Discovery of Collocation Episodes in Spatiotemporal Data*

Huiping Cao, Nikos Mamoulis, and David W. Cheung
Department of Computer Science
The University of Hong Kong
Pokfulam Road, Hong Kong
{hpcao,nikos,dcheung}@cs.hku.hk

Abstract

Given a collection of trajectories of moving objects with different types (e.g., pumas, deers, vultures, etc.), we introduce the problem of discovering collocation episodes in them (e.g., if a puma is moving near a deer, then a vulture is also going to move close to the same deer with high probability within the next 3 minutes). Collocation episodes catch the inter-movement regularities among different types of objects. We formally define the problem of mining collocation episodes and propose two scaleable algorithms for its efficient solution. We empirically evaluate the performance of the proposed methods using synthetically generated data that emulate real-world object movements.

1 Introduction

The large volume of the spatiotemporal data, (i.e., moving objects trajectories) renders their manual analysis tedious and impossible. For their efficient analysis, spatiotemporal data mining [1] is proposed for the development and application of novel computational techniques. Given a trajectory database, our goal is to unveil inter-movement regularities among objects of different types, modeled as a sequence of collocation events. Consider an application that monitors the activities of animals (e.g., via sensors attached to them). An exemplary *collocation episode* for this application could be "Once we detect that a puma is moving close to a deer for 1 minute, we expect that a vulture will also move near to this deer in 3 minutes with high probability."

A collocation episode is in fact a sequence of spatiotemporal collocation events. Such events are sets of objects moving close to each other for some period. In addition, there is a particular object type (e.g., deer), called *centric feature*, which participates in a sequence of collocations (e.g., deer-puma, deer-vulture). Finally, in a valid instance of the episode the object that instantiates the common feature should be the same in all collocation instances (e.g., the same deer appears in both deer-puma and deer-vulture collocations). Our definition of a collocation event is a temporal extension of the spatial collocation defined in [6], which models the co-existence of a set of (non-spatial) features, such as environmental observations (humidity and pollution values), plant and animal types, etc., in a spatial neighborhood. E.g., pattern (wet, bamboo) means that, with high probability, we can find bamboo plants near places with high humidity values. Existing methods that consider only spatial relationships between static features are not directly applicable for our problem, since (i) we require a temporal duration for a valid collocation event and (ii) we search for temporal episodes of such events. Our problem also has some similarity with episodes mining in sequence data [5], where frequent episodes are (partially or totally) ordered list of events. A sliding window w is used to extract subsequences in the event series, and the contribution of every subsequence to each candidate episode's frequency is counted. However, the events in our episodes are complex collocations as opposed to simple categorical values. In addition, for a valid episode instance there must be a common feature instantiation[1] (centric feature requirement), as opposed to an appearance of any event of the same type in [5].

In view of the challenges in this problem, we propose a two-step framework for mining collocation episodes. First, we apply a hash-based technique to efficiently retrieve the object pairs, whose trajectories are close during some periods and identify these intervals. Then, we provide two collocation episode mining algorithms (one Apriori-based approach and one that is based on the vertical mining paradigm) and some pruning techniques to improve the mining efficiency. Finally, we empirically evaluate the performance of the proposed methods using synthetically generated data. In the remainder of the paper, we introduce some related work, formally define the problem, outline our algorithms, and present an experimental evaluation for them.

2 Related Work

Besides the work reviewed in Section 1, our work is also related to pattern discovery in one-dimensional time series, e.g., [4] etc. Nevertheless, these problems differ in three main aspects from our work; (i) the pattern/rule element is just a symbol or an event, while our pattern unit is a topological structure with a temporal duration; (ii) patterns are defined based on a single time series, but our patterns are based on relationships among different sequences; and (iii) temporal and time-series data mining is usually based on predefined categorization of 1D values, whereas we work on a continuous (spatiotemporal) data space. In addition, several efforts have been made to extend spatial collocation patterns [6] to contain temporal aspects towards different directions. E.g., from spatiotemporal data, [9] searches for evolving collocations, which in nature are pattern components in our context, while [8] discovers topological patterns (without tempo-

*Work supported by grant HKU 7142/04E from Hong Kong RGC.

[1] Note that two trajectories of the same type (e.g., deer) may correspond to different objects (e.g., two different deers).

ral order). Finally, a related piece of work to our problem is [3], where spatiotemporal pattern queries are proposed and studied. An intuitive example of such a query is "find the moving object that is close to location A at time t_1, and then moves to region C during time interval $[t_3, t_4]$". The main differences of our work are (i) we automatically identify frequent patterns that relate the movement of objects, instead of posing explicit queries and (ii) our patterns relate two or more trajectories (that are feature instances), instead of searching for trajectories that follow a specific "route" specified by a temporal sequence of *static* regions.

3 Problem Definition

This section formally defines the spatiotemporal collocation episodes by gracefully combining the concept of *episode* in event sequences and *collocation* in spatial databases.

3.1 Spatiotemporal sequences and close subsequences

A **spatiotemporal sequence** $\mathcal{S}$ is the trajectory of a moving object. Formally, it is an ordered list of location-time pairs $(l_0, t_0), (l_1, t_1), \cdots, (l_{m-1}, t_{m-1})$ where $t_i < t_j$ if $i < j$ $(i, j \in [0, m))$. The pair (l_i, t_i) denotes that the object was at location l_i at time t_i. In practice, l_i is a 2D position (x_i, y_i), and t_i records the time represented in time units (e.g., one minute is a time unit). In the following discussion, the subscript of a location implicitly refers to its timestamp (i.e., l_i implies the location at time t_i).

Given n $(n \geq 1)$ objects $o_1, o_2, \cdots, o_n$, the trajectory of object o_i is denoted by $\mathcal{S}_i$. Figure 1 plots three exemplary sequences $\mathcal{S}_P$, $\mathcal{S}_D$ and $\mathcal{S}_V$ (abbreviating $\mathcal{S}_{Puma}$, $\mathcal{S}_{Deer}$, and $\mathcal{S}_{Vulture}$, respectively). For illustration purposes, we use 1D values to represent spatial locations, however, our discussion extends naturally to the multidimensional space.

A **subsequence** s **of** $\mathcal{S}$ is a list of continuous location-time pairs of $\mathcal{S}$: $(l_{i_1}, t_{i_1}), (l_{i_2}, t_{i_2}), \cdots, (l_{i_q}, t_{i_q})$, where for $\forall j \in [1, q]$, l_{i_j} is in $\mathcal{S}$ and $t_{i_j} + 1 = t_{i_{j+1}}$. The starting (ending) time of s is denoted by $s.t_s$ $(s.t_e)$. For a complete sequence $\mathcal{S}$ with m positions, $\mathcal{S}.t_s = t_0$ and $\mathcal{S}.t_e = t_{m-1} + 1$.

Definition 1 *A* **window** *is a time interval $[t_s, t_e)$. The time span (or length) of a window $[t_s, t_e)$ is $t_e - t_s$. A window with time span w is called a w-window.*

Definition 2 *A subsequence s is* **on window** *$[t_s, t_e)$ if $s.t_s = t_s$ and $s.t_e = t_e$. A subsequence is called a w-subsequence if it is on a w-window. Two subsequences $s_i \in \mathcal{S}_i$ and $s_j \in \mathcal{S}_j$ are* **concurrent subsequences** *if they are on the same window.*

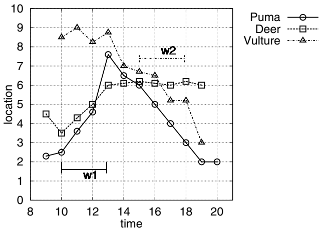

Figure 1: Example of trajectories and windows

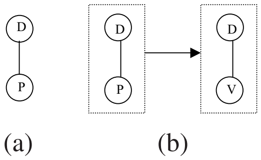

Figure 2: Collocation unit and episode

Example: Figure 1 shows two windows with time span 3: **w1**=$[10, 13)$ and **w2**=$[15, 18)$. For $\mathcal{S}_D$, the two subsequences on **w1** and **w2** are $s_{D_1} = (3.5, 10), (4.3, 11), (5.0, 12)$ and $s_{D_2} = (6.2, 15), (6.1, 16), (6.0, 17)$. For $\mathcal{S}_P$, the subsequence on **w1** is $s_{P_1} = (2.5, 10), (3.6, 11), (4.6, 12)$, while on **w2** we have $s_{V_2} = (6.7, 15), (6.5, 16), (5.2, 17)$ from $\mathcal{S}_V$. s_{D_1} and s_{P_1} (also: s_{D_2} and s_{V_2}) are concurrent subsequences. $\mathcal{S}_D$ is on window $[9, 20)$ and has eight $(t_e - t_s - w = 20 - 9 - 3)$ 3-subsequences, the second of which is on **w1**.

We define the closeness of two concurrent subsequences, using an *aggregate* function $aggDist$ for their element-to-element distances. Typically, $aggDist$ can be the maximum (max) or average (avg) of the component distances. Assuming that s_i and s_j are both on window $[t_s, t_e)$, and $dist$ is some *atomic* distance function (e.g., Euclidean distance), $maxDist(s_i, s_j) = max_{t_s \leq i_t = j_t < t_e}\{dist(l_{i_t}, l_{j_t})\}$ and $avgDist(s_i, s_j) = \frac{\sum_{t_s \leq i_t = j_t < t_e} dist(l_{i_t}, l_{j_t})}{t_e - t_s}$. A *distance threshold* ϵ is used to model closeness:

Definition 3 *Two concurrent subsequences s_i and s_j are* **close**, *denoted by $close(s_i, s_j)$, if $aggDist(s_i, s_j) \leq \epsilon$.*

Example: Assuming $\epsilon = 2.5$ and $aggDist = maxDist$, the two concurrent subsequences s_{D_1} and s_{P_1} on window **w1** of Figure 1 are close to each other since for $\forall i_t = j_t \in [10, 13)$, the location pair $l_{i_t} \in s_i$ and $l_{j_t} \in s_j$ satisfies $dist(l_{i_t}, l_{j_t}) \leq \epsilon$.

3.2 Spatiotemporal collocations and episodes thereof

Let $\mathcal{F}$ be a set of moving object types (e.g., different animals). Given a database of object trajectories, the type (or *feature*) of an object o_i is denoted by $type(o_i)$, such that $type(o_i) \in \mathcal{F}$. In general, the number of objects n in the database can be larger than the number $|\mathcal{F}|$ of types in $\mathcal{F}$; i.e., more than one object may belong to the same object type.

Definition 4 *A* **spatiotemporal collocation unit** *g (simply* unit*) is an undirected graph (V, E) where each vertex in $g.V$ is an object type in $\mathcal{F}$. The* **length** *of the unit g is the number of vertices $|g.V|$ in it. Given a* **unit time span** *w, a* **valid instance** *I_g of unit $g = (V, E)$, where $V = \{f_1, f_2, \ldots, f_{|V|}\}$, is a set of concurrent w-subsequences $\{s_1, s_2, \ldots, s_{|V|}\}$ on a window $[t_s, t_e)$ such that (i) s_i is of type f_i, $(1 \leq i \leq |V|)$ and (ii) if $(f_i, f_j) \in E$ then $close(s_i, s_j)$.*

The starting (ending) time of I_g is denoted by $I_g.t_s$ $(I_g.t_e)$. For example, the two concurrent window trajectories s_{D_1} and s_{P_1} in Figure 1 is an instance of the collocation unit in Figure 2a, and the related window is $[10, 13)$ (i.e., **w1**).

Definition 5 *A* **spatiotemporal collocation pattern** *(or* **episode***) P is an ordered list of spatiotemporal collocation units: $g_1 g_2 \cdots g_\ell$ where $\cap_{i=1}^{\ell}(g_i.V) \neq \varnothing$.*

The object types in $\cap_{i=1}^{\ell}(g_i.V)$, are called the *reference types (features)* of pattern P. The *length* of the pattern P is defined by $\sum_{i=1}^{\ell} |g_i.V|$. A pattern with length k is called a *k-pattern*. In this paper, we only consider the case that $|\cap_{i=1}^{\ell}(g_i.V)| = 1$, and we denote the common (reference) object type as f_r. The reference object type f_r is also called the *centric feature* of the pattern. Thus, we can also represent a pattern in the form $(\underline{f_r}, g_1.V - f_r) \rightarrow \cdots \rightarrow (\underline{f_r}, g_\ell.V - f_r)$, where the underlined feature is the reference feature.

Example: Figure 2b shows a 4-collocation episode, indicating that when a deer and a puma are close during $w = 3$ time units, a vulture will come close to this deer later. This episode's common feature is D and can also be represented by $(\underline{D}, P) \rightarrow (\underline{D}, V)$.

Definition 6 *Given a maximum pattern time span W, a* **valid instance** *I_P for a pattern $P = g_1 g_2 \cdots g_\ell$, is a sequence of valid unit instances $I_{g_1} I_{g_2} \cdots I_{g_\ell}$ such that (i) in all unit* *instances the reference feature f_r is instantiated by a subsequence of the same object sequence, (ii) for every $i < j$, $I_{g_i}.t_e \leq I_{g_j}.t_s$, and (iii) $I_{g_\ell}.t_e - I_{g_1}.t_s \leq W$.*

Example: Let $\epsilon = 2.5$, $w = 3$, and $W = 8$. In Figure 1, we can identify a valid instance of the episode of Figure 2b. In specific, s_{D_1} and s_{P_1} (s_{D_2} and s_{V_2}) instantiate the first (second) unit of the pattern. In addition, s_{D_1} and s_{D_2} instantiate the common feature D in both units and they are parts of the same trajectory. Furthermore, $I_{g_1}.t_e < I_{g_2}.t_s$, since the end point of **w1** is before **w2**. Finally, $I_{g_2}.t_e – I_{g_1}.t_s = 18–10 \leq W$.

Given the maximal episode time span W, we say that a W-window $[t_s, t_e)$ *covers* a pattern instance, if the time span of the instance $[I_{g_1}.t_s, I_{g_\ell}.t_e)$ satisfies $t_s \leq I_{g_1}.t_s$ and $I_{g_\ell}.t_e \leq t_e$. We use $|I_P|$ to denote the number of W-windows, which cover at least one instance of pattern P.

Definition 7 *Pattern $P = g_1 g_2 \cdots g_p$ is a* **superpattern** *of $P' = g'_1 g'_2 \cdots g'_q$ if (i) $P.f_r = P'.f_r$ and (ii) there exists q units $g_{i_1} g_{i_2} \cdots g_{i_q}$ $(1 \leq i_j < i_{j+1} \leq p, 1 \leq j < q)$ of P such that $g'_j.V \subseteq g_{i_j}.V$ and $g'_j.E \subseteq g_{i_j}.E$ for $\forall j \in [1, q]$. P' is a* **subpattern** *of P.*

For Example, $P = (\underline{A}, B, C) \rightarrow (\underline{A}, C, D) \rightarrow (\underline{A}, E)$ is a superpattern of $P' = (\underline{A}, B) \rightarrow (\underline{A}, C, D)$

To measure the interestingness of a collocation episode, we use the reference type as the key factor since it does not make sense to overcount the same instance of the reference feature (e.g., deer) with different instances of the other object types (e.g., puma, vulture) in the pattern. In addition, we consider all possible time windows W, where the pattern may appear.

Definition 8 *The frequency of a pattern P with reference object type f_r is $fr(P, w, W) = \frac{|I_P|}{\sum_{type(o_i)=f_r} |win|}$.*

Here, win is the total number of W-subsequences in all $\mathcal{S}_i$-s where $type(o_i) = f_r$.

Let min_sup be the minimum frequent threshold that the users are interested, one pattern P is frequent if $fr(P, w, W) \geq min\_sup$.

Problem Definition: Given a database of trajectories $\mathcal{S}_1, \cdots, \mathcal{S}_n$ of n moving objects, each with $type(o_i) \in \mathcal{F}$, discover all the frequent spatiotemporal collocations, subject to ϵ, a closeness duration window length w, a maximum pattern window length W, and a frequency threshold $min\_sup \in [0, 1)$.

4 Algorithms

To find the collocation episodes, the main tasks are: (i) identify the types of objects that move closely to each other, and (ii) find on which W-windows this closeness is observed.

4.1 Finding close subsequences

The first mining phase aims at discovering object pairs of different types (f_i, f_j) that have close concurrent subsequences. The ultimate objective is to identify the collocation units that may form longer episodes. For this, we scan each $\mathcal{S}_i$ of type f_i to identify its w-subsequences that are close to object subsequences of different type f_j, $j \neq i$. We store the starting position t_s of each such window $[t_s, t_s + w)$ along with the set of object types close to $\mathcal{S}_i$, during $[t_s, t_s + w)$. Eventually, each trajectory $\mathcal{S}_i$ is converted to a *feature* sequence of the form $\mathcal{S}_i^f = \{\langle F_1, t_1 \rangle, \langle F_2, t_2 \rangle, \ldots, \langle F_m, t_m \rangle\}$, where F_s is the set of object types other than f_i close to the w-subsequence of $\mathcal{S}_i$ that starts at time t_s.

A naive method for the computation of $\mathcal{S}_i^f$ for each $\mathcal{S}_i$, is to scan all the other sequences in order to identify the windows and feature-sets in each $\mathcal{S}_i^f$. We now present a hash-based technique that achieves this goal in two database scans only and is shown in Figure 3. In the first pass, all data are hashed to a 3D grid in the trajectory space (Line 1), where $\mathcal{G}$ and $\mathcal{T}$ are the projected length of each cell on spatial and temporal dimension, and are chosen to be ϵ and w respectively. Then, the algorithm performs a pass over the hashed data by examining only one hyperplane of cells at a time, corresponding to a w-period. For each cell gc, the neighboring cells in the spatial dimensions having the same temporal coordinates as gc are examined. In the 2D example of Figure 4, for cell gc, starting at time t_s, and for the trajectory $\mathcal{S}_i$ (partially) inside gc, the (shaded) cells are checked for possible containment of subsequences which are partially close to $\mathcal{S}_i$. Note that $\mathcal{S}_j$ is close to $\mathcal{S}_i$ at time t_s+2, which means that this closeness relationship can be extended to a subsequence closeness in cells of time span $[t_s + w, t_s + 2w)$. Thus, the algorithm for each $\mathcal{S}_i$ *buffers* such partially close subsequences that can be extended to $\mathcal{S}_i^f$ elements. When the next hyperplane of cells is examined, the partial closeness results are examined for potential extension and inclusion to $\mathcal{S}_i^f$, along with generation of new partial results. The sorting of cell contents by time facilitates the fast identification and extension of partial closeness results, in a merge-join fashion.

```
Algorithm getFS($\mathcal{S}_1, \cdots \mathcal{S}_n, \mathcal{D}, w$)
1.  impose a spatiotemporal $\mathcal{G} \times \mathcal{G} \times \mathcal{T}$ grid;
2.  hash all locations of $\mathcal{S}_1, \cdots, \mathcal{S}_n$ to cells;
3.  for every cell $gc$ in the grid,
4.      sort locations in $gc$ according to their time;
5.  initialize a (partial results) buffer $buf_i$ for $\mathcal{S}_i$;
6.  for each timestamp $t_s$, multiple of $w$
7.      $\mathbb{GC}$ := cells with time interval $[t_s, t_s + w)$;
8.      for each timestamp $t \in [t_s, t_s + w)$
9.          for each grid cell $gc \in \mathbb{GC}$
10.             find location pairs $(l_{i_t}, l_{j_t})$ within $\epsilon$;
11.             extend $buf_i$ and $buf_j$ for each pair;
12.             if $S_j$ in $buf_i$ is close for at least $w$
13.                 add $(f_j, t - w + 1)$ to $\mathcal{S}_i^f$;
14.             if $S_i$ in $buf_j$ is close for at least $w$
15.                 add $(f_i, t - w + 1)$ to $\mathcal{S}_j^f$;
```

Figure 3: Hash-based computation of close feature sets

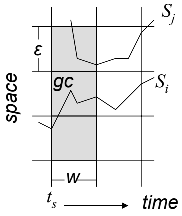

Figure 4: Example of hashing

4.2 Discovery of collocation episodes

In this section, we show two algorithms to discover frequent collocations, based on different usage of the transformed sequence of feature sets.

4.2.1 Pattern extraction from sequences of feature sets

The first algorithm **Apriori** shown in Figure 5 finds the collocations level-by-level. It takes as input the close feature sets $\mathcal{S}_i^f$ found for each $\mathcal{S}_i$, the minimum frequency $mincnt_{f_r}$ $(= min\_sup \times |win|)$ for an episode to be frequent. First, the $\mathcal{S}_i^f$'s are partitioned to $|\mathcal{F}|$ groups, one for each different $f_i \in \mathcal{F}$. Thus, the group $\mathbb{S}_{f_r}$, for feature f_r is used to find the patterns, having f_r as their reference feature. We note that the apriori property holds for frequent episodes, i.e., if P' is a superpattern of P, then $fr(P, w, W) \leq fr(P', w, W)$. In this algorithm, when we measure the length of a pattern, we exclude the reference feature f_r from the units, since it is implicit. For example, a 3-candidate $(f_i) \rightarrow (f_j, f_k)$ represents a real 5-candidate $(\underline{f_r}, f_i) \rightarrow (\underline{f_r}, f_j, f_k)$.

```
Algorithm Apriori($\mathbb{S}_{f_r}, W, mincnt_{f_r}$)
    $L_1$ := frequent 1-patterns; $\ell$ := 2;
    while ($L_{\ell-1} \neq \varnothing$)
        $C_\ell$ := gen_cand($L_{\ell-1}$);
        for each $\mathcal{S}_i^f \in \mathbb{S}_{f_r}$
            slide_window ($C_\ell, \mathcal{S}_i^f, W$);
        $L_\ell := \{P \in C_\ell |$
            $P.count \geq mincnt_{f_r}\}$;
        $\ell := \ell + 1$;
    return $L := \cup_\ell L_\ell$;
```

Figure 5: Apriori-based algorithm

```
Algorithm MJ($\mathbb{S}_{f_r}, W, mincnt_{f_r}$)
1.  generate $ITList_{f_r}(f_j)$ for each $(f_j)$;
2.  use $ITList_{fr}(f_j)$ to generate $L_1$;
3.  $\ell$ := 2;
4.  while ($L_{\ell-1} \neq \varnothing$)
5.      $C_\ell$ := gen_cand($L_{\ell-1}$);
6.      for each $P \in C_\ell$
7.          $ITList_{f_r}(P)$ :=
                MJ_count_cand($P, W$);
8.      $L_\ell := \{P \in C_\ell |$
            $P.count \geq mincnt_{f_r}\}$;
9.      $\ell := \ell + 1$;
10. return $L := \cup_\ell L_\ell$;
```

Figure 6: Merge join algorithm

Function **gen_cand**, used to generate the ℓ-candidates from $(\ell - 1)$-patterns, is exactly as that in sequential pattern mining [7], so we will not discuss it in detail. The patterns excluding the reference features are similar to the sequential patterns in transactional databases [7]. However, counting the support of our patterns is different, since we consider all positions of a sliding window, whereas for sequential patterns each transaction sequence contributes one or none to a sequential pattern (depending on whether the sequence is a superpattern of it or not). Function **slide_window** is used to count $|I_c|$ (the number of windows that contain valid instances of c) for each candidate $c \in C_\ell$ from a transformed sequence $\mathcal{S}_i^f \in \mathbb{S}_{f_r}$. In brief, the idea is to slide a W-window over $\mathcal{S}_i^f$ to get a subsequence of feature sets. For each subsequence s on a W-window, we find the candidates that have a valid instance, which is covered (i.e., supported) by s, and increase their *count*. Sliding window counting for event episodes has also been proposed in [5], however, the valid instances in our case are more difficult to count, because of the constraint that one collocation unit instance should end before the beginning of the next one (see condition (ii) in Definition 6). For example, assuming $w = 3$ and $\mathcal{S}_i^f = \{\langle(f_1, f_2), 10\rangle, \langle(f_1, f_3), 11\rangle, \langle(f_4), 14\rangle\}$, pattern $c_1 = (f_1, f_2) \rightarrow (f_4)$ is supported by $\mathcal{S}_i^f$, but pattern $c_2 = (f_1, f_2) \rightarrow (f_3)$ is not, since f_3 is close to the reference feature at time 11, which is before the end of f_2 (10+w=13). In simple words, in a valid pattern instance, the collocation unit instances should not overlap in time.

Optimizing the support counting While sliding a W-window over the transformed sequence, if the subsequence of $\mathcal{S}_i^f$ covered by the window remains the same compared to the previous window position, the set of candidates supported by the window does not change. As a result, we examine only positions of the W window, where either (i) a feature-set F is included in the window for the first time or (ii) F ceases to be included in the window (compared to the previous position). E.g., let $w = 3$, $W = 8$, and $\mathcal{S}_i^f = \{\langle(f_1, f_2), 10\rangle, \langle(f_1, f_3), 11\rangle, \langle(f_4), 14\rangle\}$. Since only three windows, $[5, 13), [6, 14), [9, 17)$, correspond to the *event* of a feature-set entering the sliding window, and two windows, $[11, 19), [12, 20)$, correspond to the *event* that a feature-set leaves the window, we just need to examine these five windows. Each feature-set $\langle F_i, t_i\rangle \in \mathcal{S}_i^f$ affects two positions of window $[t_s, t_e)$; the one with $t_e = t_i + w$ (where F_i enters the window) and the one with $t_s = t_i + 1$ (where F_i leaves it). As a result, the cost of examining a feature-set sequence $\mathcal{S}_i^f$ becomes proportional to $|\mathcal{S}_i^f|$, instead of the number of window positions (which normally is much larger).

Figure 7 shows in detail this optimized counting method applied for each $\mathcal{S}_i^f$. To avoid overcounting a pattern having more than one instance at a window position, when we detect a valid instance, we add to its support only for the window positions, where previous instances are not valid. For this, we maintain a variable $c.last$ for each candidate (initialized to –1), indicating the last known position of W, having an instance of c. In addition, the algorithm keeps track of the feature-sets fs contained in W. Whenever a feature-set F exits the sliding window, it is removed from fs. If a new F enters fs, we search for candidates for which the last unit is instantiated by some features in F (instances not affected by F are identified at earlier positions of W). I.e., only candidates, for which the features in the last unit are all contained in F, are checked for instantiation. For each candidate, if we detect a valid instance at the current window position, we look for the pattern instance with the latest starting time $I_{g_1}.t_s$. The support of the candidate is then updated with the number of window positions $I_{g_1}.t_s - t_s + 1$, during which the pattern instance remains valid (when $t_s > I_{g_1}.t_s$, the instance becomes outdated). Finally, if some window positions were already counted due to the last detected pattern for c, i.e., if $c.last \geq t_s$, then we add $I_{g_1}.t_s – c.last$ to $c.count$ (instead of $I_{g_1}.t_s – t_s + 1$), in order not to overcount the specific candidate.

```
Function slide_window($C_\ell, \mathcal{S}_i^f, W$)
    for each candidate $c$
        $c.last := -1$; $c.count := 0$; $fs := \varnothing$;
    slide a $[t_s, t_e)$ $W$-window over $\mathcal{S}_i^f$
        if some feature set $F \in fs$ becomes outdated
            $fs := fs - F$;
        if some feature set $F$ enters the window
            $fs := fs + F$;
            for each candidate $c$
                find instance of $c$ with $I_{g_\ell}$ instantiated by $F$
                    and largest possible $I_{g_1}.t_s$;
                if there exists such an instance
                    $h := \min\{I_{g_1}.t_s - t_s + 1, I_{g_1}.t_s - c.last\}$;
                    $c.count := c.count + h$;
                    $c.last := I_{g_1}.t_s$;
```

Figure 7: Optimized support counting

4.2.2 Pattern extraction by joining instances of patterns

Our second algorithm follows the vertical mining paradigm. Instead of scanning the $\mathcal{S}_i^f$ lists multiple times, while generating and counting candidates level-by-level, we keep track of the details about the instances of the patterns and join them to produce the instances of their superpatterns.

Figure 6 shows a pseudocode for this **merge join** (**MJ**) algorithm. First (Line 1), we scan the $\mathcal{S}_i^f$ lists, to produce the *instance lists* (ITLists) of all 1-patterns. For each reference feature f_r, all $\mathcal{S}_i^f \in \mathbb{S}_{f_r}$ produce the instances of 1-patterns having f_r as reference feature. Consider a feature-set $\langle F_i, t_i\rangle \in \mathcal{S}_i^f$. For each feature $f_j \in F_i$ an element (o_i, t_i) is added to list $ITList_{f_r}(f_j)$, indicating that there is an object of feature f_j close to object o_i of feature f_r at time window $[t_i, t_i + w)$. By sliding a window W over $ITList_{f_r}(f_j)$, we can compute the supports of the 1-pattern (f_j) referencing f_r. The ITlists are then used to find the frequent 1-patterns L_1 (Line 2 of the algorithm).

For counting the instances of a longer candidate pattern P (procedure **MJ_count_cand**), we slide a W-window along the two ITLists of the two subpatterns P_1 and P_2 that generate P, and merge-join the lists to create $ITList_{f_r}(P)$. For every position t of W, such that $ITList_{f_r}(P_1)$ and $ITList_{f_r}(P_2)$ contain entries of the same o_i and these entries qualify the pattern constraints, a new instance is generated for $ITList_{f_r}(P)$. Entries in the ITList of a long pattern with k units is a list of $(o_i, I_{g_1}.t_s, \ldots, I_{g_k}.t_s)$. We distinguish three cases for this merge-join process:

- P_1 and P_2 contain collocation units that are exactly the same in P. For example, $P_1 = (f_1) \rightarrow (f_2)$, $P_2 = (f_1) \rightarrow (f_3)$, $P = (f_1) \rightarrow (f_2) \rightarrow (f_3)$. In this case, $ITList_{f_r}(P_1)$, $ITList_{f_r}(P_2)$ are joined according to the t_s time of the common unit, while the rest of the temporal constraints are verified.
- P_1 and P_2 contain collocation units that are joined in P. E.g, $P_1 = (f_1, f_2)$, $P_2 = (f_2) \rightarrow (f_3)$, $P = (f_1, f_2) \rightarrow (f_3)$. In this case, $ITList_{f_r}(P_1)$, $ITList_{f_r}(P_2)$ are joined according to the t_s time of the joined units, while the rest of the temporal constraints are verified.
- P_1 and P_2 do not have common or joined units. For example, $P_1 = (f_1)$, $P_2 = (f_2)$, $P = (f_1) \rightarrow (f_2)$. In this case, we perform a *band-join* [2] between $ITList_{f_r}(P_1)$ and $ITList_{f_r}(P_2)$ to produce $ITList_{f_r}(P)$. The band-join is a straightforward extension of the merge join algorithm that replaces the equality condition by a maximum difference constraint (maximum time difference W in our example).

5 Experimental Evaluation

This section experimentally evaluates the performance of the proposed algorithms based on synthetically generated data due to the lack of real data. All experiments were run on a Pentium III Xeon 700MHz workstation with 4096MB RAM, running Solaris 9x86. The generator takes as input the following parameters: $|\mathcal{F}|$, the number of features; ℓ, the maximal length of the generated episodes; n, the number of sequences (i.e., objects); m, the maximal length of every sequence; ϵ, w, W, and min_sup, which have the same meaning as that in the problem definition. Given these parameters, we generate n trajectories, each of which is assigned to a type in $\mathcal{F}$ while making sure that the generated trajectories instantiate collocation episodes. The default values of the data generation parameters are $n = 500$, $m = 2000$, $w = 2$, $W = 20$, $|\mathcal{F}| = 40$, $\ell = 7$, $\epsilon = 1\%$ and $min\_sup = 0.03$. Unless otherwise stated, we use the same parameter values in data generation and data mining.

Performance evaluation Our methods discover the collocation episodes in two steps; first, close feature sets are found and then longer patterns are extracted from them. For the first step, apart from the proposed hash-based method, we implement a naive one by linearly scanning all other trajectories. For the second step, besides implementing the two algorithms **Apriori** and **MJ**, we also developed a non-optimized version of the **Apriori** algorithm, which does not employ the optimized counting approach shown in Figure 7. We compare the performance of four methods. *Apriori-base* applies linear scan in the first step and non-optimized Apriori for finding the patterns. *Apriori-noprune*, *Apriori*, and *MJ* use the hash-based method in the first step, and non-optimized Apriori, optimized Apriori, and MJ respectively, in the second step. The difference between the linear scan method and the hash-based approach in the first step can be seen by comparing *Apriori-base* and *Apriori-noprune*. The difference between finding collocations using the transformed sequences and the ITLists could be observed from *Apriori* and *MJ*. Finally, by comparing *Apriori-noprune* with *Apriori* we can see the effect of optimized support counting in Apriori.

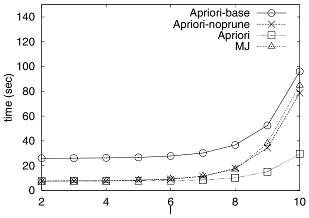

Figure 8: Time vs. ℓ

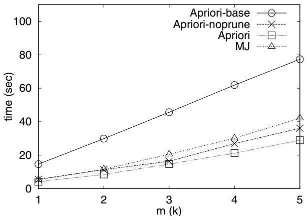

Figure 9: Time vs. m

Figure 8 shows that the mining cost increases with the maximal length ℓ of the generated episodes. In addition, since the number of candidates in each level grows exponentially to ℓ, the cost varies slightly for smaller ℓ, and increases sharply when ℓ becomes large. However, the optimized counting of Apriori slows down this exponential growth. Figure 9 illustrates the scalability of the methods over the maximal length m of the sequences. It shows that the mining cost grows nearly linear to m, exhibiting good scalability over the data volume. For changing n, the linear changing trend could be observed. To summarize, for finding close feature pairs, the hash-based technique is much faster than the linear scan method, whereas for discovering collocation episodes from feature sets, the Apriori method with the counting optimization technique performs best. On the other hand, in most cases, **MJ** is not as efficient as **Apriori**, due to the large volume of generated and joined ITLists.

6 Conclusion

In this paper, we studied the problem of discovering frequent collocation episodes from spatiotemporal data. We provided a novel and carefully designed definition of this new and important mining problem. In addition, we designed an efficient two-phase mining methodology. In the first phase, a hash-based technique is used to convert the original trajectories to sequences of close features to the corresponding object. In the second phase, an Apriori-based technique is devised to discover the frequent episodes. We showed by experimentation that the best combination of techniques in both phases is efficient and scalable.

References

[1] G. Andrienko, D. Malerba, M. May, and M. Teisseire, editors. *ECML/PKDD Workshop on Mining Spatio-Temporal Data*, 2005.

[2] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *VLDB*, 1991.

[3] M. Hadjieleftheriou, G. Kollios, P. Bakalov, and V. J. Tsotras. Complex spatio-temporal pattern queries. In *VLDB*, 2005.

[4] J. Lin, E. J. Keogh, A. W.-C. Fu, and H. V. Herle. Approximations to magic: Finding unusual medical time series. In *18th IEEE Symp. on Computer-Based Medical Systems (CBMS)*, 2005.

[5] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):9, 1997.

[6] S. Shekhar and Y. Huang. Discovering spatial co-location patterns: A summary of results. In *SSTD*, 2001.

[7] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *EDBT*, 1996.

[8] J. Wang, W. Hsu, and M.-L. Lee. A framework for mining topological patterns in spatio-temporal databases. In *CIKM*, 2005.

[9] H. Yang, S. Parthasarathy, and S. Mehta. A generalized framework for mining spatio-temporal patterns in scientific data. In *KDD*, 2005.