Mining Positional Data Streams

Jens Haase and Ulf Brefeld

Knowledge Mining & Assessment Group
Technical University of Darmstadt, Germany

Abstract. We study frequent pattern mining from positional data streams. Existing approaches require discretised data to identify atomic events and are not applicable in our continuous setting. We propose an efficient trajectory-based preprocessing to identify similar movements and a distributed pattern mining algorithm to identify frequent trajectories. We empirically evaluate all parts of the processing pipeline.

1 Introduction

Recent advances in telecommunication, sensing, and recording technologies allow for storing positions from moving objects at large scales in (near) real time. Analysing positional data streams is highly important in many applications; examples range from navigation and routing systems, network traffic, animal migration/tracking to tactics in team sports.

In this paper, we focus on identifying frequent movement patterns in positional data streams that consist of a possible infinite sequence of coordinates. Existing approaches to frequent pattern mining [3, 17] use identities of atomic events to define sequences (episodes) [12]. In positional data, events correspond to sequences of positions (i.e., trajectories) and due to the continuous domain it is very unlikely to observe a trajectory twice. Instead, we observe a multitude of different trajectories that give rise to an exponentially growing set of possibly frequent sequences. Consequentially, mining positional data can only be addressed in the context of big data.

Our contribution is threefold: (i) To remedy the absence of matching atomic events, we propose an efficient preprocessing of the positional data using locality sensitive hashing and approximate dynamic time warping. (ii) To process the resulting near-neighbour trajectories we present a frequent pattern mining algorithm that generalises Achar et al. [1] to positional data. (iii) We present a distributed algorithm for processing positional data at large-scales. Empirically, we evaluate all stages of our approach on positional data of a real soccer game where cameras and sensors realise a bird's eye view of the pitch that allows for locating the players and the ball several times per second.

2 Related Work

Spatio-temporal data mining aims to extract the behaviour and relation of moving objects from (positional) data streams and is frequently used in computational biology for mining animal movements. Trajectory-based patterns are

first introduced by [5]. These patterns represent a set of individual trajectories that share the property of visiting the same sequence of places within similar travel time. Trajectory-based approaches use a discretisation of the movements to identify places that are also known beforehand. Our contribution considers a continuous generalisation: every coordinate on the pitch is a place of interest and trajectories are relations between coordinates and travel time.

Event sequence mining has been introduced by [3] as a problem of mining frequent sequential patterns in a set of sequences. Sequential pattern mining discovers frequent subsequences as patterns in a sequence database. The most common example is the cart analysis proposed by [3]. Frequent episode discovery is a technique to describe and find patterns in a stream of events [12]. Achar et al. [1] propose the first approach to mine unrestricted episodes. Our approach generalises [1] to mining positional data streams.

The behaviour of individual players is analysed by [6] and [7]. [6] analyse groups of players and their behaviour using self organising maps on top of the positional data. Every neuron of the network represents a certain area of the pitch. Thus, whenever a player moves into such an area, the respective neuron is activated. Similarly, [7] uses positional data to assess player positions in particular areas of the pitch, such as catchable, safe or competing zones. Prior work for instance also utilises positional data to identify tactical patterns [13]. However, these approaches usually focus on detecting *a priori* known patterns in the data stream. By contrast, we leverage the findings of *trajectory pattern* and *frequent episode discovery* to devise a purely data-driven approach to find tactical patterns in positional data without making any assumptions on zones, tasks or movements.

3 Efficiently Finding Similar Movements

3.1 Representation

Given a positional data stream $\mathcal{D}$ with ℓ objects $\boldsymbol{o}_1, \ldots, \boldsymbol{o}_\ell$. Every object $\boldsymbol{o}_i$ is represented by a sequence of coordinates $\mathcal{P}_i = \langle \boldsymbol{x}_1^i, \boldsymbol{x}_2^i, \ldots \rangle$ where $\boldsymbol{x}_t = (x_1, x_2, \ldots, x_d)^\top$ denotes the position of the object in d-dimensional space at time t. A trajectory or movement of the i-th object is a subset $\boldsymbol{p}_{[t,t+m]} \subseteq \mathcal{P}_i$ of the stream, e.g., $\boldsymbol{p}_{[t,t+m]} = \langle \boldsymbol{x}_t^i, \boldsymbol{x}_{t+1}^i, \ldots, \boldsymbol{x}_{t+m}^i \rangle$, where m is the length of the trajectory. In the remainder, the time index t is omitted and each element of a trajectory is indexed by offsets $1, \ldots, m$.

For generality, we focus on finding similar trajectories where (i) the exact location of a trajectory does not matter (*translation invariance*), (ii) the range of the trajectory is negligible (*scale invariance*), and where turns such as left or right are considered identical (*rotation invariance*). Note that, depending on the application at hand, one or more of these requirements may be inappropriate and can be dropped by altering the representation accordingly.

Using the requirements (i)-(iii) gives rise to the so-called angle/arc-length representation [16] of trajectories that represents movements as a list of tuples

of angles θ_t and distances $\boldsymbol{v}_t = \boldsymbol{x}_t - \boldsymbol{x}_{t-1}$. The difference $\boldsymbol{v}_t$ is called the *movement vector* at time t and the angles are computed with respect to a (randomly drawn) reference vector $\boldsymbol{v}_{ref} = (1, 0)^\top$. Transformed trajectories are normalised by subtracting the average so that $\theta_i \in [-\pi, +\pi]$ for all i and by normalising the total distance to one. Finally, we discard the difference vectors and represent trajectories solely by their sequences of angles, $\boldsymbol{p} \mapsto \tilde{\boldsymbol{p}} = \langle \theta_1, \ldots, \theta_n \rangle$.

3.2 Approximate Dynamic Time Warping

Recall that pairs of trajectories may contain phase shifts, that is, a movement may begin slowly and then speeds-up while another starts fast and then slows down towards the end. Such phase shifts are well captured by alignment-based similarity measures such as dynamic time warping [14].

Dynamic time warping (DTW) is a non-metric distance function that measures the distance between two sequences and is often used in speech recognition problems. Given two sequences $\boldsymbol{s} = \langle s_1, \ldots, s_n \rangle$ and $\boldsymbol{q} = \langle q_1, \ldots, q_m \rangle$ and a cost function $cost(s_i, q_j)$ detailing the costs of matching s_i with q_j. The goal of dynamic time warping is to find an alignment of sequences $\boldsymbol{s}$ and $\boldsymbol{q}$ that has minimal costs subject to *boundary*, *continuity*, and *monotonicity* constraints [9]. Note that che cost function *cost* can be arbitrarily defined and the complexity of DTW is $\mathcal{O}(|\boldsymbol{s}||\boldsymbol{q}|)$ which is prohibitive for mining positional data streams.

Efficient approximations of dynamic time warping can be obtained by lower bounds. The rationale is that lower bound functions can be computed in less time and are therefore often used as pruning techniques in applications like indexing or information retrieval. The exact DTW computation only needs to be carried out if the lower bound value is above a given threshold. We make use of two lower bound functions, f_{kim} [10] and f_{keogh} [8], that are defined as follows: f_{kim} focuses on the first, last, greatest and smallest values of two sequences [10] and can be computed in $\mathcal{O}(m)$:

$$f_{kim}(\boldsymbol{s}, \boldsymbol{q}) = \max\left\{|s_1 - q_1|, |s_m - q_m|, |\max(\boldsymbol{s}) - \max(\boldsymbol{q})|, |\min(\boldsymbol{s}) - \min(\boldsymbol{q})|\right\}.$$

If the greatest and the smallest entries are normalised to a specific value their computation can be ignored and the time complexity reduces to $\mathcal{O}(1)$. The second lower bound f_{keogh} [8] uses minimum $\ell_i = \min(q_{i-r}, \ldots, q_{i+r})$ and maximum values $u_i = \max(q_{i-r}, \ldots, q_{i+r})$ for sub-sequences of the query $\boldsymbol{q}$ where r is a user defined threshold. Trivially, $u_i \geq q_i \geq \ell_i$ holds for all i and the lower bound f_{keogh} is given by $f_{keogh}(\boldsymbol{q}, \boldsymbol{s}) = \sqrt{\sum_{i=1}^{m} c_i}$ where $c_i = (s_i - u_i)^2$ if $s_i > u_i$, $c_i = (s_i - \ell_i)^2$ if $s_i < \ell_i$, and $c_i = 0$ otherwise. The function f_{keogh} can also be computed in $\mathcal{O}(m)$. The result is a non-metric distance function that only violates the triangle inequality of a metric distance.

3.3 An *N*-Best Algorithm

Given a trajectory $\boldsymbol{q} \in \mathcal{D}$, the goal is to find the most similar trajectories in $\mathcal{D}$. Trivially, a straight forward approach is to compute the DTW values of $\boldsymbol{q}$ for all

trajectories in $\mathcal{D}$ and sort the outcomes accordingly. However, this requires $|\mathcal{D}|$ DTW computations, each of which is quadratic in the length of the trajectories, and renders the approach clearly infeasible.

We now sketch how to compute the N most similar trajectories for a given query $\boldsymbol{q}$ efficiently by making use of the lower bound functions f_{kim} and f_{keogh}. The algorithm begins with computing the DTW distances of the first N entries in the database and stores the entry with the highest distance to $\boldsymbol{q}$. A loop over the remaining trajectories in $\mathcal{D}$ first applying the lower bound functions f_{kim} and f_{keogh} to efficiently filter irrelevant movements before using the exact DTW distance for the remaining candidates. Every trajectory, realising a smaller DTW distance than the current maximum, replaces its peer; auxiliary variables *maxdist* and *maxind* are updated accordingly. Note that the complexity of the algorithm is linear in the number of trajectories in $\mathcal{D}$. In the worst case, the sequences are sorted in descending order by the DTW distance, which requires to compute all DTW distances. In practice much lower run-times are observed.

A crucial factor is the tightness of the lower bound functions. The better the approximation of the DTW, the better the pruning. For $N = 1$, the maximum value drops faster towards the lowest possible value. By contrast, setting $N = |\mathcal{D}|$ requires to compute the exact DTW distances for all entries in the database. Hence, in most cases, $N \ll |\mathcal{D}|$ is required to reduce the overall computation time. The computation can trivially be distributed with Hadoop; computing distances is performed in the mapper and sorting is done in the reducer.

3.4 Distance-based Hashing

An alternative to the introduced N-Best algorithm provides locality sensitive hashing (LSH). A general class of LSH functions are called distance-based hashing (DBH) that can be used together with arbitrary spaces and (possibly non-metric) distances [4]. The hash family is constructed as follows. Let $h : \mathcal{X} \to \mathbb{R}$ be a function that maps elements $x \in \mathcal{X}$ to a real number. Choosing two randomly drawn members $x_1, x_2 \in \mathcal{X}$, the function h is defined as

$$h_{x_1,x_2}(x) = \frac{dist(x, x_1)^2 + dist(x_1, x_2)^2 - dist(x, x_2)^2}{2\, dist(x_1, x_2)}.$$

The binary hash value for x simply verifies whether $h(x)$ lies in an interval $[t_1, t_2]$, that is $h^{[t_1,t_2]}_{x_1,x_2}(x) = 1$ if $h_{x_1,x_2}(x) \in [t_1, t_2]$ and $h^{[t_1,t_2]}_{x_1,x_2}(x) = 0$ otherwise. where the boundaries t_1 and t_2 are chosen so that the probability that a randomly drawn $x \in \mathcal{X}$ lies with 50% chance within and with 50% chance outside of the interval. Given the set $\mathcal{T}$ of admissible intervals and hash function h, the DBH family is defined as the set of all admissible hash functions $h^{[t_1,t_2]}$. Using random draws from $\mathcal{H}_{DBH}$, new hash families can be constructed using *AND*- and *OR*-concatenation.

We use DBH to further improve the efficiency of an N-Best algorithm by removing a great deal of trajectories before processing them. Given a query trajectory $\boldsymbol{q} \in \mathcal{D}$, the retrieval process first identifies candidate objects that

Algorithm 1 FSATransition(α, fsa, t, $events$)

```
1: if fsa.currentState.Open = ∅ then
2:    return fsa {FSA is in final state}
3: end if
4: for n ∈ sourceNodes(fsa.currentState.Open) do
5:    for e ∈ events do
6:       if e ∼ nodeMapping_α(n) then
7:          fsa.currentState.Open = fsa.currentState.Open \ n
8:          fsa.currentState.Done = fsa.currentState.Done ∪ n
9:          fsa.lastTransition = t
10:         if fsa.startTime == undefined then
11:            fsa.startTime = t
12:         end if
13:         break inner loop {Only one possible similarity (injective episode)}
14:      end if
15:   end for
16: end for
17: return fsa
```

are hashed to the same bucket for at least one of the hash functions, and then computes the exact distances of the remaining candidates using the N-Best algorithm. As distance measure of the DBH hash family we use the lower bound f_{kim}. The computation is again easily distributed with Hadoop.

4 Frequent Episode Mining for Positional Data

The main difference between frequent episode mining and mining frequent trajectories from positional data streams is the definition of events. For positional data, every trajectory in the stream is considered an event. Thus, events may overlap and are very unlikely to occur more than just once. We resort to the previously defined approximate distance functions in the mining step.

An event stream is a time-ordered stream of trajectories. Every event is represented by a tuple (A, t) where A is an event and t denotes its timestamp. An *episode* α is a directed acyclic graph, described by a triplet $(\mathcal{V}, \leq, m)$ where $\mathcal{V}$ is a set of nodes, $\leq$ is a partial order on $\mathcal{V}$ (directed edges between the nodes), and $m : \mathcal{V} \to E$ is a bijective function that maps nodes to events in the event stream. We focus on *transitive closed episodes* [15] in the remainder, that is if node A is ordered before B ($A < B$) there must be a direct edge between A and B, that is, $\forall A, B \in \mathcal{V}$ if $A < B \implies edge(A, B)$. The partial ordering of nodes upper bounds the number of possible directed acyclic graphs on the event stream. The ordering makes it impossible to include two identical (or similar) events in the same episode. Episodes that do not allow duplicate events are called *injective episodes* [1].

An episode α is called *frequent*, if it occurs often enough in the event stream. The process of counting the episode α consists of finding all episodes that are

Algorithm 2 Map(id, α)

```
1: eventStream = loadEventStreamFormFile()
2: frequency = 0; fsas = {new FSA}
3: for all (t, events) ∈ eventsStream do
4:    for all fsa ∈ fsas do
5:       inStartState = inStartState(fsa)
6:       hasChanged = FSATransition(α, fsa, t, events)
7:       if inStartState and hasChanged then
8:          fsas = fsas ∪ new FSA
9:       end if
10:      if inFinalState(fsa) then
11:         fsas = {new FSA}
12:         frequency+ = 1
13:      else
14:         fsas = RemoveAllOlderFSAsInSameState(fsas)
15:      end if
16:   end for
17: end for
18: if frequency >= userDefinedThreashold then
19:    EMIT(blockstart − id(α), α)
20: end if
```

similar to α. A sub-episode β of an episode α can be created by removing exactly one node n and all its edges from and to n; e.g., for the episode $A \to B \to C$ the sub-episodes are $A \to B$, $A \to C$ and $B \to C$. The sub-episode of a singleton is denoted by the empty set $\varnothing$.

Frequent episodes can be found by Apriori-like algorithms [2]. The principles of dynamic programming are exploited to combine already frequent episodes to larger ones [12, 11]. We differentiate between alternating *episode generation* and *counting* phases. Every newly generated episodes must be *unique*, *transitive closed*, and *injective*. Candidates possessing infrequent sub-episodes are discarded due to the downward closure lemma [1]. We now present novel counting and episode generation algorithms for processing positional data with Hadoop.

4.1 Counting phase

The *frequency* of an episode is defined as the maximum number of non-overlapping occurrences of the episode in the event stream [11].[1] Non-overlapping episodes can be detected and counted with finite state automata (FSAs), where every FSA is tailored to accept only a particular episode. The idea is as follows. For every episode that needs to be counted, an FSA is created and the event stream is processed by each FSA. If an FSA moves out of the initial state, a new FSA is created for possibly later occurring episodes and once the final state has been

[1] Two occurrences of an episode are said to be *non-overlapping*, if no event associated with one appears in between the events associated with the other.

Algorithm 3 Align(α, β)

Require: $|nodes(\alpha)| = |nodes(\beta)|$
1: f = int array of length $|nodes(\alpha)|$
2: $used$ = boolean array of length $|nodes(\alpha)|$
3: $n = 0$
4: **for** $i = 1$ **to** $|nodes(\alpha)|$ **do**
5: $event_{i,\alpha} = m(\alpha)[i]$
6: $found =$ **false**
7: **for** $j = 1$ **to** $|nodes(\beta)|$ **do**
8: $event_{j,\beta} = m(\beta)[j]$
9: **if** (**not** $used[j]$) **and** $event_{i,\alpha} \sim event_{j,\beta}$ **then**
10: $f[i] = j$
11: $used[j] =$ **true**
12: $found =$ **true**
13: **end if**
14: **end for**
15: **if** $found =$ **false then**
16: $f[i] = -1$
17: $increment(n)$
18: **end if**
19: **end for**
20: **return** f, n

reached, the episode counter is incremented and all FSA-instances of the episode are deleted except for the one still remaining in the initial state.

Algorithm 1 shows the FSA transition function that counts an instance of an episode. Whenever the FSA reaches its final state its frequency is incremented. As input, Algorithm 1 gets the fsa instance which contains the current state, the last transition time and the first transition time. Additionally, the appropriate episode, the current time stamp and the events starting at that time stamp are passed to the function. First, in case the FSA is already in the final state, the function returns without doing anything (line 1). Algorithm 1 iterates over all source nodes in the current state and all events that had happened at time t (line 4-5). Whenever there is an event e that is similar to the appropriate event of source node n (line 6), the FSA is traversed to the next state. The algorithm also keeps track of the start time and the last transition time to check the expiry time (line 9 and line 11).

The FSA transition function allows the definition of the counting algorithm shown in Algorithm 2 as a *map*-function for the Hadoop/MapReduce framework. The function first loads the event stream[2] (line 1) and initialises an empty FSA for every episode. Next, the event stream and the FSAs are traversed and passed to the FSA transition function. Whenever an FSA leaves the start state a new FSA must be added to the set of FSAs. This ensures that there is exactly one

[2] In practice one would read the event stream block wise instead of loading the whole data at once into memory. We chose the latter for ease of presentation.

Algorithm 4 Combine(α,β)

```
1: π, n = Align(α, β)
2: if n ≠ 1 then
3:    return −1
4: end if
5: sum_α = 0; sum_β = 0
6: sum = |π|×(|π|−1)/2
7: for i = 1 to |π| do
8:    if π[i] ≥ −1 then
9:       sum_α = sum_α + i
10:      sum_β = sum_β + π[i]
11:   end if
12: end for
13: return (sum − sum_α, sum − sum_β)
```

FSA in a start state. In case an FSA reaches its final state, all other FSAs can be removed and the process starts again with only one FSA in start state. In case more than one FSA reaches the final state, Algorithm 2 removes all but the youngest one in final state as this one has higher chances to meet the expiry time constraints. The test for expiry time is not shown in the pseudo code. Instances violating the expiry time do not contribute to the frequency count. Neither do FSAs that associate overlapping events with the same object. Note that the general idea of the counting algorithm is very similar to [1]. However, due to the different notions of an event, many optimisation do not apply in our case.

Following [1] we also employ bidirectional evidence as frequencies alone are necessary but not sufficient for detecting frequent episodes. The entropy-based bidirectional evidence can be integrated in the counting algorithm, see [1] for details. We omit the presentation here for lack of space.

4.2 Generation phase

Algorithm 4 is designed to efficiently find the indices of the differentiated nodes of two episodes α and β. Therefore, it first tries to find the bijective mapping π, that maps each node (and its corresponding event) of episode α to episode β (line 1). In case such a complete mapping can not be found, π returns only the possible mappings and n contains the number of missing nodes in the mapping (see Algorithm 3). Episodes α and β are combinable, if and only if $n = 1$. The remainder of the algorithm finds the missing node indices by accumulating over the existing indices and by subtracting the accumulated result from the sum of all indices. This little trick finds the missing indices in time $O(n)$. The function returns the node indices that differentiate between α and β.

To prevent the computation of Algorithm 4 on all pairs of episodes, each episode is associated with its *block start identifier* [1]. The idea is the following. All generated episodes from an episode α share the same sub-episode. This sub-

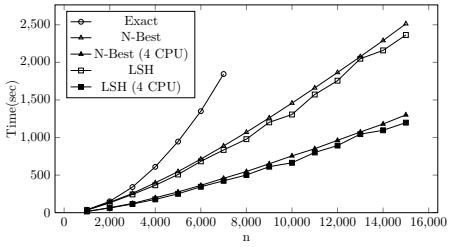
Algorithm 5 Reduce($blockstartId$, xs)

```
1: k = −1; result = ∅
2: for i = 0 to |xs| do
3:    α = xs(i); currentBlockStart = k + 1
4:    for j = i + 1 to |xs| do
5:       β = xs(j)
6:       if α.blockStart == β.blockStart then
7:          candidates = Combine(α, β)
8:          for c ∈ candidates do
9:             if transitiveClose(c) then
10:               c.blockStart = currentBlockStart
11:               result = result ∪ c
12:               k = k + 1
13:            end if
14:         end for
15:      else
16:         break
17:      end if
18:   end for
19: end for
20: EMIT(id, result)
```

episode is trivially identical to α as it originates from adding a node to α. The generation step thus takes only those episodes into account that possess the same block start identifier.

Given two combinable episodes α and β and the differentiated nodes a and b (found by Algorithm 4), it is now possible to combine these episodes to up to three new candidates, as described by [1]. The first candidate originates from adding node b to episode α including all its edges from and to b. The second candidate is generated from the first candidate by adding an edge from node a to node b and the third one adds an edge from b to a to the first candidate. In contrast to [1], we do not test wether all sub-episodes of each candidate are frequent as this would require an efficient lookup of all episodes which can be quite complex for positional data. Candidates with infrequent sub-episodes are allowed at this stage of the algorithm as they will be eliminated in the next counting step anyway.

The complete episode generation algorithm is shown in Algorithm 5. As input, a list of frequent episodes ordered by their block start identifier is given. The result of the algorithm is a list of new episodes that are passed on to the counting algorithm. In line 2 and line 4, all episode pairs are processed as long as they share the same block start identifier (line 6). Then, three possible candidates are generated (line 7) and kept in case they are transitive closed (line 9). Before adding it to the result set, the block start identifier of the new episode is updated (line 10). Analogously to the counting phase, domain specific constraints may be added to filter out unwanted episodes (e.g. in terms of expiry time, overlapping trajectories of the same object, etc.).



n	f_{kim}	f_{keogh}	LSH	Σ
1000	0%	0%	11.42%	11.42%
5000	0.28%	34.00%	16.33%	50.61%
10000	9.79%	41.51%	17.80%	60.10%
15000	17.50%	46.25%	11.82%	75.57%

Fig. 1. Left: Run-time. Right: Pruning efficiency.

5 Empirical Evaluation

5.1 Positional Data

We use positional data from the DEBS Grand Challenge 2013[3]. that is recorded from a soccer game with 8 players per team. We average player data over 100ms to obtain a single measurement for every player at each point in time. The set of trajectories is created by introducing sliding windows that begin every 500ms and last for one second. This procedure gives us 111.041 trajectories in total, 50.212 for team A, 50.245 for team B, and 10.584 for the ball.

5.2 Near Neighbour Search

The first set of experiments evaluates the run-time of the three distance functions *Exact*, *N-Best*, and *LSH*. Since the exact variant needs quadratically many comparisons in the length of the stream, we focus on only a subset of 15,000 consecutive positions of team A in the experiment and fix $N = 1000$. Figure 1 (left) shows the run-times in seconds for varying sample sizes.

Unsurprisingly, the computation time of the exact distances grows exponentially in the size of the data. By contrast, the N-Best algorithm performs slightly super-linear and significantly outperforms its exact counterpart. Prefiltering trajectories using LSH results in only a small additional speed-up. The figure also shows that distributing the computation significantly improves the run-time of the algorithms and indicates that parallelisation allows for computing near-neighbours on large data sets very efficiently. The observed improvements in run-time are the result of a highly efficient pruning strategy (Figure 1, right).

Figure 2 shows the most similar trajectories for three query trajectories. For common trajectories (top rows), the most similar trajectories are true near neighbours. It can also be seen that the proposed distance function is rotation invariant. For uncommon trajectories (bottom row), the found candidates are very different from the query. In the remainder we focus on the N-Best algorithm with for a loss-free and exact computation of the top-N matches.

[3] `http://www.orgs.ttu.edu/debs2013`

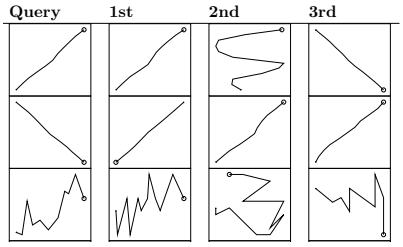


Fig. 2. Most similar trajectories for a given query

5.3 Episode discovery

The first experiments of the episode discovery algorithm focus on the influence of the parameters wrt the number of generated and counted episodes. The algorithm depends on four different parameters, the *similarity*, *frequency*, the *bidirectional evidence*, and the *expiry time*. For this set of experiments, we use the trajectories of team A to find frequent tactical patterns in the game. The results are shown in Figure 3.

The similarity threshold strongly impacts the number of generated episodes: small changes may already lead to an exponential growth in the number of trajectories and large values quickly render the problem infeasible even on medium-sized Hadoop clusters. A similar effect can be observed for the expiry time threshold. Incrementing the expiry time often requires decreasing the similarity threshold. The number of counted episodes is adjusted by the frequency threshold. As shown in the figure, the number of generated episodes can often be reduced by one or more orders of magnitudes. By contrast, the bidirectional evidence threshold affects the result only marginally.

6 Conclusion

We proposed a novel method to mining frequent patterns in positional data streams where consecutive coordinates of objects are treated as movements. We proposed an efficient preprocessing of the positional data using locality sensitive hashing and approximate dynamic time warping and presented a distributed frequent pattern mining algorithm that generalised Achar et al. [1] to positional data at large-scales.

References

1. A. Achar, S. Laxman, R. Viswanathan, and P. S. Sastry. Discovering injective episodes with general partial orders. *Data Min. Knowl. Discov.*, 25(1):67–108, 2012.
2. R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216, 1993.

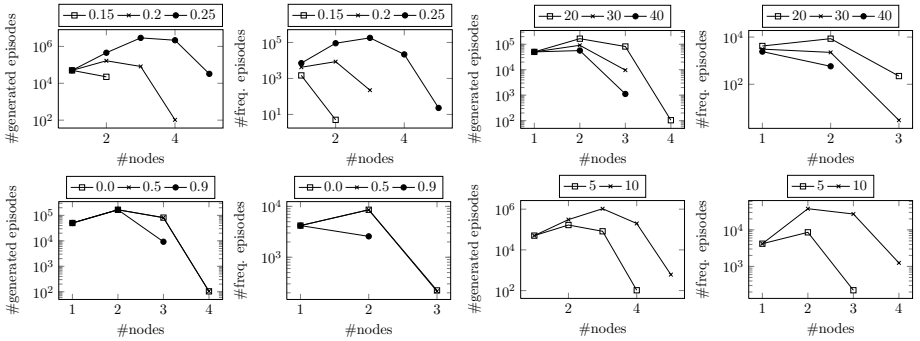


Fig. 3. Top row: Varying similarity (first and second columns) and frequency (third and fourth columns) thresholds. Bottom row: Varying bidirectional evidence (first and second columns) and expiry time (third and fourth columns) thresholds.

3. R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of ICDE*, 1995.
4. V. Athitsos, M. Potamias, P. Papapetrou, and G. Kollios. Nearest neighbor retrieval using distance-based hashing. In *Proc. of ICDE*, 2008.
5. F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi. Trajectory pattern mining. In *Proc. of KDD*, 2007.
6. A. Grunz, D. Memmert, and J. Perl. Tactical pattern recognition in soccer games by means of special self-organizing maps. *Human Movement Science*, 31(2):334–343, 2012.
7. C.-H. Kang, J.-R. Hwang, and K.-J. Li. Trajectory analysis for soccer players. In *Proc. of ICDMW*, 2006.
8. E. Keogh. Exact indexing of dynamic time warping. In *Proc. of VLDB*, 2002.
9. Eamonn J. Keogh and Michael J. Pazzani. Derivative dynamic time warping. In *In First SIAM International Conference on Data Mining (SDM2001*, 2001.
10. S.-W. Kim, S. Park, and W. W. Chu. An index-based approach for similarity search supporting time warping in large sequence databases. In *Proc. of ICDE*, 2001.
11. S. Laxman, P. S. Sastry, and K. P. Unnikrishnan. Discovering frequent episodes and learning hidden markov models: A formal connection. *IEEE Trans. on Knowl. and Data Eng.*, 17(11):1505–1517, November 2005.
12. H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):259–289, 1997.
13. M. Perše, M. Kristan, S. Kovačič, G. Vučkovič, and J. Perš. A trajectory-based analysis of coordinated team activity in a basketball game. *Computer Vision and Image Understanding*, 113(5):612 – 621, 2009.
14. L. Rabiner and B.-H. Juang. *Fundamentals of speech recognition.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
15. N. Tatti and B. Cule. Mining closed episodes with simultaneous events. In *Proc. of KDD*, 2011.
16. M. Vlachos, D. Gunopulos, and G. Das. Rotation invariant distance measures for trajectories. In *Proc. of KDD*, 2004.
17. M. J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Mach. Learn.*, 42(1-2):31–60, 2001.