Efficient Discovery of Sets of Co-occurring Items in Event Sequences

Boris Cule[1(✉)], Len Feremans[1], and Bart Goethals[1,2]

[1] University of Antwerp, Antwerp, Belgium
{boris.cule,len.feremans}@uantwerpen.be
[2] Monash University, Melbourne, Australia

Abstract. Discovering patterns in long event sequences is an important data mining task. Most existing work focuses on frequency-based quality measures that allow algorithms to use the anti-monotonicity property to prune the search space and efficiently discover the most frequent patterns. In this work, we step away from such measures, and evaluate patterns using cohesion—a measure of how close to each other the items making up the pattern appear in the sequence on average. We tackle the fact that cohesion is not an anti-monotonic measure by developing a novel pruning technique in order to reduce the search space. By doing so, we are able to efficiently unearth rare, but strongly cohesive, patterns that existing methods often fail to discover. The data and software related to this paper are available at https://bitbucket.org/len_feremans/sequencepatternmining_public.

1 Introduction

Pattern discovery in sequential data is a well-established field in data mining. The earliest attempts focused on the setting where data consisted of many (typically short) sequences, where a pattern was defined as a (sub)sequence that re-occurred in a high enough number of such input sequences [2].

The first attempt to identify patterns in a single long sequence of data was proposed by Mannila et al. [8]. The presented WINEPI method uses a sliding window of a fixed length to traverse the sequence, and a pattern is then considered frequent if it occurs in a high enough number of these sliding windows. An often-encountered critique of this method is that the obtained frequency is not an intuitive measure, since it does not correspond to the actual number of occurrences of the pattern in the sequence. For example, given sequence $axbcdayb$, and a sliding window length of 3, the frequency of itemset $\{a, b\}$ will be equal to 2, as will the frequency of itemset $\{c, d\}$. However, pattern $\{a, b\}$ occurs twice in the sequence, and pattern $\{c, d\}$ just once, and while the method is motivated by the need to reward c and d for occurring right next to each other, the reported frequency values remain difficult to interpret.

Laxman et al. [7] attempted to tackle this issue by defining the frequency as the maximal number of non-intersecting minimal windows of the pattern in the sequence. In this context, a minimal window of the pattern in the sequence

P. Frasconi et al. (Eds.): ECML PKDD 2016, Part I, LNAI 9851, pp. 361–377, 2016.
DOI: 10.1007/978-3-319-46128-1_23

is defined as a subsequence of the input sequence that contains the pattern, such that no smaller subsequence also contains the pattern. However, while the method uses a relevance window of a fixed length, and disregards all minimal windows that are longer than the relevance window, the length of the minimal windows that do fit into the relevance window is not taken into account at all. For example, given sequence *axyzbcd*, with a relevance window larger than 4, the frequency of both itemset $\{a, b\}$ and itemset $\{c, d\}$ would be equal to 1.

Cule et al. [4] propose an amalgam of the two approaches, defining the frequency of a pattern as the maximal sum of weights of a set of non-overlapping minimal windows of the pattern, where the weight of a window is defined as the inverse of its length. However, this method, too, struggles with the interpretability of the proposed measure. For example, given sequence *axbcdayb* and a relevance window larger than 3, frequency of $\{a, b\}$ would be $2/3$, while frequency of $\{c, d\}$ would be $1/2$. On top of this, as the input sequence grows longer, the sum of these weights will grow, and the defined frequency can take any real positive value, giving the user no idea how to set a sensible frequency threshold.

All of the techniques mentioned above use a frequency measure that satisfy the so-called Apriori property [1]. This property implies that the frequency of a pattern is never smaller than the frequency of any of its superpatterns (in other words, frequency is an *anti-monotonic* quality measure). While this property is computationally very desirable, since large candidate patterns can be generated from smaller patterns, and generating unnecessary candidates can be avoided, the undesirable side-effect is that larger patterns, which are often more useful to the end users, will never be ranked higher than all their subpatterns. On top of this, all these methods focus solely on how often certain items occur near each other, and do not take occurrences of these items far away from each other into account. Consequently, if two items occur frequently, and through pure randomness often occur near each other, they will form a frequent itemset, even though they are, in fact, in no way correlated.

In another work, Cule et al. [3] propose a method that steps away from anti-monotonic quality measures, and introduce a new interestingness measure that combines the coverage of the pattern with its cohesion. Cohesion is defined as a measure of how near each other the items making up an interesting itemset occur on average. However, the authors define the coverage of an itemset as the sum of frequencies of all items making up the itemset, which results in a massive bias towards larger patterns instead. Furthermore, this allows for a very infrequent item making its way into an interesting itemset, as long as all other items in the itemset are very frequent and often occur near the infrequent item. As a result, the method is not scalable for any sequence with a large alphabet of items, which makes it unusable in most realistic data sets.

Hendrickx et al. [6] tackle a related problem in an entirely different setting. Given a graph consisting of labelled nodes, they attempt to discover which labels often co-occur. In this context, they aim to discover cohesive itemsets (sets of labels), by computing average distances between the labels, where the distance between two nodes is defined as the length of the shortest path between them

(expressed as the number of edges on this path). While the authors also experimented with sequential data, after first converting an input sequence into a graph, by converting each event into a node labelled by the event type, and connecting neighbouring events by an edge, this approach is not entirely suitable for sequential data. More precisely, in a graph setting, an itemset can only be considered fully cohesive if all its occurrences form a clique in the graph. Clearly, in a sequence, for any itemset of size larger than 2, it would be impossible to form a clique, since each node (apart from the first one and the last one) has exactly two edges – one connecting the node to the event that occurred last before the event itself, and the other connecting it to the event that occurred first after the event itself. For example, given sequence *abcd* (converted into a graph), the cohesion of itemset $\{a, b\}$ would be equal to 1, the cohesion of $\{a, b, c\}$ would be $3/4$, and the cohesion of $\{a, b, c, d\}$ would be $3/5$ (we omit the computational details here), which is clearly not intuitive.

In this work, we use the cohesion introduced by Cule et al. [3] as a single measure to evaluate cohesive itemsets. We consider itemsets as potential candidates only if each individual item contained in the itemset is frequent in the dataset. This allows us to filter out the infrequent items at the very start of our algorithm, without missing out on any cohesive itemsets. However, using cohesion as a single measure brings its own computational problems. First of all, cohesion is not an anti-monotonic measure, which means that a superset of a non-cohesive itemset could still prove to be cohesive. However, since the size of the search space is exponential in the number of frequent items, it is impossible to evaluate all possible itemsets. We solve this by developing a tight upper bound on the maximal possible cohesion of all itemsets that can still be generated in a particular branch of the depth-first-search tree. This bound allows us to prune large numbers of potential candidate itemsets, without having to evaluate them at all. Furthermore, we present a very efficient method to identify minimal windows that contain a particular itemset, necessary to evaluate its cohesion. Our experiments show that our method discovers patterns that existing methods struggle to rank highly, while dismissing obvious patterns consisting of items that occur frequently, but are not at all correlated. We further show that we achieve these results quickly, thus demonstrating the efficiency of our algorithm.

The rest of the paper is organised as follows. In Sect. 2 we formally describe the problem setting and define the patterns we aim to discover. Section 3 provides a detailed description of our algorithm, while in Sect. 4 we present a thorough experimental evaluation of our method, in comparison with a number of existing methods. We present an overview of the most relevant related work in Sect. 5, before summarising our main conclusions in Sect. 6.

2 Problem Setting

The dataset consists of a single event sequence $s = (e_1, \ldots, e_n)$. Each event e_k is represented by a pair (i_k, t_k), with i_k an event type (coming from the domain of all possible event types) and t_k an integer time stamp. For any $1 < k \leq n$, it

holds that $t_k > t_{k-1}$. For simplicity, we omit the time stamps from our examples, and write sequence $(e_1, \ldots, e_n)$ as $i_1 \ldots i_n$, implicitly assuming the time stamps are consecutive integers starting with 1. In further text, we refer to event types as *items*, and sets of event types as *itemsets*.

For an itemset $X = \{i_1, ..., i_m\}$, we denote the set of occurrences of items making up X in a sequence s with $N(X) = \{t|(i,t) \in s, i \in X\}$. For an item i, we define the *support* of i in an input sequence s as the number of occurrences of i in s, $sup(i) = |N(\{i\})|$. Given a user-defined support threshold *min_sup*, we say that an itemset X is *frequent* in a sequence s if for each $i \in X$ it holds that $sup(i) \geq min\_sup$.

To evaluate the cohesiveness of an itemset X in a sequence s, we must first identify minimal occurrences of the itemset in the sequence. For each occurrence of an item in X, we will look for the minimal window within s that contains that occurrence and the entire itemset X. Formally, given a time stamp t, such that $(i,t) \in s$ and $i \in X$, we define the *size of the minimal occurrence of X around t* as $W_t(X) = \min\{t_e - t_s + 1 | t_s \leq t \leq t_e \text{ and } \forall i \in X\ \exists (i,t') \in s, t_s \leq t' \leq t_e\}$.

We further define the *size of the average minimal occurrence of X in s* as

$$\overline{W}(X) = \frac{\sum_{t \in N(X)} W_t(X)}{|N(X)|}.$$

Finally, we define the *cohesion* of itemset X, with $|X| > 0$, in a sequence s as $C(X) = \frac{|X|}{\overline{W}(X)}$. If $|X| = 0$, we define $C(X) = 1$.

Given a user-defined cohesion threshold *min_coh*, we say that an itemset X is *cohesive* in a sequence s if it holds that $C(X) \geq min\_coh$.

Note that the cohesion is higher if the minimal occurrences are smaller. Furthermore, a minimal occurrence of itemset X can never be smaller than the size of X, so it holds that $C(X) \leq 1$. If $C(X) = 1$, then every single minimal occurrence of X in s is of length $|X|$.

A single item is always cohesive, so to avoid outputting all frequent items, we will from now on consider only itemsets consisting of 2 or more items. An optional parameter, *max_size*, can be used to limit the size of the discovered patterns. Formally, we say that an itemset X is a *frequent cohesive itemset* if $1 < |X| \leq max\_size$, $\forall i \in X : sup(i) \geq min\_sup$ and $C(X) \geq min\_coh$.

Cohesion is not an anti-monotonic measure. A superset of a non-cohesive itemset could turn out to be cohesive. For example, given sequence *abcxacbybac*, we can see that $C(\{a,b\}) = C(\{a,c\}) = C(\{b,c\}) = 6/7$, while $C(\{a,b,c\}) = 1$. While this allows us to eliminate bias towards smaller patterns, it also brings computational challenges which will be addressed in the following section.

3 Algorithm

In this section we present a detailed description of our algorithm. We first show how we generate candidates in a depth-first manner, before explaining how we can prune large numbers of potential candidates by computing an upper bound

Algorithm 1. FCI_{SEQ} finds frequent cohesive itemsets in a sequence

```
1 FI = all frequent items;
2 sort FI on support in ascending order;
3 FC = ∅;
4 DFS(⟨∅, FI⟩);
5 return FC
```

Algorithm 2. $\text{DFS}(\langle X, Y \rangle)$ depth-first search

```
1 if C_max(X, Y) ≥ min_coh then
2     if Y = ∅ then
3         if |X| > 1 then
4             FC = FC ∪ {X};
5     else
6         a = first(Y);
7         if |X ∪ {a}| ≤ max_size then
8             DFS(⟨X ∪ {a}, Y \ {a}⟩);
9         DFS(⟨X, Y \ {a}⟩);
```

of the cohesion of all itemsets that can be generated within a branch of the search tree, and we end the section by providing an efficient method to compute the sum of minimal windows of a particular itemset in the input sequence.

3.1 Depth-First-Search

The main routine of our FCI_{SEQ} algorithm is given in Algorithm 1. We begin by scanning the input sequence, identifying the frequent items, and storing their occurrence lists for later use. We then sort the set of frequent items on support in ascending order (line 2), initialise the set of frequent cohesive itemsets FC as an empty set (line 3), and start the depth-first-search process (line 4). Once the search is finished, we output the set of frequent cohesive itemsets FC (line 5).

The recursive DFS procedure is shown in Algorithm 2. In each call, X contains the candidate itemset, while Y contains items that are yet to be enumerated. In line 1, we evaluate the pruning function $C_{max}(X, Y)$ to decide whether to search deeper in the tree or not. This function will be described in detail in Sect. 3.2. If the branch is not pruned, there are two possibilities. If we have reached a leaf node (line 2), we add the discovered cohesive itemset to the output (provided its size is greater than 1). Alternatively, if there are more items to be enumerated, we pick the first such item a (line 6) and make two recursive calls to the DFS function—the first with a added to X (this is only executed if $X \cup \{a\}$ satisfies the *max_size* constraint), and the second with a discarded.

3.2 Pruning

At any node in the search tree, X denotes all items currently making up the candidate itemset, while Y denotes all items that are yet to be enumerated. Starting from such a node, we can still generate any itemset Z, such that $X \subseteq Z \subseteq X \cup Y$ and $|Z| \leq max\_size$. In order to be able to prune the entire branch of the search tree, we must therefore be certain that for every such Z, the cohesion of Z cannot satisfy the minimum cohesion constraint.

In the remainder of this section, we first define an upper bound for the cohesion of all itemsets that can be generated in a particular branch of the search tree, before providing a detailed proof of its soundness. Given itemsets X and Y, with $|X| > 0$ and $X \cap Y = \emptyset$, the $C_{max}(X, Y)$ pruning function used in line 1 of Algorithm 2 is defined as

$$C_{max}(X,Y) = \frac{\min(max\_size, |X \cup Y|)(|N(X)| + N_{max}(X,Y))}{\sum_{t \in N(X)} W_t(X) + \min(max\_size, |X \cup Y|) N_{max}(X,Y)},$$

where

$$N_{max}(X,Y) = \max_{\substack{Y_i \subseteq Y, \\ |Y_i| \leq max\_size - |X|}} |N(Y_i)|.$$

For $|X| = 0$, we define $C_{max}(X, Y) = 1$.

Note that if $Y = \emptyset$, $C_{max}(X, Y) = C(X)$, which is why we do not need to evaluate $C(X)$ before outputting X in line 4 of Algorithm 2.

Before proving that the above upper bound holds, we will first explain the intuition behind it. When we find ourselves at node $\langle X, Y \rangle$ of the search tree, we will first evaluate the cohesion of itemset X. If X is cohesive, we need to search deeper in the tree, as supersets of X could also be cohesive. However, if X is not cohesive, we need to evaluate how much the cohesion can still grow if we go deeper into this branch of the search tree. Logically, starting from $C(X) = \frac{|X|}{W(X)} = \frac{|X||N(X)|}{\sum_{t \in N(X)} W_t(X)}$, the value of this fraction will grow maximally if the nominator is maximised, and the denominator minimised. Clearly, as we add items to X, the nominator will grow, and it will grow maximally if we add as many items to X as possible. However, as we add items to X, the denominator must grow, too, so the question is how it can grow minimally. In the worst case, each new window added to the sum in the denominator will be minimal (i.e., its length will be equal to the size of the new itemset), and the more such windows we add to the sum, the higher the overall cohesion will grow.

For example, given sequence acb and a cohesion threshold of 0.8, assume we find ourselves in node $\langle \{a, b\}, \{c\} \rangle$ of the search tree. We will then first find the smallest windows containing $\{a, b\}$ for each occurrence of a and b, i.e., $W_1(\{a, b\}) = W_3(\{a, b\}) = 3$. It turns out that $C(\{a, b\}) = \frac{2 \times 2}{3+3} = \frac{2}{3}$, which is not cohesive enough. However, if we add c to itemset $\{a, b\}$, we know that the size of the new itemset will be 3, we know the number of occurrences of items from the new itemset will be 3, and the nominator will therefore be equal to 9.

For the denominator, we have no such certainties, but we know that, in the worst case, the windows for the occurrences of a and b will not grow (i.e., each smallest window of $\{a, b\}$ will already contain an occurrence of c), and the windows for all occurrences of c will be minimal (i.e., of size 3). Indeed, when we evaluate the above upper bound, we obtain $C_{max}(\{a, b\}, \{c\}) = \frac{3\times(2+1)}{6+3\times1} = \frac{9}{9} = 1$. We see that even though the cohesion of $\{a, b\}$ is $\frac{2}{3}$, the cohesion of $\{a, b, c\}$ could, in the worst case, be as high as 1. And in our sequence acb, that is indeed the case. The above example also demonstrates the tightness of our upper bound, as the computed value can, in fact, turn out to be equal to the actual cohesion of a superset yet to be generated.

We now present a full formal proof of the soundness of the proposed upper bound. In order to do this, we will need the following lemma.

Lemma 1. *For any six positive numbers a, b, c, d, e, f, with $a \leq b$, $c \leq d$ and $e \leq f$, it holds that*

1. *if $\frac{a+c+e}{b+e} < 1$ then $\frac{a+c+e}{b+e} \leq \frac{a+d+f}{b+f}$.*
2. *if $\frac{a+c+e}{b+e} \geq 1$ then $\frac{a+d+f}{b+f} \geq 1$.*

Proof. We begin by proving the first claim. To start with, note that if $\frac{a+c+e}{b+e} < 1$, then $\frac{a+c}{b} < 1$. For any positive number f with $e \leq f$, it therefore follows that $\frac{a+c+e}{b+e} \leq \frac{a+c+f}{b+f}$. Finally, for any positive number d, with $c \leq d$, it holds that $\frac{a+c+f}{b+f} \leq \frac{a+d+f}{b+f}$, and therefore $\frac{a+c+e}{b+e} \leq \frac{a+d+f}{b+f}$. For the second claim, it directly follows that if $\frac{a+c+e}{b+e} \geq 1$, then $\frac{a+c}{b} \geq 1$, $\frac{a+d}{b} \geq 1$, and $\frac{a+d+f}{b+f} \geq 1$. □

Theorem 1. *Given itemsets X and Y, with $X \cap Y = \emptyset$, for any itemset Z, with $X \subseteq Z \subseteq X \cup Y$ and $|Z| \leq$ max_size, it holds that $C(Z) \leq C_{max}(X, Y)$.*

Proof. We know that $C(Z) \leq 1$, so the theorem holds if $|X| = 0$. Assume now that $|X| > 0$. First recall that $C(Z) = \frac{|Z|}{W(Z)} = \frac{|Z||N(Z)|}{\sum_{t\in N(Z)} W_t(Z)}$. We can rewrite this expression as

$$C(Z) = \frac{(|X| + |Z\backslash X|)(|N(X)| + |N(Z\backslash X)|)}{\sum_{t\in N(X)} W_t(Z) + \sum_{t\in N(Z\backslash X)} W_t(Z)}.$$

Further note that for a given time stamp in $N(X)$, the minimal window containing Z must be at least as large as the minimal window containing only X, and for a given time stamp in $N(Z\backslash X)$, the minimal window containing Z must be at least as large as the size of Z. It therefore follows that

$$\sum_{t\in N(X)} W_t(Z) \geq \sum_{t\in N(X)} W_t(X) \text{ and } \sum_{t\in N(Z\backslash X)} W_t(Z) \geq |Z||N(Z\backslash X)|,$$

and, as a result,

$$C(Z) \leq \frac{|X||N(X)| + |Z\backslash X||N(X)| + |Z||N(Z\backslash X)|}{\sum_{t \in N(X)} W_t(X) + |Z||N(Z\backslash X)|}.$$

Finally, we note that, per definition, $|Z\backslash X| \leq \min(max\_size, |X \cup Y|) - |X|$, and, since Z is generated by adding items from Y to X, until either $|Z| = max\_size$ or there are no more items left in Y, $|N(Z\backslash X)| \leq N_{max}(X, Y)$.

At this point we will use Lemma 1 to take the proof further. Note that, per definition, $C(X) = \frac{|X||N(X)|}{\sum_{t \in N(X)} W_t(X)} \leq 1$. We now denote

$$a = |X||N(X)| \text{ and } b = \sum_{t \in N(X)} W_t(X).$$

Furthermore, we denote

$$c = |Z\backslash X||N(X)|,\ d = (\min(max\_size, |X \cup Y|) - |X|)|N(X)|,$$
$$e = |Z||N(Z\backslash X)|, \text{ and } f = \min(max\_size, |X \cup Y|)N_{max}(X, Y).$$

Since a, b, c, d, e and f satisfy the conditions of Lemma 1, we know that it holds that

1. if $\frac{|X||N(X)|+|Z\backslash X||N(X)|+|Z||N(Z\backslash X)|}{\sum_{t \in N(X)} W_t(X)+|Z||N(Z\backslash X)|} < 1$ then
$\frac{|X||N(X)|+|Z\backslash X||N(X)|+|Z||N(Z\backslash X)|}{\sum_{t \in N(X)} W_t(X)+|Z||N(Z\backslash X)|} \leq$
$\frac{|X||N(X)|+(\min(max\_size,|X\cup Y|)-|X|)|N(X)|+\min(max\_size,|X\cup Y|)N_{max}(X,Y)}{\sum_{t \in N(X)} W_t(X)+\min(max\_size,|X\cup Y|)N_{max}(X,Y)}$.
2. if $\frac{|X||N(X)|+|Z\backslash X||N(X)|+|Z||N(Z\backslash X)|}{\sum_{t \in N(X)} W_t(X)+|Z||N(Z\backslash X)|} \geq 1$ then
$\frac{|X||N(X)|+(\min(max\_size,|X\cup Y|)-|X|)|N(X)|+\min(max\_size,|X\cup Y|)N_{max}(X,Y)}{\sum_{t \in N(X)} W_t(X)+\min(max\_size,|X\cup Y|)N_{max}(X,Y)} \geq 1$.

Finally, note that
$\frac{|X||N(X)|+(\min(max\_size,|X\cup Y|)-|X|)|N(X)|+\min(max\_size,|X\cup Y|)N_{max}(X,Y)}{\sum_{t \in N(X)} W_t(X)+\min(max\_size,|X\cup Y|)N_{max}(X,Y)} =$
$\frac{\min(max\_size,|X\cup Y|)(|N(X)|+N_{max}(X,Y))}{\sum_{t \in N(X)} W_t(X)+\min(max\_size-|X|,|Y|)N_{max}(X,Y)} = C_{max}(X, Y)$.
From the first claim above, it follows that if $\frac{|X||N(X)|+|Z\backslash X||N(X)|+|Z||N(Z\backslash X)|}{\sum_{t \in N(X)} W_t(X)+|Z||N(Z\backslash X)|}$ < 1, then $C(Z) \leq C_{max}(X, Y)$. From the second claim, it immediately follows that if $\frac{|X||N(X)|+|Z\backslash X||N(X)|+|Z||N(Z\backslash X)|}{\sum_{t \in N(X)} W_t(X)+|Z||N(Z\backslash X)|} \geq 1$, then $C_{max}(X, Y) \geq 1$, and since, per definition $C(Z) \leq 1$, $C(Z) \leq C_{max}(X, Y)$. This completes the proof. □

Since an important feature of computing an upper bound for the cohesion is to establish how much cohesion could grow *in the worst case*, we need to figure out which items from Y should be added to X to reach this worst case. As has been discussed above, the worst case is actually materialised by adding as many as possible items from Y, and by first adding those that have the most occurrences. However, if the *max_size* parameter is used, it is not always possible to add all items in Y to X. In this case, we can only add $max\_size - |X|$ items to X, which is why we defined $N_{max}(X, Y)$ as

$$N_{max}(X,Y) = \max_{\substack{Y_i \subseteq Y, \\ |Y_i| \leq max\_size - |X|}} |N(Y_i)|.$$

Clearly, if $|X \cup Y| \leq max\_size$, $N_{max}(X,Y) = N(Y)$. If not, at first glance it may seem computationally very expensive to determine $|N(Y_i)|$ for every possible Y_i. However, we solve this problem by sorting the items in Y on support in ascending order. In other words, if $Y = \{y_1, \ldots, y_n\}$, with $sup(y_i) \leq sup(y_{i+1})$ for $i = 1, \ldots, n-1$, then we can compute $N_{max}(X,Y)$ as

$$N_{max}(X,Y) = \sum_{i \in \{1,\ldots,max\_size-|X|\}} |N(\{y_{n-i+1}\})|.$$

As a result, the only major step in computing $C_{max}(X,Y)$ is that of computing $\sum_{t \in N(X)} W_t(X)$, as the rest can be computed in constant time. The procedure for computing $\sum_{t \in N(X)} W_t(X)$ is explained in detail in Sect. 3.3.

3.3 Computing the Sum of Minimal Windows

The algorithm for computing the sum of minimal windows is shown in Algorithm 3. For a given itemset X, the algorithm keeps a list of all time stamps at which items of X occur in the *positions* variable. The *nextpos* variable keeps a list of next time stamps for each item, while *lastpos* keeps a list of the last occurrences for each item. Since we need to compute the minimal window for each occurrence, we keep on doing this until we have either computed them all, or until the running sum has become large enough to safely stop, knowing that the branch can be pruned (line 7). Concretely, by rewriting the definition of $C_{max}(X,Y)$, we know we can stop if we are certain the sum will be larger than

$$W_{max}(X,Y) = \frac{\min(max\_size, |X \cup Y|)(|N(X)| + (N_{max}(X,Y)(1 - min\_coh)))}{min\_coh}.$$

When a new item comes in (line 14), we update the working variables, and compute the first and last position of the current window (line 17). If the smallest time stamp of the current window has changed, we go through the list of active windows and check whether a new shortest length has been found. If so, we update it (line 21). We then remove all windows for which we are certain that they cannot be improved from the list of active windows (line 23), and update the overall sum (line 24). Finally, we add the new window for the current time stamp to the list of active windows (line 25).

Note that the sum of minimal windows is independent of Y, the items yet to be enumerated. Therefore, if the branch is not pruned, the recursive DFS procedure shown in Algorithm 2 will be called twice, but X will remain unchanged in the second of those calls (line 9), so we will not need to recompute the sum of windows, allowing us to immediately evaluate the upper bound in the new node of the search tree.

Algorithm 3. SUM_MIN_WINS($\langle X, Y \rangle$) sums minimal windows of X

```
1  smw ← 0; index ← 0;
2  positions ← position for every item in X;
3  nextpos ← {positions[i_1][0], positions[i_2][0], positions[i_3][0], ...};
4  lastpos ← {−∞, −∞, −∞, ...};
5  prev_min ← −∞; active_windows ← ∅;
6  while index < |N(X)| do
7      if smw + (|N(X)| + |active_windows| − index) × |X| > W_max(X, Y) then
8          return ∞;
9      current_pos ← ∞;
10     current_item ← ∅;
11     for i in X do
12         if current_pos > nextpos[i] then
13             current_pos ← nextpos[i];
14             current_item ← i;
15     lastpos[current_item] ← current_pos;
16     nextpos[current_item] ← next(positions[current_item], current_pos);
17     minpos ←min(lastpos); maxpos ←max(lastpos);
18     if minpos ≠ −∞ and minpos > prev_min then
19         for window ∈ active_windows do
20             newwidth ← maxpos−min(minpos, window.pos) + 1;
21             window.width ←min(window.width, newwidth);
22             if window.pos < minpos or window.width == |X| or
                window.width < (maxpos − window.pos + 1) then
23                 active_windows ← active_windows\{window};
24                 smw ← smw + window.width;
25     active_windows ←
        active_windows ∪ {window(current_pos, maxpos − minpos + 1)};
26     prev_min ← minpos; index ← index + 1;
27 smw ← smw+sum(window.width|window ∈ active_windows);
28 return smw;
```

We illustrate how the algorithm works on the following example. Assume we are given the input sequence *aabccccacb*, and we are evaluating itemset $\{a, b, c\}$. Table 1 shows the values of the main variables as the algorithm progresses. As each item comes in, we update the values of *nextpos* and *lastpos* (other variables are not shown in the table). In each iteration, we compute the current best minimal window for the given time stamp as max(*lastpos*)−min(*lastpos*)+1. We also update the values of any previous windows that might have changed for the better (this can only happen if min(*lastpos*) has changed), using either the current window above if it contains the time stamp of the window's event, or the window stretching from the relevant time stamp to max(*lastpos*). Finally, before proceeding with the next iteration, we remove all windows for which we are certain that they cannot get any smaller from the list of active windows.

Table 1. Computation of minimal windows.

Time	Item	*nextpos*	*lastpos*	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}
0	-	$(1,3,4)$	$(-\infty,-\infty,-\infty)$	-	-	-	-	-	-	-	-	-	-
1	a	$(2,3,4)$	$(1,-\infty,-\infty)$	∞	-	-	-	-	-	-	-	-	-
2	a	$(8,3,4)$	$(2,-\infty,-\infty)$	∞	∞	-	-	-	-	-	-	-	-
3	b	$(8,10,4)$	$(2,3,-\infty)$	∞	∞	∞	-	-	-	-	-	-	-
4	c	$(8,10,5)$	$(2,3,4)$	**4**	**3**	**3**	**3**	-	-	-	-	-	-
5	c	$(8,10,6)$	$(2,3,5)$	-	-	-	-	4	-	-	-	-	-
6	c	$(8,10,7)$	$(2,3,6)$	-	-	-	-	4	5	-	-	-	-
7	c	$(8,10,9)$	$(2,3,7)$	-	-	-	-	4	5	6	-	-	-
8	a	$(\infty,10,9)$	$(8,3,7)$	-	-	-	-	**4**	5	6	6	-	-
9	c	$(\infty,10,\infty)$	$(8,3,9)$	-	-	-	-	-	5	6	6	7	-
10	b	(∞,∞,∞)	$(8,10,9)$	-	-	-	-	-	**5**	**4**	**3**	**3**	**3**

In the table, windows that are not active are marked with '-', while definitively determined windows are shown in bold. We can see that, for example, at time stamp 4, we have determined the value of the first four windows. Window w_1 cannot be improved on, since time stamp 1 has already dropped out of *lastpos*, while the other three windows cannot be improved since 3 is the absolute minimum for a window containing three items. At time stamp 8, we know that the length of w_5 must be equal to 4, since any new window to come must stretch at least from time stamp 5 to a time stamp in the future, i.e., at least 9. Finally, once we have reached the end of the sequence, we mark all current values of still active windows as determined.

4 Experiments

In order to demonstrate the usefulness of our method, we chose datasets in which the discovered patterns could be easily discussed and explained. We used two text datasets, the *Species* dataset containing the complete text of *On the Origin of Species by Means of Natural Selection* by Charles Darwin[1], and the *Moby* dataset containing *Moby Dick* by Herman Melville[2]. We processed both sequences using the Porter Stemmer[3] and removed the stop words. After preprocessing, the length of the *Species* dataset was 85 450 items and the number of distinct items was 5 547, and the length of *Moby* was 88 945 and the number of distinct items was 10 221.

We performed two types of experiments. In Sect. 4.1, we qualitatively compare our output to that of three existing methods, while in Sect. 4.2, we provide

[1] Taken from http://www.gutenberg.org/etext/22764.
[2] Taken from http://www.gutenberg.org/etext/15.
[3] http://tartarus.org/~martin/PorterStemmer/.

a performance analysis of our FCI_{SEQ} algorithm. To ensure reproducibility, we have made our implementations and datasets publicly available[4].

4.1 Quality of Output

In the first set of experiments, we compared the patterns we discovered to those found by three existing pattern mining algorithms—WINEPI, LAXMAN[5] and $\text{MARBLES}_{\text{W}}$[6]. As discussed in Sect. 1, these algorithms use a variety of frequency-based quality measures to evaluate the patterns. Since the available implementations were made with the goal of discovering partially ordered episodes, we had to post-process the output in order to filter out only itemsets. Therefore, making any kind of runtime comparisons would be unfair on these methods, since they generate many more candidates. Consequently, in this section we limit ourselves to a qualitative analysis of the output.

For all methods, we set the relevant thresholds low enough in order to generate tens of thousands of patterns. We then sorted the output on the respective quality measures—the sliding window frequency for WINEPI, the non-overlapping minimal window frequency for LAXMAN, the weighted window frequency for $\text{MARBLES}_{\text{W}}$, and cohesion for FCI_{SEQ}. We used pattern size to break any ties in all four methods, and the sum of support of individual items making up an itemset as the third criteria for FCI_{SEQ}. Patterns that were still tied were ordered alphabetically. The frequency threshold was set to 30 for WINEPI in both datasets, 5 for LAXMAN in *Origin* and 4 in *Moby*, and 1 for $\text{MARBLES}_{\text{W}}$ in both datasets, with the sliding window size set to 15. We ran FCI_{SEQ} with the cohesion threshold set to 0.01, and the support threshold to 5 for *Origin* and 4 for *Moby*. Since none of the existing methods produced any itemsets consisting of more than 5 items, we limited the *max_size* parameter to 5.

The top 5 patterns discovered by the different methods are shown in Table 2. We can see that there are clear differences between the patterns we discovered and those discovered by the existing methods, which all produced very similar results. First of all, the patterns ranked first and second in our output for the *Origin* dataset are of size 3, which would be theoretically impossible for WINEPI and $\text{MARBLES}_{\text{W}}$, and highly unlikely for LAXMAN, since all three use anti-monotonic quality measures. Second, we observe that the patterns we discover are in fact quite rare in the dataset, but they are very strong, since all occurrences of these patterns are highly cohesive. Concretely, the phrase *tierra del fuego* occurs seven times in the book, and none of these words occur anywhere else in the book. The value of this pattern is therefore quite clear—if we encounter any one of these three words, we can be certain that the other two can be found nearby. However, the only other method that ranked this pattern in the top 10 000 was $\text{MARBLES}_{\text{W}}$, which ranked it $8\,357^{\text{th}}$. On the other hand, pattern *mobi dick* is

[4] https://bitbucket.org/len_feremans/sequencepatternmining_public.

[5] The algorithm was given no name by its authors.

[6] The implementations of all three methods were downloaded from http://users.ics.aalto.fi/ntatti/software/closedepisodeminer.zip.

Table 2. Top 5 patterns discovered by the different methods.

Dataset	FCI_{SEQ}	Winepi	Laxman	$\text{Marbles}_{\text{W}}$
Species	tierra del fuego	natur select	natur select	natur select
	natura facit saltum	speci varieti	speci form	speci varieti
	del tierra	speci form	speci varieti	speci distinct
	del fuego	speci natur	speci natur	speci form
	natura facit	speci distinct	speci distinct	life condit
Moby	mobi dick	whale sperm	whale boat	whale sperm
	vinegar cruet	whale boat	ship whale	whale white
	deuteronomi deacon	ship whale	whale sperm	ship whale
	defend plaintiff	whale white	head whale	whale boat
	erskin defend plaintiff	head whale	sea whale	head mast

both cohesive and frequent, and was ranked 14^{th} by Winepi, 22^{nd} by Laxman and 7^{th} by $\text{Marbles}_{\text{W}}$. None of the other patterns in our top 5 in either dataset were ranked in the top 3 000 patterns by any of the other algorithms. We conclude that in order to find the very strong, but rare, patterns, such as *tierra del fuego* or *vinegar cruet*, with the existing methods the user would need to wait a long time before a huge output was generated, and would then need to trawl through tens of thousands of itemsets in the hope of finding them. Our algorithm, on the other hand, ranks them at the very top.

The patterns discovered by other methods typically consist of words that occur very frequently in the book, regardless of whether the occurrences of the words making up the itemset are correlated or not. For example, words *speci* and *varieti* occur very often, and, therefore, also often co-occur. In fact, this pattern was ranked $82\,261^{\text{st}}$ by FCI_{SEQ}, with a cohesion of just over 0.01, indicating that the average distance between nearest occurrences of *speci* and *varieti* was close to 100, which clearly demonstrates that this pattern is spurious. Clearly, while the very top patterns seem very different, there was still some overlap between the output generated by the various methods. For example, pattern *natur select*, ranked first in the *Origin* dataset by the existing methods, was ranked 17^{th} by FCI_{SEQ}, which shows that our method is also capable of discovering very frequent patterns, as long as they are also cohesive. Similarly, pattern *life condit* ranked 66^{th} in our output, and pattern *speci distinct* 129^{th}.

Table 3 shows the size of the overlap between the patterns discovered by FCI_{SEQ} and those discovered by the other three methods. We compute the size of the overlap within the top k patterns for each method, for varying values of k. We note that, in relative terms, the overlap actually drops as k grows, since our method ranks many large patterns highly, which is not the case for the other three methods. For example, the top 500 patterns discovered by FCI_{SEQ} on the *Species* dataset contain 388 patterns of size larger than 3, while the top 500 patterns produced by the other three methods do not contain a single pattern of

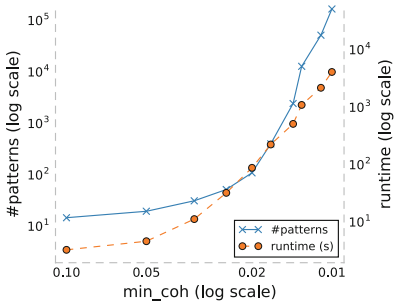
Table 3. Overlap in the top k patterns discovered by FCI_{SEQ} and other methods.

Dataset	k	WINEPI	LAXMAN	$\text{MARBLES}_{\text{W}}$
Species	100	12	13	11
	500	28	35	25
	2 500	80	102	68
Moby	100	11	13	11
	500	27	28	26
	2 500	56	66	49

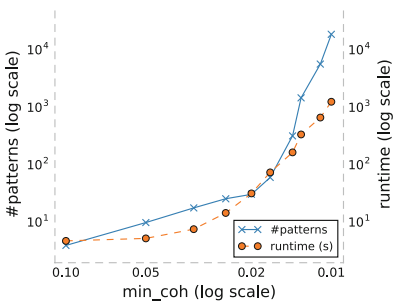
size larger than 3. Even in the top 100 patterns we discovered in both datasets less than half were of size 2, while the top 100 produced by all other methods on either dataset always contained at least 96 patterns of size 2. This demonstrates the benefit of using a non-anti-monotonic quality measure, which allows us to rank the best patterns on top regardless of size, while frequency-based methods will, per definition, rank all subpatterns of a large pattern higher than (or at least as high as) the pattern itself.

4.2 Performance Analysis

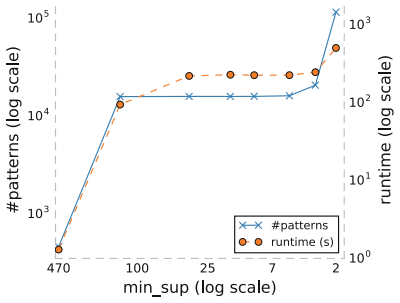
We tested the behaviour of our algorithm when varying the three thresholds. The results are shown in Fig. 1. As expected, we see that the number of patterns increases as the cohesion and support thresholds are lowered. In particular, when the cohesion threshold is set too low, the size of the output explodes, as even random combinations of frequent items become cohesive enough. However, as the support threshold decreases, the number of patterns stabilises, since rarer items typically only make up cohesive itemsets with each other, so only a few new patterns are added to the output (when we lower the support threshold to 2, we see another explosion as nearly the entire alphabet is considered frequent). In all settings, it took no more than a few minutes to find tens of thousands of patterns. With reasonable support and cohesion thresholds, we could even set the *max_size* parameter to ∞ without encountering prohibitive runtimes, allowing us to discover patterns of arbitrary size (in practice, the size of the largest pattern is limited due to the characteristics of the data, so output size stops growing at a certain point). Since existing methods use a relevance window, defining how far apart two items may be in order to still be considered part of a pattern, the existing methods can never achieve this. For example, using a window of size 15 implies that no pattern consisting of more than 15 items can be discovered. Finally, while we kept both *min_sup* and *min_coh* relatively high in the presented experiments with *max_size* set to ∞, it should be noted that a much lower *min_sup* could be used in combination with a higher *min_coh* in order to quickly find only the most cohesive patterns, including the rare ones.



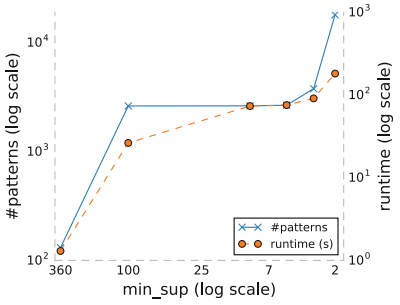
(a) $min\_sup = 4$, $max\_size = 5$



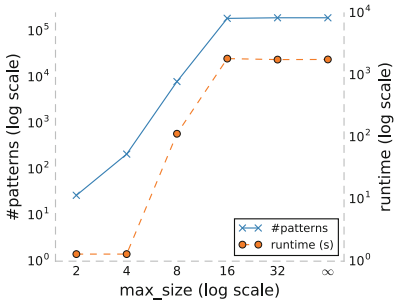
(b) $min\_sup = 4$, $max\_size = 5$



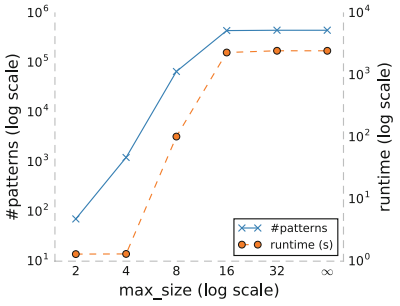
(c) $min\_coh = 0.01$, $max\_size = 4$



(d) $min\_coh = 0.01$, $max\_size = 5$



(e) $min\_sup = 350$, $min\_coh = 0.015$



(f) $min\_sup = 250$, $min\_coh = 0.01$

Fig. 1. Impact of various thresholds on output size and runtime. (a) Varying *min_coh* on *Species*.(b) Varying *min_coh* on *Moby*. (c) Varying *min_sup* on *Species*. (d) Varying *min_sup* on *Moby*. (e) Varying *max_size* on *Species*. (f) Varying *max_size* on *Moby*.

5 Related Work

We have examined the most important related work in Sect. 1, and experimentally compared our work with the existing methods in Sect. 4. Here, we place our work into the wider context of sequential pattern mining.

At the heart of most pattern mining algorithms is the need to reduce the exponential search space into a manageable subspace. When working with an anti-monotonic quality measure, such as frequency, the Apriori property can be deployed to generate candidate patterns only if some or all of their subpatterns have already proved frequent. This approach is used in both breadth-first-search (BFS) and depth-first-search (DFS) approaches, such as Apriori [1] and FP-growth [5] for itemset mining in transaction databases, GSP [2], SPADE [12] and PrefixSpan [9] for sequential pattern mining in sequence databases, or Winepi [8] and Marbles [4] for episode mining in event sequences.

For computational reasons, non-anti-monotonic quality measures are rarely used, or are used to re-rank the discovered patterns in a post-processing step. Recently, Tatti proposed a way to measure the significance of an episode by comparing the lengths of its occurrences to expected values of these lengths if the occurrences of the patterns' constituent items were scattered randomly [10]. However, the method uses the output of an existing frequency-based episode miner [11], and then simply assigns the new values to the discovered patterns. In this way, the rare patterns, such as those discussed in Sect. 4 will once again not be found. Our FCI_{SEQ} algorithm falls into the DFS category, but the proposed quality measure is not anti-monotonic, and we have had to rely on an alternative pruning technique to reduce the size of the search space. We believe the additional computational effort to be justified, as we manage to produce intuitive results, with the most interesting patterns, which existing methods sometimes fail to discover at all, ranked at the very top.

6 Conclusion

In this paper, we present a novel method for finding valuable patterns in event sequences. We evaluate the pattern quality using cohesion, a measure of how far apart the items making up the pattern are on average. In this way, we reward strong patterns that are not necessarily very frequent in the data, which allows us to discover patterns that existing frequency-based algorithms fail to find. Since cohesion is not an anti-monotonic measure, we rely on an alternative pruning technique, based on an upper bound of the cohesion of candidate patterns that have not been generated yet. We show both theoretically and empirically that the method is sound, the upper bound tight, and the algorithm efficient, allowing us to discover large numbers of patterns reasonably quickly. While the proposed approach concerns itemset mining, most of the presented work can be applied to mining other pattern types, such as sequential patterns or episodes.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: International Conference on Very Large Data Bases, pp. 487–499 (1994)
2. Agrawal, R., Srikant, R.: Mining sequential patterns. In: International Conference on Data Engineering, pp. 3–14 (1995)
3. Cule, B., Goethals, B., Robardet, C.: A new constraint for mining sets in sequences. In: SIAM International Conference on Data Mining, pp. 317–328 (2009)
4. Cule, B., Tatti, N., Goethals, B.: Marbles: mining association rules buried in long event sequences. Stat. Anal. Data Min. **7**(2), 93–110 (2014)
5. Han, J., Pei, J., Yin, Y., Mao, R.: Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Data Min. Knowl. Disc. **8**(1), 53–87 (2004)
6. Hendrickx, T., Cule, B., Goethals, B.: Mining cohesive itemsets in graphs. In: International Conference on Discovery Science, pp. 111–122 (2014)
7. Laxman, S., Sastry, P.S., Unnikrishnan, K.: A fast algorithm for finding frequent episodes in event streams. In: ACM SIGKDD Conference on Knowledge Discovery and Data Mining, pp. 410–419 (2007)
8. Mannila, H., Toivonen, H., Verkamo, A.I.: Discovery of frequent episodes in event sequences. Data Min. Knowl. Disc. **1**(3), 259–289 (1997)
9. Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., Hsu, M.C.: Mining sequential patterns by pattern-growth: the prefixspan approach. IEEE Trans. Knowl. Data Eng. **16**(11), 1424–1440 (2004)
10. Tatti, N.: Discovering episodes with compact minimal windows. Data Min. Knowl. Disc. **28**(4), 1046–1077 (2014)
11. Tatti, N., Cule, B.: Mining closed strict episodes. Data Min. Knowl. Disc. **25**(1), 34–66 (2012)
12. Zaki, M.J.: SPADE: an efficient algorithm for mining frequent sequences. Mach. Learn. **42**(1–2), 31–60 (2001)