the following we refer to it as P_1) is the product of the three prime numbers, $P_1 = 3 * 5 * 7 = 105$ that has the property that does not exist the product of any other three prime numbers that gives as results 105.

As it is easy to see this solution allows to easily manage trajectories since containment and frequency count can be done efficiently by simple mathematical operations. Anyway, this solution does not allow to distinguish among ABC, ACB, BAC, BCA, CAB, CBA, since the *trajectory number* (i.e. the product result) for these trajectories is always 105. To this end we can exploit another fundamental theorem of arithmetics.

Theorem 2 (Chinese Remainder Theorem). *Suppose that $n_1, n_2, \cdots, n_k$ are pairwise relatively prime numbers. Then, there exists W (we refer to it as* witness*) between 0 and $N = n_1 \cdot n_2 \cdots n_k$ solving the system of simultaneous congruences: $W\%n_1 = a_1$, $W\%n_2 = a_2$, ..., $W\%n_k = a_k$.*

Then, by Theorem 2, there exists W_1 between 0 and $P_1 = 3 * 5 * 7 = 105$. In our example, the witness W_1 is 52 since $52\%3 = 1 = pos(A)$ and $52\%5 = 2 = pos(B)$ and $52\%7 = 3 = pos(C)$. We can compute W_1 efficiently using the extended Euclidean algorithm. From the above properties it follows that in order to fully encode a trajectory (i.e. keeping the region sequence) it suffices to store two numbers its prime number product (we refer to it as trajectory number) and its witness. As a nice side-effect in order to obtain any information about region positions in the trajectory we can test with the maximum efficiency containment relationships (a simple division) and order checking (a sequence of divisions). In order to assure that no problem will arise in the encoding phase and witness computation we assume that the first prime number we choose for encoding is greater than the trajectory size. So for example if the trajectory length is 3 we encode it using prime numbers 5,7,11. A devil's advocate may argue that multiple occurrences of the same region leading to cycles violate the injectivity of the encoding function. To this end the following example will clarify our strategy.

Dealing with cycles. Consider the following trajectory $T_2 = ABCAD$, we have a problem while encoding region A since it appears twice in the first and fourth position. We need to assure that the encoding value of A is such that we can say that both $pos(A) = 1$ and $pos(A) = 4$ hold(we do not want two separate encoding value since the region is the same and we are interested in the order difference). Assume that A is encoded as $(41)_5$ (i.e. 41 on base 5, we use 5 base since the trajectory length is 5)) this means that A occurs in positions 1 and 4. The decimal number associated to it is $A = 21$, and we chose as the encoding for $A = 23$ that is the first prime number greater than 21. Now we encode the trajectory using $A = 23$, $B = 7$, $C = 11$, $D = 13$ thus obtaining $P_2 = 23023$ and $W_2 = 2137$ (since the remainder we need for A is 21). As it easy to see we are still able to properly encode even trajectories containing cycles. As a final notice we point out that the above calculation is made really fast by exploiting a parallel algorithm for multiplication operation. We do not report here the pseudo code for the encoding step explained above due to space limitations. Finally, one may argue that the size of prime numbers could be large, however in our case it is bounded since the number of regions is small as confirmed by several empirical studies [5](always less than a hundred of regions for real life applications we investigated).

Definition 3 (Region Encoding). *Given a set $R = \{R_1, R_2, \cdots, R_n\}$ of regions, a function enc from R to $\mathcal{P}$ (the positive prime numbers domain) is a* region encoding function *for R.*

Definition 4 (Trajectory Encoding). *Let $T_i = R_1, R_2 \cdots R_n$ be a regioned trajectory. A trajectory encoding ($E(T_i)$) is a function that associates T_i with a pair of integer numbers $\langle P_i, W_i \rangle$ where $P_i = \prod_{1..n} enc(R_i)$ is the trajectory number and W_i is the witness for P_i.*

Once we encode each trajectory as a pair $E(T)$ we can store trajectories in a binary search tree making the search, update and verification operations quite efficient since at each node we store the $E(T)$ pair. It could happen that there exists more than one trajectory encoded with the same value P but different witnesses. In this case, we store once the P value and the list of witnesses saving space for pointers and for the duplicate P's value. Consider the following set of trajectories along with their encoding values (we used region encoding values: $A = 5$, $B = 7$, $C = 11$, $D = 13$, $E = 15$): (ABC, $\langle 385, 366 \rangle$), (ACB, $\langle 385, 101 \rangle$), (BCDE, $\langle 15015, 3214 \rangle$), (DEC, $\langle 2145, 872 \rangle$). ABC and ACB will have the same P value (385) but their witnesses are $W_1 = 366$ and $W_2 = 101$, so we are still able to distinguish them.

Claim. The proposed trajectory encoding scheme performs a one-to-one encoding for any trajectory.

The proof easily follows by Theorem 2 by injectivity of prime numbers conversion properties.

Method: $checkContainment$
Input:
Two trajectories encodings $E(T_1)$ and $E(T_2)$;
Output:
Yes if $T_1 \in T_2$, no otherwise;
Method:
1: **if** $P_1 > P_2$ **return NO**
2: **if** $P_2 \% P_1 = 0$ then
3: **for each** R_i^1
4: **if** $pos((next(R_i^1))^2) < pos(R_i^2)$ **return NO**
5: **return YES**

Method: $updateTree$
Input:
a trajectory tree B_T and a new window NEW;
Output:
the updated version of B_T;
Vars:
a tree node N;
Method:
1: **for each** $T_i \in NEW$
2: $N = depthSearch(E(T_i))$
3: **if** $N \neq null$
4: $B_T = updateFrequency(N, E(T_i))$
5: **else** $insertNode(E(T_i))$
6: **if** $memoryneeded\, deleteOlderNode(B_T)$
7: **return** B_T

Fig. 4. Algorithms for checking containment and update tree

Once computed the encoded trajectories we build our synopsis by storing them in a search binary tree associated to the input trajectories. At each node of the binary tree we store the trajectory number, the witness and the frequency count for each trajectory. The insertion, remove and update operations have the usual meaning.

A simple algorithm for checking ordered containment. In order to perform efficient trajectory verification we should be able to check containment relationships between trajectories. More in detail, given two trajectories, we call them T_1 and T_2, and their encodings $E(T_1)$ and $E(T_2)$ we need to answer this question 'Is T_1 a sub-sequence of T_2?'. To this end we can exploit the mathematical features of encoded trajectories to design a simple and effective algorithm. In the following we denote the index of a region $R_i \in T_i$ as $pos(R_i)$ and the region following R_i in T_i as $next(R_i)$, we recall that we do not need to store the region positions since they can be obtained by simple math calculation on witnesses. If a region R_i appears into two different trajectories T_l and T_m we denote it as R_i^l and R_i^m.

The algorithm first checks if P value of T_1 is a divider of P value of T_2, if the latter holds this means that all the regions belonging to T_1 also belongs to T_2. To test if containment holds we need to check that every pair of consecutive regions in T_1 are also consecutive in T_2. Consider again our toy example and suppose to check wether $T_1 = BC$ is contained in $T_2 = ABC$. we have that $P_2 = 385$ and $P_1 = 77$. We test that $P_2 \% P_1 = 0$, now we have that $pos(B^2) = 2 < 3 = pos(C^2)$ thus the answer to the containment test is YES. In order to build a fast verifier for trajectory frequencies, we store an additional information at each node, i.e. the frequency of the trajectories stored at that node. Obviously if we have more than one witness for a given P value we will store the frequencies for each witness. Again, doing this we prevent new nodes insertion thus saving memory space. The initial tree is built for the first window W_1, we call it B_T. As new trajectories arise in the stream we continuously update it. In particular, for every incoming trajectory $T_i \in NEW$ we search the node N it belongs to by performing a $depthSearch$ on B_T of the encoding values for T ($E(T_i)$). If such a node N exists (this means that T_i is frequent) we update its frequency. In particular we update the frequency of the witness W_i for T_i (we recall that there may exists more than one witness for the same P value). If the trajectory does not exist in the tree we insert the corresponding node (annotating its timestamp). Finally, as the stream flows if we need to release memory we delete the older nodes (i.e. the ones with older timestamps).

# input sequences	Our Algorithm			T-Patterns		
	times	# regions	# patterns	times	# regions	# patterns
10,000	1.412	94	62	4.175	102	54
20,000	2.115	98	71	6.778	107	61
50,000	3.876	96	77	14.206	108	67
100,000	7.221	104	82	30.004	111	73

(a)

# input sequences	Our Algorithm			T-Patterns		
	times	# regions	# patterns	times	# regions	# patterns
10,000	1.205	94	41	4.175	102	37
20,000	2.003	98	50	6.778	107	43
50,000	3.442	96	59	14.206	108	49
100,000	6.159	104	65	30.004	111	58

(b)

Table 1. Performances comparison against static RoI

The proposed verifier is faster and require less memory than a traditional FP-tree that keeps a separate path for each trajectory. Moreover it improves the conditionalization step since it is implicitly performed when encoding trajectories values.

5 Experimental Results

In this section we will show the experimental results for our algorithms. We used the GPS dataset[16](this dataset being part of *GeoLife*project). It records a broad range of users outdoor movements, thus, the dataset allows a severe test for our frequent sequential pattern mining. In order to compare the effectiveness of our approach we compared it with *T-Patterns* system described in [3]. In particular since *T-Patterns* does not offer streaming functionalities we compare our system using a single large window and compare the extracted patterns w.r.t. the regioning performed. More in detail we compare our results w.r.t. the Static and Dynamic regioning offered by *T-Patterns*.

Comparison against Static RoI. In the following, we compare our algorithm against *T-Patterns* with static RoI by measuring the execution times, the number of extracted regions and the number of extracted patterns for a given support value. Table 1(a) and (b) summarize respectively the results obtained on sets of 10,000 up to 100,000 trajectories extracted for the GPS dataset with 0.5% and 1% as min support value. Table 1(a) shows that when the number of input trajectories increases the execution times linearly increases and our execution time is lower than *T-Patterns*. This can be easily understood since we exploit a really fast frequent miner. A more interesting result is the one on number of extracted regions. As explained in previous sections, we exploit PCA and we focus on regions along principal directions, this allow us to obtain less region but more informative since many trajectories will cross them. As a consequence having a smaller number of accurate regions allows more patterns to be extracted as confirmed in Table 1(a). The intuition behind this result is that when considering a smaller number of regions this imply a greater number of trajectories crossing those regions. The above features are confirmed by the results reported in Table 1(b) for 1% minimum support value (obviously it will change the execution times and number of patterns while the number of extracted regions is equal to previous table). Interestingly enough, the execution times for our algorithm slightly decrease as the min support value increases and this is due to the advantage we get by the verification strategy.

Comparison against Dynamic RoI. In the following, we compare our algorithm against *T-Patterns* with dynamic RoI by measuring the execution times, the number of extracted regions and the number of extracted patterns for a given support value. Tables 2(a) and (b) summarize respectively the results obtained on sets of of 10,000 up to 100,000 trajectories extracted for the GPS dataset with 0.5% and 1% as min support value. Also for this comparison, Table 2(a) shows, for 0.5% minimum support value, that when the number of input trajectories increases the execution times linearly increases and our execution time is better than *T-Patterns*. The other improvements obtained with our algorithm have the same rationale explained above. These features are confirmed by the results reported in 2(b) for 1% minimum support value.

Verifier Effectiveness. As mentioned above *T-Pattern* does not support streaming functionalities so we will report here only our performances on trajectory streams. We first report the results for our verifier using different number of input patterns. Table 3 summarizes the results we obtained on GPS dataset. For this experiment, we provided

# input sequences	Our Algorithm			T-Patterns		
	times	# regions	# patterns	times	# regions	# patterns
10,000	1.412	94	62	4.881	106	56
20,000	2.115	98	71	7.104	111	66
50,000	3.876	96	77	15.306	112	69
100,000	7.221	104	82	302.441	115	75

(a)

# input sequences	Our Algorithm			T-Patterns		
	times	# regions	# patterns	times	# regions	# patterns
1,000	1.205	94	41	5.002	105	40
2,000	2.003	98	50	7.423	108	46
5,000	3.442	96	59	15.974	113	53
10,000	6.159	104	65	32.558	116	60

(b)

Table 2. Performances comparison against dynamic RoI

Running Times	# Patterns
910	10,000
1,172	15,000
1,654	20,000
2,883	50,000
3,756	100,000

(a)

Running Times	# Patterns
634	10,000
972	15,000
1,231	20,000
2,033	50,000
2,977	100,000

(b)

Table 3. Verifier Effectiveness Results

a predefined set of patterns to verify, we first set minimum support to 0 thus verification coincides with counting. We vary the number of patterns given to the algorithms as shown in Table 3(a). In addition we run the same experiment while setting various minimum support value, the results are reported in Table 3(b) only for 0.5% value due to space limitations.

It is worth noticing that the proposed verifier is extremely fast and that the execution time is less than linear w.r.t. the number of input patterns. Such a result is relevant since in many practical situations, including those where data arrival rate is very high, continuously mining the data set is either impractical or unfeasible. For such cases, using the fast verifier allows to monitor the data stream in order to (i) confirm the validity of existing patterns, and (ii) detect any occurrence of concept-shift. In fact, our experiments suggest that concept-shift is always associated with a significant number ($> 5 - 10\%$) of frequent patterns becoming infrequent. Therefore, only when such changes are observed we need to call on a mining algorithm, which will discover the new patterns.

Mining Algorithm Performances. In this section we report the results we ran to test the performances of the proposed incremental mining algorithm for large sliding windows. At the best of our knowledge our algorithm is the first proposal for dealing with frequent pattern mining on trajectory streams so we do not have a "gold" standard to compare with, however the results obtained are really satisfactory since the running times are almost insensitive to the window size. We recall that the algorithm goal is maintaining frequent trajectories over large sliding windows. Indeed, the results shown in Table 4(a) show that the delta-maintenance based approach presented here is scalable with respect to the window size. Finally, we report the total number of frequent pattern as windows flow (the results shown in Table 4(b)). They are computed using a window size of 10,000 trajectories) for a minimum support value of 0,1 %. Indeed we report the total number of patterns that have been frequent wether they are still frequent or not, this information is provided to take into account the concept shift for data being monitored. The results in Table 4(b) shows that after 200 windows being monitored the number of patterns that resulted frequent in some window is more than doubled this means that the users habits heavily changed during the two years period.

6 Conclusion

In this paper we tackled this problem by introducing a very fast algorithm to verify the frequency of a given set of sequential patterns. The fast verifier has been exploited in order to solve the sequential pattern mining problem under the realistic assumption that we are mostly interested in the new/expiring patterns. This delta-maintenance approach effectively mines very large windows with slides, which was not possible before. In summary we have proposed an approach highly efficient, flexible, and scalable to solve the frequent pattern mining problem on data streams with very large windows. Our

Running Times	*Windows size*
773	10,000
891	25,000
1,032	50,000
1,211	100,000
1,304	500,000
2,165	1,000,000

(a)

# Window	*# Patterns*
1	85
10	106
20	125
50	156
100	189
200	204

(b)

Table 4. Mining Algorithm Results

work is subject to further improvements in particular we will investigate: 1) further improvements to the regioning strategy; 2) refining the incremental maintenance to deal with maximum tolerance for delays between slides.

7 Acknowledgments

The authors would like to thank Barzan Mozafari for the invaluable discussions that originated this work.

References

1. R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, 1994.
2. Johannes Fischer, Volker Heun, and Stefan Kramer. Optimal string mining under frequency constraints. In *PKDD*, pages 139–150, 2006.
3. Fosca Giannotti, Mirco Nanni, Fabio Pinelli, and Dino Pedreschi. Trajectory pattern mining. In *KDD*, pages 330–339, 2007.
4. J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.
5. H. Jeung, M. Lung Yiu, X. Zhou, C. S. Jensen, and H. T. Shen. Discovery of convoys in trajectory databases. *PVLDB*, 1(1):1068–1080, 2008.
6. I.T. Jolliffe. *Principal Component Analysis*. Springer Series in Statistics, 2002.
7. A. Kügel and E. Ohlebusch. A space efficient solution to the frequent string mining problem for many databases. *Data Min. Knowl. Discov.*, 17(1):24–38, 2008.
8. C-H Lee and C-W Chung. Efficient storage scheme and query processing for supply chain management using rfid. In *SIGMOD08*, pages 291–302, 2008.
9. J-G Lee, J. Han, X. Li, and H. Gonzalez. *TraClass*: trajectory classification using hierarchical region-based and trajectory-based clustering. *PVLDB*, 1(1):1081–1094, 2008.
10. J-G Lee, J. Han, and K-Y Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD07*, pages 593–604, 2007.
11. J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, 2005.
12. Y. Liu, L. Chen ., J. Pei, Q. Chen, and Y. Zhao. Mining frequent trajectory patterns for activity monitoring using radio frequency tag arrays. In *PerCom*, pages 37–46, 2007.
13. E. Masciari. Trajectory clustering via effective partitioning. In *FQAS*, pages 358–370, 2009.
14. Elio Masciari. Warehousing and querying trajectory data streams with error estimation. In *DOLAP*, pages 113–120, 2012.
15. B. Mozafari, H. Thakkar, and C. Zaniolo. Verifying and mining frequent patterns from large windows over data streams. In *ICDE*, pages 179–188, 2008.
16. Y. Zheng, Q. Li, Y. Chen, and X. Xie. Understanding mobility based on gps data. In *UbiComp 2008*, pages 312–321, 2008.