Evaluation of Algorithms for Sequential Pattern Mining in Long Event Sequences

Josse Coen

A dissertation submitted in partial fulfillment of the requirements for the degree of Master in Computer Science: Data Science.

Principal supervisor: Prof. Dr. Bart Goethals
Assistant supervisor: Dr. Boris Cule

September 2018

Department of Mathematics and Computer Science
University of Antwerp

Contents

1. Introduction

In data mining, the goal is generally to extract useful information from datasets that are too large for a person to draw conclusions from without studying the data extensively, without any kind of summarization or other means of processing. Data mining encompasses the analysis of different kinds of data using a variety of methods.

One of the subdomains in data mining is frequent pattern mining. In frequent pattern mining, a large database of transactions — each transaction consisting of a (relatively small) set of items — is mined for frequent *itemsets*, that is, sets of items that often co-occur within transactions. A commonly cited use case are supermarket transactions, where each item in a transaction is a product that a customer bought during a visit to the store. In that case, frequent itemsets contain products that are frequently bought together.

A well-known algorithm for mining itemsets in such a transactional database is Apriori, introduced by Agrawal and Srikant [2], which relies on the fact that an itemset cannot be more frequent than any of its subsets, allowing the search space to be pruned heavily. It uses a breadth-first approach — first finding all 1-sized itemsets, then all 2-sized itemsets, and so on — generating larger candidates from smaller itemsets that are known to be frequent.

From itemsets that have been found frequent (or otherwise interesting), *association rules* [2] are used to find sets of items where the occurrence of one set can be a good predictor of another.

A first exploration into mining patterns in data of a sequential nature still presumed a database of transactions, with (relatively short) sequences as transactions instead of sets [1]. Since the data format is similar to that of typical frequent pattern mining, mining algorithms are often similar to those in frequent pattern mining as well.

Later, Mannila et al. introduced a new kind of setting for mining patterns in single, long *events sequences* [9], in which *events* occur at certain points in time.

Such sequences may represent different kinds of data. A few examples:

- Activity logs: whether or not a pattern of activity is cause for alarm [9].
- Machine logs, where the goal of mining data could be to predict failures beforehand and perform maintenance as needed.
- Any kind of text: books, articles, tweets, transcripts, ...
- Logs of user behaviour: How does a person use a graphical user interface? What

patterns do gamblers exhibit when playing video poker?

- Biosequences [8].

Though the main goal of finding patterns remains equivalent, the mining techniques used for these kinds of data stray further from those in the transactional setting. In sequential pattern mining, the main bottleneck is the length of the sequence, rather than the number of transactions.

Whereas patterns consist of itemsets in frequent pattern mining, in sequential pattern mining we speak of *episodes*. If one wants to assess the interestingness of an episode by how frequently it appears in the sequence, then the search space can be pruned in much the same way that Apriori does in frequent pattern mining.

In Chapter 2 we begin by formally defining the structure of the datasets we'll operate on: *event sequences*. Then we will move on to *patterns*: how do we define a pattern on an event sequence? And how do we quantify how interesting a pattern is in regard to an event sequence?

Then, in Chapter 3 we will present and implement algorithms which mine patterns according to the interestingness measures we defined.

In Chapter 4 we experiment to make an assessment of the implementation, in terms of:

1. the performance: how efficient is the implementation with a variety of datasets and parameters?
2. the quality of the output: can we find interesting patterns in different datasets? How does our implementation compare to other implementations which use other interestingness measures and classes of episodes?

To make such an evaluation, we will use a number of datasets of different kinds, fit them into event sequences, study the runtime of the algorithm across a range of parameters, and judge the quality of the output.

2. Problem statement

In this chapter we expound the fundamentals of what we'll be working on in later chapters. We begin by defining the structure of the datasets — *event sequences* composed of *events*, each event occurring at a certain point in time.

Next, we define patterns, named *episodes* in sequential pattern mining, and cover the classes of episodes we will be mining. We define several *frequency measures* proposed by different researchers, which quantify how interesting an episode is, with the goal of finding the most interesting episodes in a sequence.

Finally, from episodes we can construct *association rules*, which can uncover correlations between the different events in an event sequence. On association rules, *confidence measures* are defined, so that a high-scoring rule indicates a large correlation between events.

2.1. Event sequences

Definition 1. A *sequence event*, or *event* for short, is a pair (A, t) where $A \in \Sigma$ is an event type from a given set of event types Σ, and t is an integer timestamp, denoting the point in time at which the event occurred.

Definition 2. An *event sequence* $\boldsymbol{s}$ is a triple (s, T_s, T_e), where s is an ordered sequence of events

$$s = \langle (A_1, t_1), (A_2, t_2), \ldots, (A_n, t_n) \rangle$$

such that $t_i \leq t_{i+1}$ for all $i = 1, \ldots, n-1$, and any given pair (A, t) appears at most once.

With $\boldsymbol{s}_i$ we refer to the pair (A_i, t_i) in s.

T_s and T_e are timestamps such that $T_s \leq t_1$ and $t_n < T_e$. They mark the beginning and the end of the sequence, respectively.

If a sequence event (A, t) is in s, we say that event (A, t) *occurs* in $\boldsymbol{s}$ at timestamp t.

Figure 2.1 shows a visualization of a possible event sequence.

Note that multiple events can occur at the same timestamp, if they have different event types. Some implementations do not allow this, however.

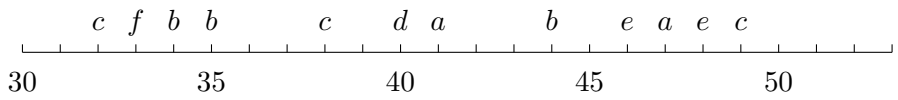


Figure 2.1.: A visual representation of a sequence.

For the sake of simplicity, we use the notation $s_1 \cdots s_n$ to mean the sequence $(\langle (s_1, 1), \ldots, (s_n, n) \rangle, 1, n+1)$.

Since sequences are allowed to be very long, and we are mostly looking for events occurring relatively close to each other, we would like to be able to consider parts of a sequence while ignoring the rest of it. Therefore we use the concept of a window on a sequence.

> **Definition 3.** Given a sequence $\boldsymbol{s} = (s, T_s, T_e)$ and two timestamps t_s and t_e, we define a *window* on $\boldsymbol{s}$ to be an event sequence $\boldsymbol{s}[t_s, t_e) = (w, t_s, t_e)$ where w contains all events (A, t) in s where $t_s \leq t < t_e$. We call $\rho = t_s - t_e$ the *width* of the window. $\boldsymbol{s}[t_s, t_e)$ is a *proper* subwindow of $\boldsymbol{s}$ if w contains fewer events than $\boldsymbol{s}$. Two windows $\boldsymbol{s}[a, b)$ and $\boldsymbol{s}[c, d)$ *overlap* if $[a, b) \cap [c, d) \neq \emptyset$. Otherwise we call them non-overlapping, or *disjoint*.

Note that in the previous definition, t_s and t_e do not have to be within the sequence itself. A window on a sequence can thus start before the sequence starts, or end after the sequence ends. This will be important for some of the algorithms.

Figure 2.2 illustrates a window on the sequence of Figure 2.1.

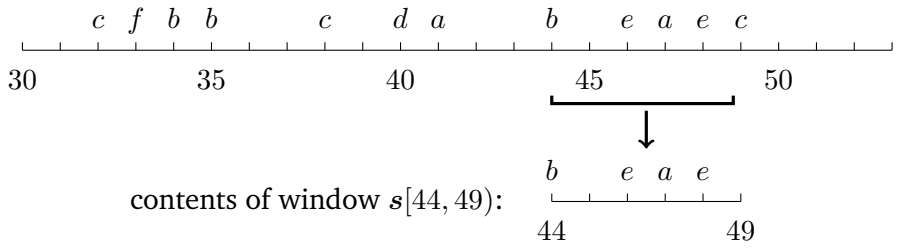


Figure 2.2.: A window of width 5 on a sequence $\boldsymbol{s}$.

2.2. Episodes

Given a pattern and a sequence, we would like to quantify how interesting the pattern is. One way to define *interesting* is how often it occurs in the sequence. While other interpretations of interestingness exist, we will mainly focus on *frequency measures*, some of which we'll discuss in section 2.3. First we formally define patterns.

Definition 4. An *episode* α is a directed acyclic graph with labelled nodes, that is, $\alpha = (V, E, lab)$, where $V = (v_1, \ldots, v_k)$ is the set of nodes, E is the set of directed edges, and *lab* is a function $lab\colon V \to \Sigma$, mapping each node v_i to an event type. If $lab(v) = A$, then we say that node v is of (event) type A, and that α contains A, or A is in α.

We write $|\alpha|$ to mean the number of nodes in an episode's graph. We call $|\alpha|$ the *size* of α.

Definition 5. A node n in an episode graph is a *descendant* of a node m if there is a path from m to n. Conversely m is an *ancestor* of n in that case.

Definition 6. Given a sequence $\boldsymbol{s}$ and an episode α we say that $\boldsymbol{s}$ *covers* α, or α *occurs in* $\boldsymbol{s}$, if there is an injective map f mapping each node v_i to a valid index such that:

1. the node v_i in α and the corresponding sequence element $\boldsymbol{s}_{f(v_i)}$ are of the same event type: $\boldsymbol{s}_{f(v_i)} = lab(v_i)$, and
2. if there is an edge (v_i, v_j) in α, then we must have $f(v_i) < f(v_j)$. In other words, the ancestors of v_j must occur in $\boldsymbol{s}$ before v_j. If the mapping f is surjective, that is, all events in $\boldsymbol{s}$ are used, we will say that $\boldsymbol{s}$ is an *instance* of α.

More intuitively, the above definition says that an episode occurs in a sequence if the event types corresponding to the episode's nodes appear in the sequence, respecting the order of the episode's edges.

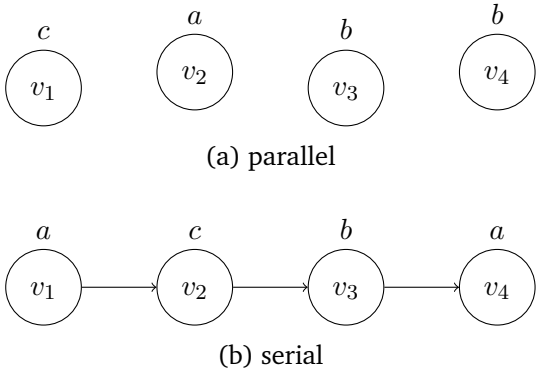
Definition 7. Episodes α and β are said to be *equivalent* if each sequence that covers α also covers β, and vice versa.

In this thesis, we'll be limiting ourselves to two subcategories of episodes:

- **Parallel episodes.** A parallel episode is an episode for which the set of edges is empty. As such, no constraints are placed on the order in which event types occur in a sequence. An example is shown in Figure 2.3a. In text we'll write parallel episodes by their event types using the following notation:

  $$\{A_1, A_2, \ldots, A_n\}$$

  by which we mean a parallel episode with n nodes, and event types A_1 through A_n. Note that though the notation reminds strongly of the notation for a mathematical set, it does not actually represent a set: a parallel episode $\{a, a, b, c\}$ is not equivalent to an episode $\{a, b, c\}$.
- **Serial episodes.** A serial episode is an episode for which the edges cause the nodes to have a total order. That way, in any occurrence of a serial episode in a sequence, the event types appear in the same order. Figure 2.3b shows an example.



(a) parallel

(b) serial

Figure 2.3.: An example of an episode graph for each of the episode class we will consider, with the event types shown above the nodes.

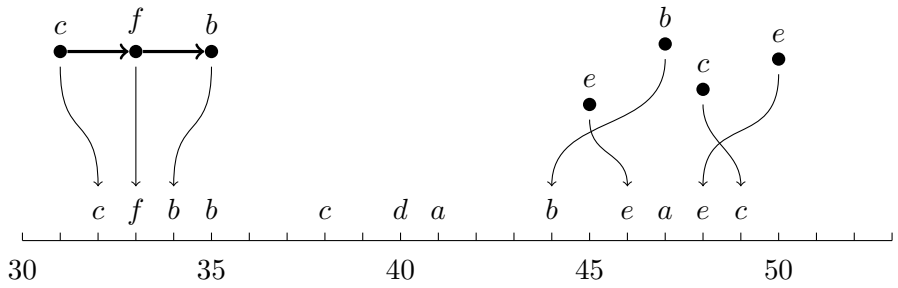


Figure 2.4.: Showing an occurrence of a serial episode $c \to f \to b$ and an occurrence of a parallel episode $\{b, c, e, e\}$ in the sequence of Figure 2.1.

For serial episodes we will use the notation

$$A_1 \to A_2 \to \cdots \to A_n$$

where A_i is the event type of the i-th node, and a node of event type A_i is an ancestor of a node of event type A_j if $i < j$.

Note that episodes which consist of only one node are both parallel and serial.

Figure 2.4 shows an example of two occurrences of episodes in a sequence — on the left an occurrence of serial episode $c \to f \to b$, and on the right an occurrence of a parallel episode $\{b, c, e, e\}$.

Because we only consider parallel and serial episodes, it is easy to represent an episode in a data structure — we don't need to explicitly store a graph with nodes and edges:

- Parallel episodes can be stored in an array, where each element is simply the event type of the episode. Strictly speaking, the order of the elements in the array doesn't matter, but it can be useful for them to follow some order on the set of event types.

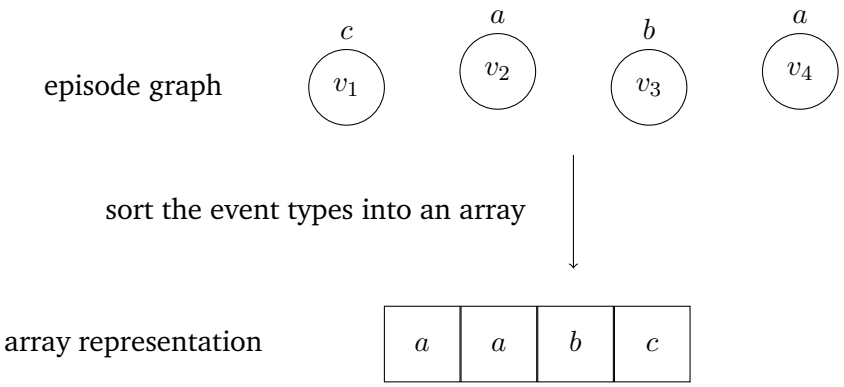


Figure 2.5.: A parallel episode's graph representation, with the label of each node shown above, and its array representation.

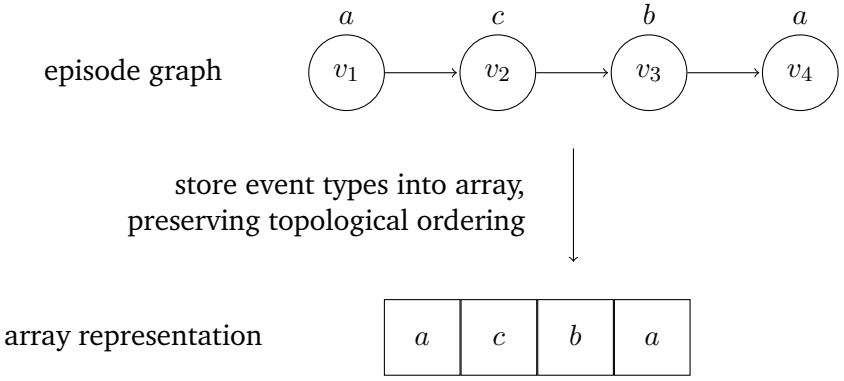


Figure 2.6.: A serial episode's graph representation, with the label of each node shown above, and its array representation.

In that way each parallel episode has a unique array representation. Figure 2.5 shows the representation visually.

- Serial episodes can also be stored in an array, but here the order of the elements is defined by the edges of the episode. That is, the event types are ordered according to a topological sort of the nodes. Figure 2.6 shows this visually.

When discussing the algorithms we will mostly consider their array representation, and address the i-th element of an episode array α with $\alpha[i]$.

Definition 8. Given two episodes α and β, we say that α is a *subepisode* of β, denoted $\alpha \subseteq \beta$, if the set of all sequences that cover β is a subset of the set of all sequences that cover α. α is a *proper subepisode* of β, denoted $\alpha \subset \beta$, if $\alpha \subseteq \beta$ and α and β are not equivalent.

Of course, considering all event sequences that cover an episode is an impossible task, so the above definition does not give rise to a practical way of determining whether one

episode is a subepisode of another. Instead we must look at the episodes themselves, and reason about all of the ways in which they might occur in a sequence. For parallel and serial episodes, it is quite straightforward to do so, as described below. More general cases have been reasoned about by Tatti and Cule [10], but are beyond the scope of this thesis.

- A parallel episode α is a subepisode of a parallel episode β if:
  1. $|V(\alpha)| \leq |V(\beta)|$, and
  2. for each event type A in α, the number of nodes v in α for which $lab(v) = A$, is no greater than the number of nodes for which the same holds in β.
- A serial episode α is a subepisode of a serial episode β if:
  1. the conditions for parallel episodes are satisfied, and additionally
  2. if node v is an ancestor of node u in α, then there exist nodes u', v' in β such that v' is an ancestor of u' in β and $lab(u) = lab(u')$ and $lab(v) = lab(v')$. Put more simply, the ordering of event types must be preserved.

2.3. Frequency measures for episodes

Now that we have defined patterns on event sequences, we would like to be to be able to quantify how interesting a pattern is in regard to a sequence.

Traditionally, the most common interpretation of interestingness is frequency, which intends to express how often an episode occurs in the sequence. The more often an episode occurs, the more interesting it is considered to be.

Exactly how the frequency of an episode in a sequence should be defined, is not immediately clear. We will present three methods to measure the frequency of an episode. In a later chapter, we implement a mining algorithm based on them.

In data mining, the main challenge is to find ways of mining information efficiently. In particular, exponential pattern growth is problematic. For itemset mining, the Apriori algorithm [2] was able to prune large amounts of itemsets because an itemset can only be frequent if all of its subsets are frequent. In sequential pattern mining, the search space is even larger, because an event type can occur any number of times in an episode. However, we can employ a similar technique if we define *monotonically decreasing* frequency measures on episodes.

> **Definition 9.** A frequency measure is *monotonically decreasing* if each episode is at least as frequent as any of its superepisodes.

Given such a frequency measure, if we find an episode to be infrequent, then we know that all of its superepisodes will be infrequent as well.

2.3.1. Fixed-window frequency

The first frequency measure we discuss was originally proposed by Mannila et al. in [9], and is based on windows of a fixed length.

> **Definition 10.** Given a window width ρ and a sequence $\boldsymbol{s}$, we define the *fixed-window frequency* of an episode α in $\boldsymbol{s}$, denoted $fr_f(\alpha, \boldsymbol{s})$, to be the number of windows of width ρ in $\boldsymbol{s}$ covering the episode:
>
> $$fr_f(\alpha, \boldsymbol{s}) = |\{\boldsymbol{s}[t, t+\rho) \mid \boldsymbol{s}[t, t+\rho) \text{ covers } \alpha\}|$$

Using this definition, we can find the probability of finding an occurrence of the episode in a randomly chosen window of width ρ:

$$\frac{fr_f(\alpha, \boldsymbol{s})}{|\{\boldsymbol{s}[t, t+\rho) \mid t \in [T_s - \rho + 1, T_e]\}|}$$

For a visual example of the fixed-window frequency of an episode, see Figure 2.7. It shows all of the windows of width 8 that cover the episodes shown in Figure 2.4. We observe that $fr_f(\{b, c, e, e\}) = 6$, and $fr_f(c \to f \to b) = 3$.

In event sequence mining, it is desirable for a frequency measure to give more weight to an occurrence that spans a shorter time range. The farther apart the events composing an occurrence, the more likely it is that the occurrence is accidental. Figure 2.7 shows that the events for the serial episode are close together, such that a larger number of windows cover it, giving it a higher score than the parallel episode, where the events giving rise to the occurrence are more spread out, such that fewer windows cover the episode. So, the fixed-window frequency does have this appealing trait.

Upon closer inspection, the frequency values themselves are often counterintuitive, as they don't reflect very well how many occurrences there really are. For instance, the fixed-window frequency is highly dependent on the chosen window width — even the contribtion of a single occurrence towards the whole frequency. If an episode has n occurrences sufficiently far apart (such that no two occurrences appear in one window) increasing ρ by 1 causes the fixed-window frequency to increase by n, while the total number of windows covering the sequence increases by only 1. This amplifies the frequency of already-frequent episodes, compared to less frequent ones. And it makes for frequency values that aren't easily interpretable.

Another consideration is that a lot of information about the contents of the windows is lost. Take the toy example of Figure 2.8. With a window size of 9 or more, the fixed-window frequency of $\{a, b\}$ in Figure 2.8b and Figure 2.8a is equal, even though one would naturally say that the episode occurs at least twice as often in Figure 2.8b. Therefore ρ should be carefully chosen.

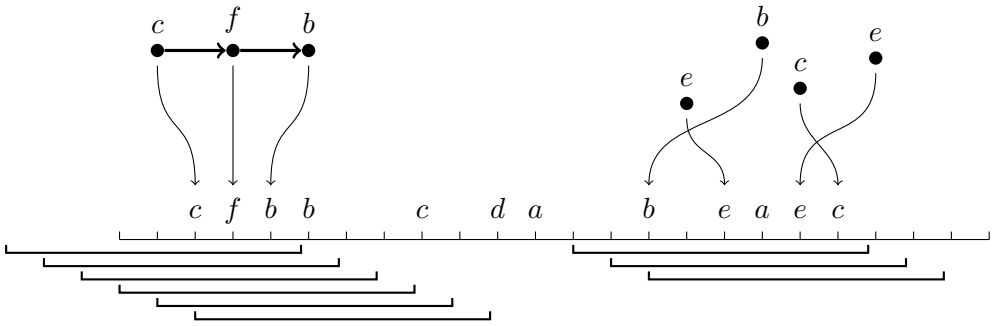
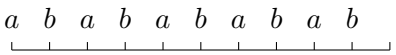


Figure 2.7.: All of the windows of width 8 that cover two episodes in a sequence; a parallel episode $\{b, c, e, e\}$ and a serial episode $c \to f \to b$.

(a) Few events.



(b) Many events.

Figure 2.8.: Two sequences. With $\rho \geq 9$, the fixed-window frequency of $\{a, b\}$ is equal in both sequences.

2.3.2. Disjoint-window frequency

Another approach towards defining the frequency of an episode in a sequence is to use minimal windows. The idea was already put forward by Mannila et al. in [9], and was expanded upon by Laxman et al. in [7] and [6].

> **Definition 11.** Given a sequence s, an episode α, and a positive integer ρ, a window $\boldsymbol{s}[t_s, t_e)$ is called a *minimal window* of α in $\boldsymbol{s}$, if:
>
> 1. $t_e - t_s \leq \rho$,
> 2. $\boldsymbol{s}[t_s, t_e)$ covers α, and
> 3. no proper subwindow of $\boldsymbol{s}[t_s, t_e)$ covers α.
>
> We denote the set of all minimal windows of α in $\boldsymbol{s}$ with $mw(\alpha, \boldsymbol{s})$, or simply $mw(\alpha)$ if $\boldsymbol{s}$ is known from the context. Given a set of minimal windows W, we define $dis(W)$ to be true if all windows in W are pairwise disjoint, and false otherwise.

Requirement 1 puts a limit on the width of a minimal window, under the assumption that windows of greater width aren't useful. This limit is also convenient for the mining algorithms.

If we mention a *minimal occurrence* of an episode, we refer to the occurrence that gives

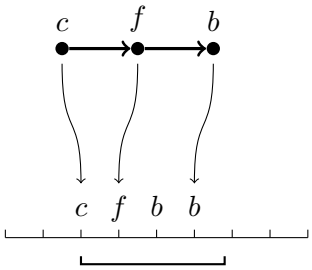


Figure 2.9.: A window for episode $c \to f \to b$ that minimally covers a certain occurrence, yet is not a minimal window.

rise to a minimal window of the episode.

To further illustrate the requirements for a window being minimal, consider the following example. For serial episode $c \to f \to b$, observe the occurrence in Figure 2.9: instead of picking the first b, now we chose the second b. While this is a valid occurrence for the episode, the smallest window covering this particular occurrence contains an even smaller window that still covers the episode — with a different occurrence — and so the larger window is not a minimal window of the episode.

Minimal windows of an episode can still overlap, however. As a very simple example consider the sequence $a\ b\ a$ and episode $\{a, b\}$. Then both $[1, 3)$ and $[2, 4)$ are minimal windows of $\{a, b\}$. They cannot overlap arbitrarily, though: given two minimal windows for an episode $[a, b)$ and $[c, d)$, (without loss of generality) it holds that $a < c \Leftrightarrow b < d$. If this wasn't the case, then one could be contained in the other, contradicting the initial assumption that both windows are minimal.

Simply defining $fr_m(\alpha) = |mw(\alpha)|$ is not a viable option for efficiently mining episodes, because it doesn't satisfy the monotonicity property. As an example, take the sequence $a\ b\ a\ c\ b\ c$, and serial episode $a \to b \to c$. The minimal windows of the episode in the sequence are $mw(a \to b \to c) = \{[1, 5), [3, 7)\}$, while subepisode $a \to c$ has only one minimal window: $mw(a \to c) = \{[3, 5)\}$. So for this sequence, $a \to b \to c$ would have a greater frequency than one of its subepisodes $a \to c$. As stated before, this is computationally undesirable.

As proven in the previous example, an episode may have more minimal windows than one of its subepisodes. This is due to the fact that minimal windows can overlap — the minimal window of $a \to c$ is contained within two minimal windows of $a \to b \to c$. If we forced ourselves to drop either of the two minimal windows of $a \to b \to c$, the monotonicity property would be satisfied for the example. This leads us to the disjoint-window frequency [7].

Definition 12. The *disjoint-window frequency* of an episode α in a sequence $\boldsymbol{s}$, denoted $fr_m(\alpha)$, is defined as the maximal number of non-overlapping minimal

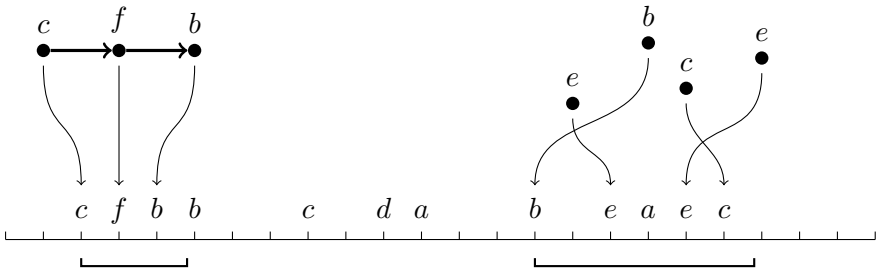


Figure 2.10.: All of the minimal windows that cover $\{b, c, e, e\}$ and $c \rightarrow f \rightarrow b$ in the sequence, for $6 \leq \rho \leq 10$.

windows within $\boldsymbol{s}$ that contain episode α. Formally:

$$fr_m(\alpha) = \max\{|W| \mid W \subseteq mw(\alpha) \wedge dis(W)\}$$

The disjoint-window frequency is monotonically decreasing, allowing for Apriori-style pruning. By the non-overlapping requirement, it effectively disallows events from contributing multiple times towards the frequency of an episode.

Compared to the fixed-window frequency, the disjoint-window frequency expresses the frequency of an episode in a sequence more naturally. Consider the example in Figure 2.8, where the fixed-window frequency didn't distinguish between the two cases at all, above a certain window width. For any $\rho \geq 2$, the disjoint-window frequency of $\{a, b\}$ is 2 for the sequence in Figure 2.8a, and 5 for the sequence in Figure 2.8b. So the disjoint-window frequency does not suffer the information loss problem for larger ρ described in section 2.3.1.

Moreover, with the disjoint-window frequency, once the window width ρ employed by an algorithm is large enough to discover a certain minimal minimal window $[a, b)$, the contribution of that occurrence to the disjoint-window frequency is constant for all $\rho \geq b - a$. This wasn't the case for the fixed-window frequency, as illustrated in section 2.3.1.

The disjoint-window frequency does have a disadvantage compared to the fixed-window frequency as well. The width of a minimal window is unaccounted for. So a small minimal window contributes just as much to the disjoint-window frequency as a large one. As we explained in section 2.3.1, occurrences spanning a short time range are more likely to be significant than ones spanning a long range. Figure 2.10 shows the minimal windows for the same example as Figure 2.7. Where the fixed-window frequency assigned a greater frequency to the shorter occurrence, the disjoint-window frequency treats them both equally. Thus here, again, ρ should be carefully chosen, as to not include minimal windows that are too large to be meaningful.

2.3.3. Weighted-window frequency

In the preceding sections we studied some of the characteristics of the fixed-window frequency and the disjoint-window frequency. We liked that the fixed-window frequency rates closely-packed occurrences higher than ones which are spread out a lot; but we also liked the way the disjoint-window frequency considers specific occurrences more attentively. The weighted-window frequency, introduced by Cule et al. in [4], is based on minimal windows, but considers the value of a minimal window to be inversely proportional to its width.

Definition 13. We define the *weight* of a minimal window $[a, b)$ to be the inverse of its width:

$$\frac{1}{b-a}$$

Given a set of windows W, the *total weight* of W is the sum of all the weights:

$$\sum_{[a,b)\in W} \frac{1}{b-a}$$

The *weighted-window frequency* of an episode α in a sequence $\boldsymbol{s}$, denoted $fr_w(\alpha)$, is defined as

$$fr_w(\alpha) = \max \left\{ \sum_{[a,b)\in W} \frac{1}{b-a} \,\middle|\, W \subseteq mw(\alpha) \wedge dis(W) \right\}$$

In words, the weighted-window frequency is the sum of weights of a set of disjoint minimal windows that maximizes the total weight.

The frequency values that the weighted-window frequency produces aren't as easily interpretable as those for the disjoint-window frequency, where the frequency simply expresses the number of minimal occurrences. But the fact that not all minimal windows are equally valuable, can be very useful. For instance, with both of the previous frequency measures, the window width had to be carefully chosen in order to have meaningful results. With the weighted-window frequency, on the other hand, the window width can be as large as computationally feasible, because any minimal windows greater than ρ will contribute at most $1/(\rho + 1)$ to the frequency of an episode — which is to say, not very much for large ρ.

A drawback of the disjoint-window frequency is that it will disadvantage large episodes more so than the disjoint-window frequency, because generally, larger episodes naturally require larger — less-weighted — windows. So, compared to the weighted-window frequency, the disjoint-window frequency is more likely to find false positives, but less likely to miss true positives.

2.4. Association rules

We've quantified the interestingness of patterns in different ways. Besides expressing the interestingness of a single episode, it might also be desirable to know if an occurrence of an episode gives rise to the occurrence of certain other events nearby. Association rules allow us to find such correlations, similarly to association rules in itemset mining.

> **Definition 14.** Given two episodes α and β such that $\alpha \subset \beta$, we can express an *association rule* $\alpha \Rightarrow \beta$. We call α the *head* of the rule, and β the *tail* of the rule.

An association rule $\alpha \Rightarrow \beta$ can be read as *an occurrence of α leads to an occurrence of β*. Or, *if we find an occurrence of α, then we are likely to find an occurrence of β nearby*.

If we state that $\alpha \Rightarrow \beta$ holds in the context of a sequence, we must be confident stating it. Thus, we want to be able to express *how* confident we are about that statement, and define *confidence measures* to quantify this confidence. We will introduce a number of measures in section 2.5.

The purpose of the search for confident association rules is to find episodes α which are good predictors of superepisodes β.

Typical algorithms (including ours) only consider association rules consisting of episodes that are frequent in the sequence — episodes that were mined from the sequence. It should be noted, though, that association rules consisting of infrequent episodes can be confident as well. But those are much harder to find. The main reason for only considering frequent episodes is to limit the number of candidates to make mining algorithms tractable. Therefore we work under the assumption that association rules are only interesting if they consist of interesting episodes.

2.5. Confidence measures for association rules

Given an association rule $\alpha \Rightarrow \beta$, we can define a confidence value $c(\alpha \Rightarrow \beta)$, which expresses the likelihood of finding an occurrence of β, given an occurrence of α in a sequence. So, intuitively, if we've found an occurrence of α in the sequence, $c(\alpha \Rightarrow \beta)$ rule should express the probability of finding an occurrence of β "surrounding" — and containing — the occurrence of α.

We will present three methods to measure the confidence of an association rule, each based on its equivalent frequency measure defined in section 2.3.

2.5.1. Fixed-window confidence

A first method is to use the fixed-window frequency of episodes [9].

Definition 15. Given a window width ρ and episodes α and β, such that $\alpha \subset \beta$, we define the *fixed-window confidence* of the association rule $\alpha \Rightarrow \beta$, denoted $c_f(\alpha \Rightarrow \beta)$, to be the ratio of their respective frequencies:

$$c_f(\alpha \Rightarrow \beta) = \frac{fr_f(\beta)}{fr_f(\alpha)}$$

Intuitively, this measure of confidence expresses the amount of windows that contain β, relative to the windows that contain α. In probabilistic terms, it answers the following: *if we pick a random window in which α occurs, how likely are we to find β as well?*

In section 2.3.1 we established that the frequency values produced by the fixed-window frequency heavily depend on the chosen window width. So again, in order for the results to be meaningful, the window width should not be chosen too large.

2.5.2. Minimal-window confidence

The confidence of an association rule can also be based on minimal windows [4]. As opposed to the fixed-window confidence, here we don't directly base the confidence measure upon the corresponding frequency measure. To obtain a confidence value of an association rule, we consider the minimal windows of both episodes involved.

Contrary to the disjoint-window frequency, for association rules we *do* use overlapping minimal windows, since the use of non-overlapping minimal windows for the disjoint-window frequency is just a means to define an anti-monotonic frequency measure. For the minimal-window confidence, there is no reason not to consider all minimal occurrences.

A naive approach to a confidence measure based on minimal windows, could be to define the confidence of $\alpha \Rightarrow \beta$ as

$$\frac{|mw(\beta)|}{|mw(\alpha)|}$$

similarly to the fixed-window confidence. However, we're trying to express the probability of finding β, given that we've found α. If a minimal window of β contains multiple minimal windows of α, the above expression does not represent the intended meaning.

Take for example the rule $\{a\} \Rightarrow \{a, a\}$. Each minimal window of the tail contains at least two minimal windows of the head. Using the naive approach, this rule would have a confidence of at most $1/2$.

Instead, for each minimal window of α it is determined whether or not it is contained within a minimal window of β.

Definition 16. Given episodes α and β, such that $\alpha \subset \beta$, and a minimal window $\boldsymbol{s}[a, b)$ of episode α. If there exists a minimal window $\boldsymbol{s}[c, d)$ of β such that

$[a, b) \subset [c, d)$, we define the *minimal-extensibility* of occurrence $\boldsymbol{s}[a, b)$ of α into an occurrence of β

$$ext_m(\boldsymbol{s}[a, b), \alpha, \beta) = 1$$

If such a window doesn't exist, we define $ext_m(\boldsymbol{s}[a, b)) = 0$.

Now the minimal-window confidence is defined by the percentage of minimal windows of α that are contained within a minimal window of β.

Definition 17. Given episodes α and β such that $\alpha \subset \beta$, we define the *minimal-window confidence* of the association rule $\alpha \Rightarrow \beta$, denoted $c_m(\alpha \Rightarrow \beta)$, to be:

$$c_m(\alpha \Rightarrow \beta) = \frac{\sum_{w \in mw(\alpha)} ext_m(w, \alpha, \beta)}{|mw(\alpha)|}$$

Coming back to the example: using this definition, $\{a\} \Rightarrow \{a, a\}$ can obtain confidence values of up to 1, in case a never appears alone — more precisely, if for each event of type a there is another a within a timespan of ρ.

2.5.3. Weighted-window confidence

In order to calculate the confidence of association rules based on weighted windows [4], we first define the weighted-extensibility.

Definition 18. Given episodes α and β, such that $\alpha \subset \beta$, and a minimal window $\boldsymbol{s}[a, b)$ of episode α. Assume that there exists a minimal window $\boldsymbol{s}[c, d)$ such that $[a, b) \subset [c, d)$, and that this is the smallest window that contains $s[a, b)$, then we define the *weighted-extensibility* of occurrence $\boldsymbol{s}[a, b)$ of α into an occurrence of β as

$$ext_w(\boldsymbol{s}[a, b), \alpha, \beta) = \frac{b - a}{d - c}$$

If there exists no such minimal window of β, we define $ext_w(\boldsymbol{s}[a, b), \alpha, \beta) = 0$.

The inverse of the weighted-extensibility tells us how much we need to enlarge a minimal occurrence of α to find a minimal occurrence of β.

Using the weighted-extensibility, the weighted-window confidence is defined very similarly to the minimal-window confidence.

Definition 19. Given episodes α and β, such that $\alpha \subset \beta$, we define the *weighted-window confidence* of the association rule $\alpha \Rightarrow \beta$, denoted $c_w(\alpha \Rightarrow \beta)$, to be the average weighted-extensibility of an occurrence of α into an occurrence of β:

$$c_w(\alpha \Rightarrow \beta) = \frac{\sum_{w \in mw(\alpha)} ext_w(w, \alpha, \beta)}{|mw(\alpha)|}$$

As with the respective frequency measures, the weighted-window confidence is more conservative in rating the confidence of an association rule $\alpha \Rightarrow \beta$ than the minimal-window confidence. The minimal-window confidence values all minimal windows of α contained within a minimal window of β equally. The weighted-window nuances in the sense that if a minimal window of α needs to be extended a lot before finding a minimal occurrence of β, then the small occurrence of α by itself is unlikely to have played a big role in the existence of the large occurrence of β.

3. Algorithms

Now that we have defined episodes and association rules on sequences, and their respective measures of frequency and confidence, we are ready to present algorithms that find all frequent episodes and confident association rules. We won't repeat the definitions, but it is useful to keep them in mind, in order to see exactly how the algorithms accomplish their task.

We will start with the algorithms that find episodes, and then move on to association rules. For each, we'll give a sketch of the full procedure outlining the high-level steps, before drilling down into the details.

3.1. Finding frequent episodes: high-level algorithm

Algorithm 1 High-level algorithm for finding frequent episodes.
Input: A set Σ of event types, an episode class $\mathcal{E}$ (parallel or serial), a frequency measure Ψ an event sequence $\boldsymbol{s}$ over Σ, a window width ρ, and a frequency threshold *min_fr*.
Output: The collection of episodes that are frequent in the sequence in terms of the input parameters.

1: $\mathcal{C}_1 \leftarrow \{\langle A \rangle \mid A \in \Sigma\}$
2: $l \leftarrow 1$
3: **while** $\mathcal{C}_l \neq \emptyset$ **do**
4: $\quad$ /* Database pass (different subalgorithms depending on $\mathcal{E}$) */
5: $\quad$ compute $\mathcal{F}_l \leftarrow \{\alpha \in \mathcal{C}_l \mid fr_\Psi(\alpha, \boldsymbol{s}, \rho) \geq \text{min\_fr}\}$
6: $\quad$ $l \leftarrow l + 1$
7: $\quad$ /* Candidate generation (algorithm 2) */
8: $\quad$ compute $\mathcal{C}_l = \{\alpha \mid |\alpha| = l \wedge \forall \beta \mid \beta \subset \alpha : fr_\Psi(\beta, \boldsymbol{s}, \rho) \geq \text{min\_fr}\}$
9: **for all** $i \mid |\mathcal{F}_i| > 0$ **do**
10: $\quad$ output $\mathcal{F}_i$

Algorithm 1 (adapted from [9]) describes the high-level procedure to find all frequent episodes of a given class of episodes — parallel or serial — that are frequent in an event sequence $\boldsymbol{s}$, given window width ρ and frequency threshold *min_fr*. It is a breadth-first algorithm, that is, all frequent episodes of size l are computed before those of size $l + 1$. The algorithm finds either parallel or serial episodes, not both at the same time, since some of the subalgorithms are tailored specifically to a class of episodes.

The construction of the first set of candidates is a special case. $\mathcal{C}_1$ consists of simply all possible episodes with one node. That's one episode per event type, and so $|C_1| = |\Sigma|$. After constructing $\mathcal{C}_1$ a loop is entered where ever-bigger episodes are constructed and then filtered by frequency in the sequence.

The following subsections will cover all of the subalgorithms. Candidate generation is identical for all frequency measures — with only a slight variation between parallel and serial episodes (Algorithm 2, Section 3.2). Determining which of the candidates are frequent, is more involved and highly depends on both the episode class and the frequency measure (Algorithm 3 in Section 3.3, Algorithm 4 in Section 3.4, Algorithm 5 in Section 3.5, Algorithm 6 in Section 3.6, Algorithm 7 in Section 3.7, Algorithm 8 in Section 3.8, Algorithm 9 in Section 3.9).

3.2. Candidate generation

Algorithm 2 Generating candidate parallel episodes of size $l + 1$ from frequent parallel episodes of size l.
Input: A sorted array $\mathcal{F}_l$ of frequent parallel episodes of size l.
Output: A sorted array of candidate parallel episodes of size $l + 1$.

1: $\mathcal{C}_{l+1} \leftarrow$ empty array
2: $k \leftarrow 0$
3: **if** $l = 1$ **then**
4: **for** $h \leftarrow 1$ to $|\mathcal{F}_l|$ **do** $\mathcal{F}_l$.block_start$[h] \leftarrow 1$
5: **for** $i \leftarrow 1$ to $|\mathcal{F}_l|$ **do**
6: current_block_start $\leftarrow k + 1$
7: **for** $j \leftarrow i$; $\mathcal{F}_l$.block_start$[j] = \mathcal{F}_l$.block_start$[i]$; $j \leftarrow j + 1$ **do**
8: $/*$ $\mathcal{F}_l[i]$ and $\mathcal{F}_l[j]$ have $l - 1$ first event types in common, build a potential candidate α as their combination. $*/$
9: **for** $x \leftarrow 1$ to l **do** $\alpha[x] \leftarrow \mathcal{F}_l[i][x]$
10: $\alpha[l + 1] \leftarrow \mathcal{F}_l[j][l]$
11: $/*$ Build and test subepisodes β that do not contain $\alpha[y]$ $*/$
12: **for** $y \leftarrow 1$ to $l - 1$ **do**
13: **for** $x \leftarrow 1$ to $y - 1$ **do** $\beta[x] \leftarrow \alpha[y]$
14: **for** $x \leftarrow y$ to l **do** $\beta[x] \leftarrow \alpha[x + 1]$
15: **if** β is not in $\mathcal{F}_l$ **then** continue with the next j at line 7
16: $/*$ All subepisodes are in $\mathcal{F}_l$, store α as candidate $*/$
17: $k \leftarrow k + 1$
18: $\mathcal{C}_{l+1}[k] \leftarrow \alpha$
19: $\mathcal{C}_{l+1}$.block_start$[k] \leftarrow$ current_block_start
20: output $\mathcal{C}_{l+1}$

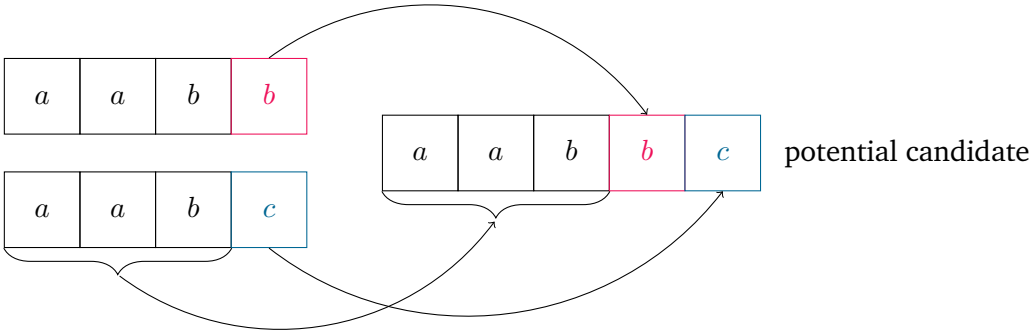


Figure 3.1.: Two episodes with a shared prefix being combined into a larger episode.

Algorithm 2 (from [9]) generates candidates of size $l + 1$ from a collection of frequent parallel episodes of size l. It can be easily modified to generate serial episodes, as we'll show later. Thanks to the monotonic property of the frequency measures we implement, some candidates can be immediately proven infrequent. More specifically, if there exists an infrequent subepisode β of candidate α, then α is not frequent. Conveniently, all frequent subepisodes of size l are already given as input to the algorithm. So when the algorithm constructs a candidate of size $l+1$, all of its subepisodes of size l are checked to be frequent by testing whether they are in the input collection $\mathcal{F}_l$. If one of the subepisodes is not in $\mathcal{F}_l$, then it is infrequent and, consequently, the potential candidate cannot be frequent either. Subepisodes smaller than l do not need to be checked anymore, because any less-than-l-sized subepisode of the potential candidate is also a subepisode of one of the l-sized subepisodes and any l-sized subepisodes that have turned out to be frequent are already known to have frequent subepisodes.

Parallel episodes are constructed such that the elements in their arrays are sorted according to event type. In this way, a parallel episode has a unique array representation.

Potential candidates of size $l + 1$ are generated by combining frequent episodes of size l which share a prefix of $l - 1$ elements. In other words, they are only allowed to differ in their last element. The newly constructed candidate shares these $l - 1$ as well, and is then followed by the last elements of both α and β. This method of candidate generation is similar to the way in which candidate itemsets are generated in the Apriori algorithm [9]. Lines 9 and 10 implement this. Figure 3.1 shows an example of two episodes being combined, and Figure 3.2 shows visually how larger candidates build upon smaller episodes.

The algorithm assumes that the input array of frequent l-sized episodes $\mathcal{F}_l$ is sorted lexicographically. That is, the episodes are firstly sorted by the first event type in their array representation; those episodes which share their first element are sorted by the second element, and so on. Therefore episodes which share a prefix are grouped together. The algorithm constructs new episodes in such a way that the output is also ordered lexico-

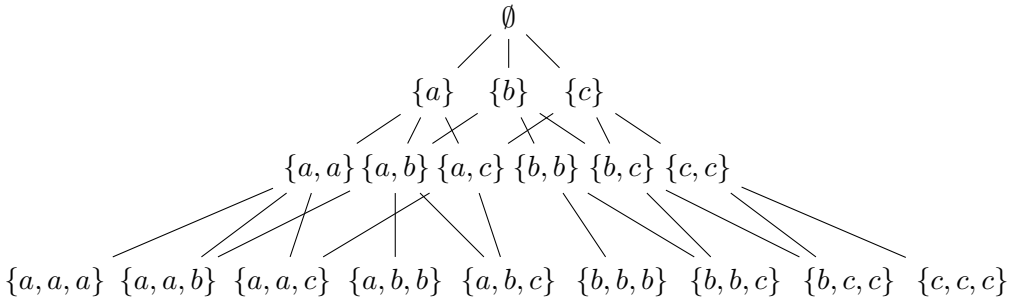


Figure 3.2.: Parallel episode construction for $\Sigma = \{a, b, c\}$ up to size 3.

graphically, and since part of the output of one run becomes the input of the next, the assumption about the input is always true.

As mentioned previously, a potential candidate is generated from two episodes that share an $(l-1)$-prefix in their array representation. And with the episodes being ordered as described, all of the episodes that share a prefix are grouped together. As a result, episodes can be grouped into *blocks*, where all of the episodes in a block share the first $l-1$ elements. All episodes within the same block are combined. Figure 3.3 illustrates the block structure.

The block structure is represented as follows. Each episode α is associated with a value *block_start*, which is the array index of the first episode in the block which contains α, so it points "back" to the beginning of the block. The first episode in each block points to itself. In this way, it can be easily tested whether two episodes belong to the same block. Figure 3.4 shows this representation for the episodes in Figure 3.3.

The algorithm constructs this block structure while generating candidates of size $l+1$, but it also uses the blocks of the l-sized episodes given as input. So it is important that they are preserved and maintained in between different runs of Algorithm 2. Section 3.9 goes into more detail about this.

We mentioned earlier that in the array representation of parallel episodes, elements are sorted according to event type. We would like to construct parallel candidates for which this is the case as well. We know that the input array of frequent episodes is sorted, and that all episodes in a block share an $(l-1)$-prefix. Consequently, for any two episodes $\mathcal{C}[i]$ and $\mathcal{F}[j]$ in a block such that $i \leq j$, it holds that $\mathcal{F}[i][l] \leq \mathcal{F}[j][l]$. Now, if we always construct a new candidate α as $\langle \mathcal{F}[i][1], \ldots, \mathcal{F}[i][l], \mathcal{F}[j][l] \rangle$ and choose i and j such that $i \leq j$, then α's array representation is sorted by construction. This explains the nested-loop structure of the algorithm (lines 5 and 7), where the inner array index j is always greater than or equal to the outer index i.

With serial episodes on the other hand, the order of elements in the array is not bound by an order on the event types. Serial candidate generation can be accomplished by a small change to the algorithm. Line 7 needs to be changed to:

for $j \leftarrow \mathcal{F}_l$.block_start$[i]$; $\mathcal{F}_l$.block_start$[j] = \mathcal{F}_l$.block_start$[i]$; $j \leftarrow j+1$ **do**

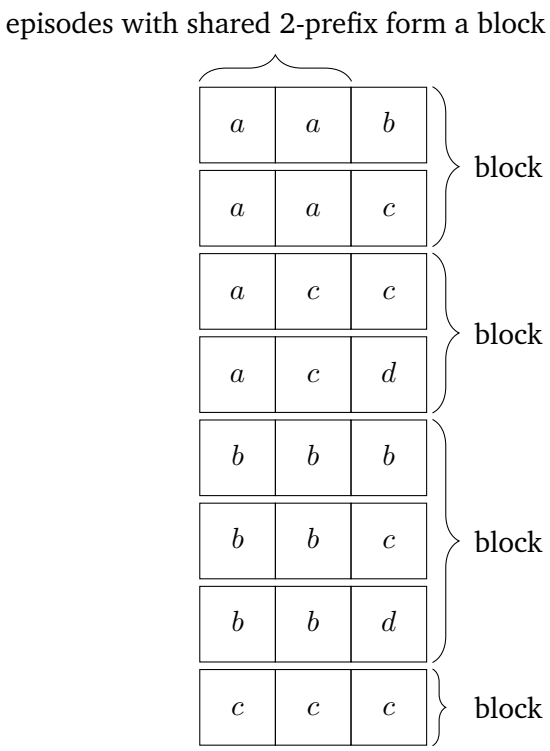


Figure 3.3.: Blocks in candidate generation algorithm with parallel episodes of size 3.

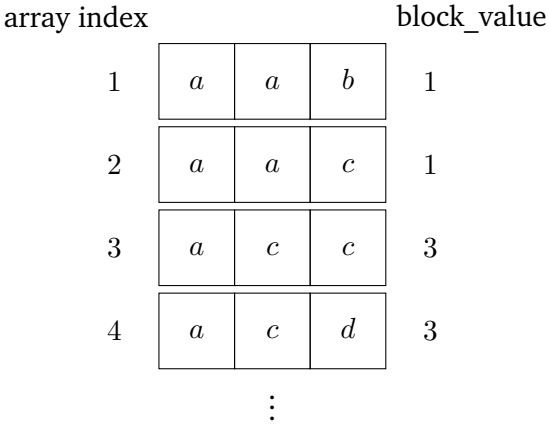


Figure 3.4.: How blocks are represented in the candidate generation algorithm.

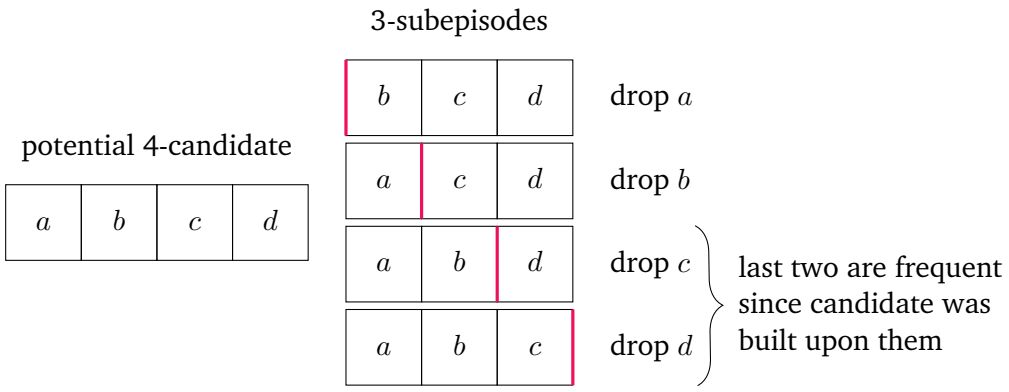


Figure 3.5.: The construction of l-sized subepisodes for an $(l+1)$-sized candidate.

By initializing j to $\mathcal{F}_l$.block_start$[i]$, j can be less than i, and candidates α for which $\alpha[l] > \alpha[l+1]$ will be generated, alleviating the constraint on parallel episodes.

Given the fact that $\mathcal{F}_l$ is sorted lexicographically, and because of the order in which the algorithm traverses $\mathcal{F}_l$, it is easy to see that the output array $\mathcal{C}_{l+1}$ of candidates will be sorted lexicographically as well.

After constructing a potential candidate (lines 9 and 10), it needs to be checked whether all of its subepisodes are frequent. Constructing all l-sized subepisodes of an $(l+1)$-sized candidate α is straightforward. For each subepisode, one node from α's graph is left out, until all nodes have been left out. For serial episodes, the edges of the subepisode are constructed such that the order of the topological sort is preserved.

For any potential candidate, two subepisodes in particular are already known to be frequent, namely the episodes it was built from. Those subepisodes are obtained by leaving out the last and the second to last array elements, respectively. That's why $l-1$ is the largest value that y takes in the algorithm.

We should note that in the subepisode construction procedure described above, some of the subepisodes may be equivalent to each other, when an event type occurs multiple times in a candidate. For candidate $\{a, a, b\}$, subepisode $\{a, b\}$ will be constructed twice.

3.2.1. Notes on the implementation

After constructing a subepisode, the algorithm tests whether or not it appears in $\mathcal{F}_l$. There are a few different options to realize this check.

1. Performing a linear search of the array, time is linear in the number of episodes in $\mathcal{F}_l$.
2. Thanks to the fact that $\mathcal{F}_l$ is sorted, we can perform a binary search, which has

logarithmic time complexity.

3. One other option is to use a hash table, which requires copying the episodes in $\mathcal{F}_l$ into a different data structure during initialization, but can provide constant time lookup afterwards.

Taking the first option as a naive first approach turned out to be too slow.

The two latter options performed much better, and gave comparable running times, likely indicating that the subepisode membership test wasn't a bottleneck anymore.

3.3. Determining the fixed-window frequency for parallel episodes

Algorithm 3 (adapted from [9]) recognizes parallel episodes in an event sequence $\boldsymbol{s} = (s, T_s, T_e)$. As stated before, parallel episodes impose no order on the occurrence of events in the window. Therefore, to recognize an episode in the sequence, it suffices to know that there are enough events of each type currently in the window. More formally, a parallel episode α occurs in a window if for each event type A, the window contains at least as many events of type A as there are nodes of type A in α's graph.

The algorithm makes one pass over the sequence, timestamp by timestamp, using a sliding window of size ρ. At any point during iteration, there are a few timestamps of interest.

In the algorithm — and further algorithms that iterate the sequence — variable *start* always refers to the smallest timestamp of the current window. It is the timestamp that will be dropped from the window the following iteration; at that timestamp we find the "oldest" events still in the window. The "newest" events in the window — those which just entered the window — are found at $(\text{start} + \rho - 1)$. We'll call the former the *back* of the sliding window, and the latter the *front*.

To prevent having to consider a special case for the beginning and the end of the iteration over the sequence, the algorithm starts with just the first valid timestamp T_s at the front of the sliding window. All other timestamps within the window are outside of the sequence at this point. This first window can be written as $\boldsymbol{s}[T_s - \rho + 1, T_s + 1)$. At the end of the sequence, the same happens; the last subwindow of the iteration is $\boldsymbol{s}[T_e, T_e + \rho)$, which contains just the last timestamp in $\boldsymbol{s}$. Figure 3.6 shows this way of iterating visually.

3.3.1. Operation

Recognition is accomplished as follows. For each event type A, a counter A.count is maintained. This counter denotes how many events of type A are currently in the window. When a new event of type A enters the window (line 15 and further), this counter

Algorithm 3 Recognizing parallel episodes using the fixed window frequency measure.
Input: A collection $\mathcal{C}$ of parallel episodes, an event sequence $\boldsymbol{s} = (s, T_s, T_e)$, a window width ρ, and a frequency threshold *min_fr*.
Output: The episodes of $\mathcal{C}$ that are frequent in $\boldsymbol{s}$ with respect to ρ and *min_fr*.

```
1:  /* Initialization */
2:  for all α in C do
3:      for all A in α do
4:          A.count ← 0
5:          for i ← 1 to |α| do contains(A, i) ← ∅
6:  for all α in C do
7:      for all A in α do
8:          a ← number of elements of type A in α
9:          contains(A, a) ← contains(A, a) ∪ {α}
10:     α.event_count ← 0
11:     α.freq_count ← 0
12: /* Recognition */
13: for start ← T_s − ρ + 1 to T_e do
14:     /* Bring new events to the window */
15:     for all events (A, t) in s such that t = start + ρ − 1 do
16:         A.event_count ← A.event_count + 1
17:         for all α ∈ contains(A, A.count) do
18:             α.event_count ← α.event_count + A.count
19:             if α.event_count = |α| then α.in_window ← start
20:     /* Drop out old events from the window */
21:     for all events (A, t) in s such that t = start − 1 do
22:         for all α ∈ contains(A, A.count) do
23:             if α.event_count = |α| then
24:                 α.freq_count ← α.freq_count − α.in_window + start
25:             α.event_count ← α.event_count − A.count
26:         A.event_count ← A.event_count − 1
27: /* Output */
28: for all episodes α in C do
29:     if α.freq_count ≥ min_fr then output α
```

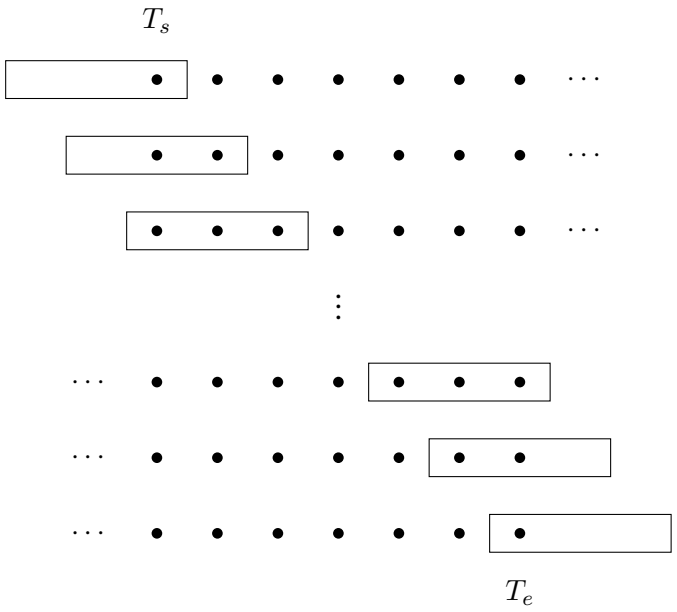


Figure 3.6.: Visualization of the way Algorithm 3 and further algorithms pass a sliding window over the sequence. Black dots represent timestamps that belong to the sequence, regardless of whether events occur at that timestamp.

is incremented by 1. Then, for all episodes α containing A.count nodes of type A, a per-episode counter α.event_count is increased by A.count, indicating that there are currently enough events of type A in the window to satisfy an occurrence. If this is the case for all event types in α, then α.event_count $= |\alpha|$, and so the window contains α.[1] To mark the point in time at which the episode entered the sliding window, variable α.in_window gets set to *start*.

When an event of type A leaves the window (line 21 and further), for all episodes α with A.count nodes of event type A, if α.event_count $= |\alpha|$, then α has been contained in a number of windows, but is now no longer. The episode was covered by (start $-$ α.in_window) subsequent windows, and so α.freq_count gets updated accordingly. After performing this for all relevant episodes, A.count gets decreased by 1 to reflect the fact that an event of type A has left the sliding window.

Figure 3.7 illustrates this the recognition of a parallel episode by Algorithm 3.

[1] Actually, the algorithm will find occurrences of episodes in the window $\boldsymbol{s}[\text{start} - 1, \text{start} + \rho)$, which is a window of size $\rho + 1$, which is one greater than the specified window size ρ. This is because new events coming into the window at time (start $+ \rho - 1$) are processed before old events are removed. However, since the events at time (start $- 1$) are removed immediately afterwards — before the sliding window moves again — occurrences spanning a width of $\rho + 1$ will have no effect on α's frequency. This will become clear later, when we explain how the algorithm determines the frequency.

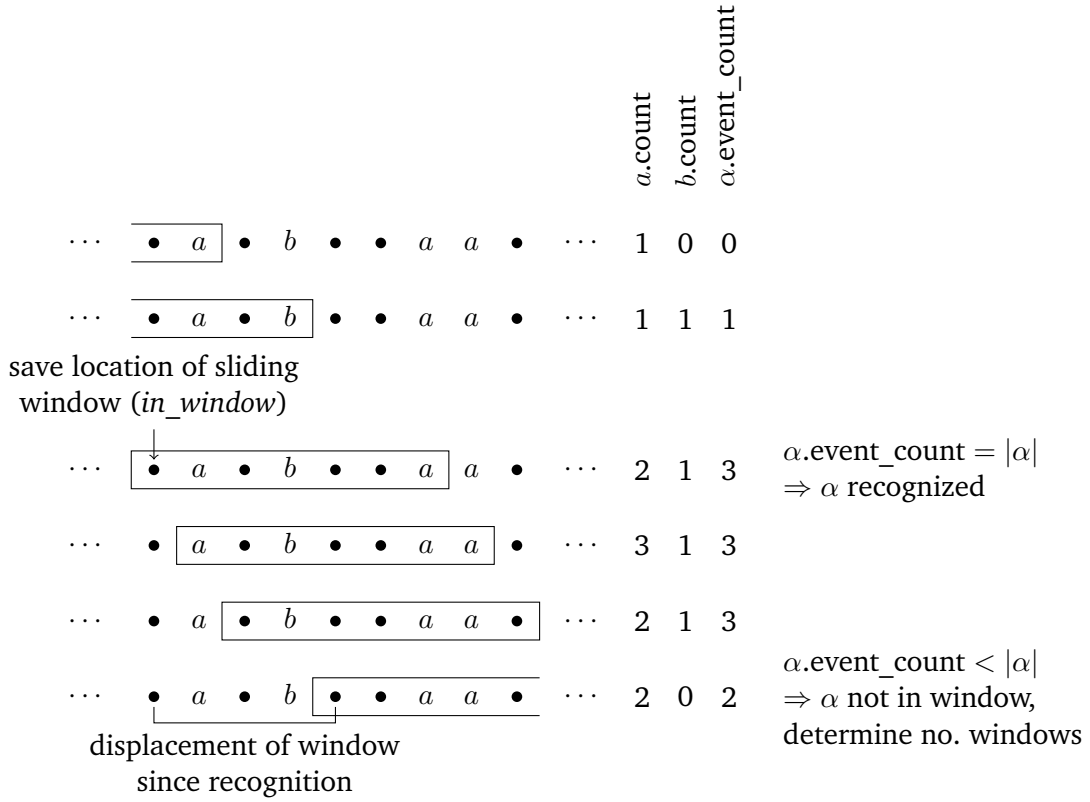


Figure 3.7.: Recognition of parallel episode $\alpha = \{a, a, b\}$ using the fixed window frequency measure. Black dots are timestamps which don't contain a or b.

3.3.2. Notes on the implementation

The *contains* data structure requires some sort of key-value storage. Typically, these types of data structures are either tree-based or hash-based. Lookup of a single key is generally faster with a hash-based data structure, while traversal of all elements is faster with a tree-based solution, especially if the traversal needs to be in-order. In this case, *contains* never needs to be traversed, so one would expect a hash-based data structure to be the best choice. This turned out to be the case experimentally. *contains* should store a reference to the episodes in $\mathcal{C}$, such that all updates to episode-specific variables are performed on $\mathcal{C}$.

3.4. Determining the fixed-window frequency for serial episodes

Algorithm 4 determines the fixed-window frequency of serial episodes in a sequence. By definition serial episodes impose a total order on the nodes, and consequently on the events in the sequence in order to satisfy an occurrence of an episode. Serial episodes are therefore recognized using simple automata, instances of which advance state as events are encountered. Just like the algorithm for recognizing parallel episodes, Algorithm 4 (adapted from [9]) iterates over the sequence once.

Each episode has its own automaton, which consists of $|\alpha|$ states: each state corresponds to a node in the episode. For an episode $(\{v_1, \ldots, v_n\}, \{(v_p, v_q) \mid p < q\}, lab)$, state j refers to node v_j. Then an instance of the automaton for α being in a state j denotes that the episode has been recognized up to (and including) the j-th node. When in state $j < n$ and upon encountering an event (A, t) such that $A = lab(v_{j+1})$, the instance will transition to state $j + 1$.

When an automaton instance of α reaches state $|\alpha|$, the episode has been successfully recognized, and from that point on, the episode will occur in a number of windows as the sliding window slides over the sequence. Once recognized, determining the occurrence's contribution to the frequency count happens in much the same way as it does in Algorithm 3 — store the point in time at which the episode first occurred within the sliding window, and when the occurrence leaves the sliding window, determine the displacement of the sliding window since the stored timestamp.

Regardless of the state an automaton instance reached, the instance is removed when the timestamp at which the instance was initialized falls out of the sliding window.

Figure 3.8 illustrates the principle with an example.

3.4.1. Data structures

The algorithm uses some bookkeeping data structures, which get updated as the sequence gets read. We'll discuss the most important ones.

- **waits** maps an event type to a set of pairs of the form (α, j), where α is an episode and j represents a state in the episode's automaton. If a pair (α, j) is in waits(A), and $j > 1$, then α has an instance of its automaton currently in state $(j - 1)$ and is waiting for an event of type A to advance to the next state. Throughout the iteration over the sequence, waits$(\alpha[1])$ will always contain $(\alpha, 1)$ for each episode α, since a new automaton instance should be instantiated any time the first event of a serial episode occurs.
- **begins_at** maps a timestamp to a set of pairs (α, j). If (α, j) is in begins_at(t), then α has an instance of its automaton in state j, and it was initialized at timestamp t.

Algorithm 4 Recognizing serial episodes using the fixed window frequency measure.
Input: An array of serial episodes $\mathcal{C}$, an event sequence $\boldsymbol{s} = (s, T_s, T_e)$, a window width ρ, and a frequency threshold *min_fr*.
Output: The episodes of $\mathcal{C}$ that are frequent in $\boldsymbol{s}$ with respect to ρ and *min_fr*.

1: **for all** $\alpha \in \mathcal{C}$ **do** ▷ Initialization
2: **for** $i \leftarrow 1$ to $|\alpha|$ **do**
3: α.initialized $\leftarrow$ *uninitialized*
4: waits($\alpha[i]$) $\leftarrow \emptyset$
5: **for all** $\alpha \in \mathcal{C}$ **do**
6: waits($\alpha[1]$) $\leftarrow$ waits($\alpha[1]$) $\cup \{(\alpha, 1)\}$
7: α.freq_count $\leftarrow 0$
8: **for** $t \leftarrow T_s - \rho$ to $T_s - 1$ **do** begins_at(t) $\leftarrow \emptyset$
9: **for** start $\leftarrow T_s - \rho + 1$ to T_e **do** ▷ Recognition
10: begins_at(start $+ \rho - 1$) $\leftarrow \emptyset$
11: transitions $\leftarrow \emptyset$
12: **for all** events (A, t) in s such that $t =$ start $+ \rho - 1$ **do**
13: **for all** $(\alpha, j) \in$ waits(A) **do**
14: **if** $j = |\alpha| \wedge \alpha$.initialized$[j]$ = *uninitialized* **then**
15: α.in_window $\leftarrow$ start
16: **if** $j = 1$ **then**
17: transitions $\leftarrow$ transitions $\cup \{(\alpha, 1,$ start $+ \rho - 1)\}$
18: **else**
19: transitions $\leftarrow$ transitions $\cup \{\alpha, j, \alpha$.initialized$[j-1]\}$
20: begins_at(α.initialized$[j-1]$) $\leftarrow$
21: begins_at(α.initialized$[j-1]$) $\setminus \{(\alpha, j-1)\}$
22: α.initialized$[j-1] \leftarrow$ *uninitialized*
23: waits(A) $\leftarrow$ waits(A) $\setminus \{(\alpha, j)\}$
24: **for all** $(\alpha, j, t) \in$ transitions **do**
25: **if** α.initialized$[j] \neq$ *uninitialized* **then**
26: begins_at(α.initialized$[j]$) $\leftarrow$ begins_at(α.initialized$[j]$) $\setminus \{(\alpha, j)\}$
27: α.initialized$[j] \leftarrow t$
28: begins_at(t) $\leftarrow$ begins_at(t) $\cup \{(\alpha, j)\}$
29: **if** $j < |\alpha|$ **then**
30: waits($\alpha[j+1]$) $\leftarrow$ waits($\alpha[j+1]$) $\cup \{(\alpha, j+1)\}$
31: **for all** $(\alpha, l) \in$ begins_at(start $- 1$) **do**
32: **if** $l = |\alpha|$ **then**
33: α.freq_count $\leftarrow \alpha$.freq_count $- \alpha$.in_window $+$ start
34: **else**
35: waits($\alpha[l+1]$) $\leftarrow$ waits($\alpha[l+1]$) $\setminus \{(\alpha, l+1)\}$
36: α.initialized$[l] \leftarrow$ *uninitialized*
37: **for all** episodes α in $\mathcal{C}$ **do**
38: **if** α.freq_count $\geq$ min_fr **then**
39: output α

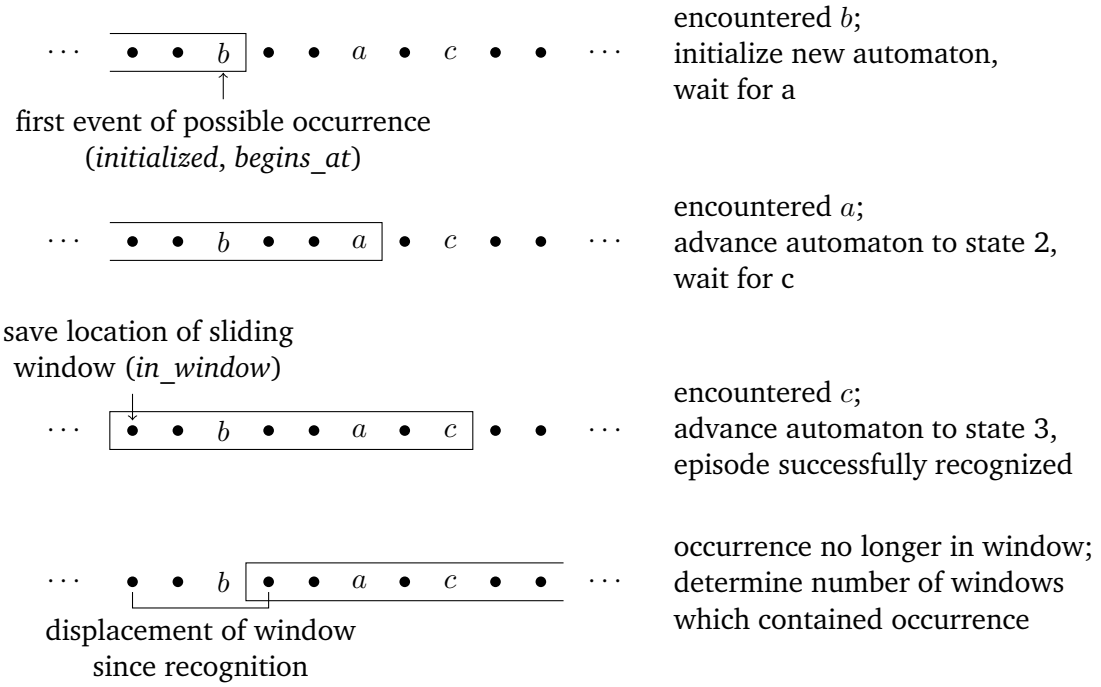


Figure 3.8.: Illustration for the recognition of serial episode $b \rightarrow a \rightarrow c$. Black dots are timestamps which don't contain b, a or c.

- Each episode α has an $|\alpha|$-element array called **initialized**, where α.initialized$[j]$ contains the timestamp in the sequence at which the instance currently in state j was initialized. If for a certain state there is currently no active instance, its corresponding element in the initialized array will be some special value — [9] chose 0, which we changed to *uninitialized* for clarity and such that 0 can be a valid timestamp if necessary.

  Note that *initialized* allows at most one automaton to be in a given state at any time. This approach works because it is not useful to have multiple instances in the same state. If one automaton instance reaches a common state with another instance, they will simply make transitions simultaneously until the earlier instance gets removed. It suffices to maintain the instance which reached the common state last, since it was initialized at a later timestamp, and thus will be the last to be removed.

- All of the state transitions to be performed for a given timestamp are collected in a list, **transitions**, before they actually get executed. If they were executed immediately, an automaton could be incorrectly overwritten if there there are multiple events in a single timestamp. We will come back to this in section 3.4.4.

3.4.2. Operation

With the data structures just described in mind, we will go through the algorithm in detail to see how it operates.

The initialization is rather straightforward. One thing to note is that the permanent pairs of the *waits* sets are constructed (line 6). As stated before, for each episode α, waits$(\alpha[1])$ contains $(\alpha, 1)$ from the start, and throughout the algorithm, to initialize a new automaton instance for α any time an event of type $\alpha[1]$ occurs.

Then a sliding window is passed over the sequence, in the same manner Algorithm 3 does (line 9). Each time the sliding window is advanced, first the events that just came into the window are processed (line 12), at time $(\text{start} + \rho - 1)$. Say an event of type A came in. Thanks to the *waits* data structure, all automaton instances that should make a state transition are found efficiently. For each of the pairs (α, j) in waits(A), the following steps are performed:

1. Lines 14 and further: If the automaton instance has reached state $|\alpha|$, that is, the episode now occurs within the sliding window,[2] then α.in_window gets set to *start* to mark the time at which the episode entered the sliding window. Except if another instance is already in state $|\alpha|$, then α.in_window was already set. If we would overwrite the value in that case, the windows covering the previous occurrence would not be counted.
2. Lines 16 and further: The transition to be made is captured in the list of transitions.
3. Lines 20 and further: The information regarding the old state is removed from data structures *begins_at*, *initialized*, and *waits*, if we're not initializing a new automaton instance.

When all new events have been processed, the transitions are finalized: the new state information is written to *initialized*, *begins_at*, and *waits*.

After applying the state transitions, the events no longer in the window are dropped. This entails looking up all automata which were instantiated at timestamp $(\text{start} - 1)$ using the *begins_at* data structure (line 32 and further):

1. If the automaton instance was in state $|\alpha|$, the episode occurred within the window until now. When the episode was first recognized, the location of the sliding window was stored in α.in_window, and so the number of fixed windows in which the episode occurred can be determined by the displacement of the sliding window since then.
2. If the automaton instance was not in state $|\alpha|$, the *waits* data structure is updated to reflect the removal of the instance.
3. The *initialized* array of the episode is updated to reflect the removal of the automa-

[2] Again, as explained in footnote 1 on page 30, this statement needs to be nuanced. The same exact reasoning applies here.

ton instance.

3.4.3. Adapting the algorithm presented in [9]

While implementing this algorithm we found an oversight in [9] for the pseudocode of Algorithm 4 (Algorithm 5 in [9]), related to multiple instances of an episode's automaton reaching a common state. Note that in [9], the update to *begins_at* of lines 25 and 26 are missing. (In their version of the algorithm, these statements should appear in between lines 26 and 27.)

In lines 25 through 30, a transition of an automaton instance is being applied. The *initialized* array gets updated, potentially overwriting a previous instance in the same state. *begins_at* gets updated with the new state, but without the statements of lines 25 and 26, the information about the potentially overwritten instance does not get removed from *begins_at*. We'll illustrate how this affects the output with an example.

Assume that the statements aren't there, as in [9]. Using a window width of 2, and scanning the sequence $e\,a\,e\,c$. Consider the recognition of the serial episode $\alpha = e \to c$. Obviously the sequence contains α, and a window width of 2 suffices to recognize an instance of the episode in the sequence. Upon encountering the first e, an automaton instance gets initialized for α (lines 25 through 30). This includes, as discussed before:

- setting the timestamp in the *initialized* array;
- adding $(\alpha, 2)$ to the waits(c) set (so that if c is encountered, the automaton can transition to the next state);
- adding $(\alpha, 1)$ to begins_at(1) (to facilitate the removal of the the automaton instance once timestamp 1 falls out of the window).

When the second e is read (at time 3 and when start $= 2$), a new instance of the automaton is created, and since the previous instance is still in state 1 — no c has been encountered — the previous instance must be overwritten. Indeed, this happens for the *initialized* data structure at line 25. However, *begins_at* is never updated to reflect this: $(\alpha, 1)$ remains in begins_at(1). Because of this, the newly created automaton is wrongly removed just afterwards, because timestamp 1 — the timestamp at which the first instance was initialized — fell out of the window (while iterating begins_at$(\text{start} - 1 = 1)$, lines 32 through 36). Then $(\alpha, 2)$ is no longer in waits(c), and α will fail to be recognized.

This inconsistency in *begins_at* is a problem any time an older instance of an episode's automaton instance gets overwritten by a more recent one reaching the same state. In some cases, as in the example, an occurrence of an episode may fail to be recognized entirely, while in other cases it may get recognized but with an incorrect frequency count, again due to an automaton instance being removed early.

3.4.4. The necessity of the *transitions* data structure

Earlier we mentioned that the *transitions* data structure, which captures all transitions to execute at a timestamp before finalizing them, exists for good reason. In short, an automaton instance in state j could incorrectly overwrite an instance in state $j + 1$ if both instances are to transition. Consider the following example. Say $\alpha = a \rightarrow b \rightarrow c$ is in the collection of candidates, and consider the event sequence $\langle (a, 1), (b, 2), (a, 3), (b, 4), (c, 4) \rangle$. Note that the last two events occur at the same timestamp. Assume the window width is at least 4 so that it is sufficiently large to recognize α.

1. When $(a, 1)$ enters the window, a new automaton for α gets initialized.
2. When $(b, 2)$ comes in, the automaton transitions to state 2.
3. When $(a, 3)$ comes in, a new automaton instance for α is initialized. Now α's *initialized* array contains:

$$\langle 3, 1, \textit{uninitialized} \rangle$$

4. Next, $(b, 4)$ comes in. The most recently initialized instance will transition to state 2.
5. Finally, $(c, 4)$ enters the window. The oldest automaton instance should transition to state 3.

If at step 4 the transition were performed immediately, then the *initialized* entry of the older automaton instance would be incorrectly overwritten; *initialized* contains

$$\langle \textit{uninitialized}, 3, \textit{uninitialized} \rangle$$

while it should contain

$$\langle \textit{uninitialized}, 3, 1 \rangle$$

Now the data structures are in an inconsistent state: according to *initialized* there is only one automaton instance, while the other data structures still contain information regarding the overwritten instance. Moreover, α doesn't get recognized, while it clearly occurs in the sequence.

Collecting all transitions to be performed at a given timestamp before executing them, will prevent *initialized* entries being incorrectly overwritten.

3.4.5. Notes on the implementation

Like Algorithm 3, this algorithm needs key-value storage in order to be able to look up information efficiently. More specifically for *waits*. Again, lookup is important, traversal is not. Therefore, again, a hash-based data structure is the best choice.

While at first sight, the *begins_at* data structure might seem best-suited for key-value storage as well, note that it only needs to contain information about the subsequent timestamps currently in the window $[\text{start} - 1, \text{start} + \rho)$. So, the $\rho + 1$ sets can be accessed in constant-time complexity using a circular array. Each time the sliding window is advanced, the oldest information is dropped to make room for new information. With a 1-indexed buffer, begins_at(t) can be accessed with index $(t - \text{start} + 2)$.

3.5. Determining the disjoint-window frequency

Algorithms for finding the disjoint-window frequency of parallel and serial episodes were presented by [6]. Their algorithms find only the disjoint-window frequency, however. We also want to be able to determine the minimal-window confidence of association rules, and so we also need minimal windows themselves — including overlapping windows. Therefore, we determine the disjoint-window frequency in two steps:

1. First, we find all minimal windows, with Algorithm 6 (in section 3.6) and Algorithm 7 (in section 3.7).
2. Given those minimal windows, Algorithm 5 (in this section) finds the disjoint-window frequency.

Later, the minimal windows found in step 1 can be reused to generate association rules.

Now let's look into the execution of step 2.

Algorithm 5 Determining the disjoint-window frequency of an episode.
Input: List W of minimal windows of the episode in a sequence s, ordered by time in the sequence.
Output: The episode's disjoint-window frequency in s.

```
1: u ← 0
2: b_l ← −∞
3: for all [a, b) ∈ W do
4:     if b_l ≤ a then
5:         u ← u + 1
6:         b_l ← b
7: output u
```

Given a list of all of the minimal windows that cover an episode, ordered by timestamp, the disjoint window frequency can be calculated as follows. Select the first window if the list is not empty. Then, while iterating the list, select a window if it does not overlap with the most recently selected window. The disjoint-window frequency is the number of windows that were selected.

Given that for any two minimal windows of an episode in a sequence, one cannot be a subwindow of the other, it is intuitively clear that this procedure produces a maximal

selection of disjoint windows, and thus finds the disjoint-window frequency.

Algorithm 5 implements this procedure. b_l contains the end timestamp of the window that was selected last. Then if a later window starts before b_l, then it overlaps with the last selected window; if not, it gets selected.

3.6. Finding all minimal windows of parallel episodes

Algorithm 6 Finding all of the minimal windows of a collection $\mathcal{C}$ of parallel episodes in a sequence $\boldsymbol{s}$.
Input: A collection $\mathcal{C}$ of parallel episodes, an event sequence $\boldsymbol{s} = (s, T_s, T_e)$, a window width ρ, and a frequency threshold *min_fr*.
Output: All episodes in $\mathcal{C}$, along with all minimal windows of each episode (including overlapping windows).

```
1:  /* Initialization */
2:  for all event types A do
3:      Q(A) ← empty queue
4:  for all α in C do
5:      for all A in α do
6:          consider_max(α, A) ← 0
7:          num_needed(α, A) ← number of elements of type A in α
8:          contains(A) ← contains(A) ∪ {α}
9:  /* Recognition */
10: for start ← T_s − ρ + 1 to T_e − 1 do
11:     /* Drop out old events from the window */
12:     for all events (A, t) in s such that t = start − 1 do
13:         Q(A).pop()
14:     /* Bring in new events to the window */
15:     for all events (A, t) in s such that t = start + ρ − 1 do
16:         Q[A].push(t)
17:         for all α in contains(A) do
18:             consider_max(α, A) ← consider_max(α, A) + 1
19:             if ∀A in α : min(|Q(A)|, consider_max(α, A)) ≥ num_needed(α, A) then
20:                 /* Minimal window found; determine the start */
21:                 window_start ←
22:                     min{t | A in α ∧ t = Q(A)[|Q(A)| − num_needed(α, A) + 1]}
23:                 for all A in α do q ← Q(A)
24:                     if window_start = q[|q| − num_needed(α, A)) + 1] then
25:                         consider_max(α, A) ← num_needed(α, A) − 1
26:                 append [window_start, start + ρ) to α.minimal_windows
```

Finding the minimal windows of parallel episodes is slightly more complex than determining their fixed-window frequency.

Thanks to the requirement that all minimal windows have a width of at most ρ, we can keep using a sliding window of fixed width, just like the algorithms for the fixed-window frequency. Then only the part of the sequence included in the sliding window needs to be considered at any time, and any minimal window we find will be a subwindow of the current sliding window. We'll still be scanning the front of the sliding window for new events as before. Now, when a new event (A, t) comes in, we need to answer the following questions regarding episodes α containing A:

1. With this new event in the window, is there a minimal window $[a, \text{start} + \rho)$ of α which ends at the front of the current sliding window?
2. If so, where does the minimal window start: what is a?

Since minimal windows will often be smaller than the fixed-width sliding window, a previously recognized minimal window may still be within the sliding window several timestamps after it has been recognized. We need some way of excluding that previous minimal window from consideration, on a per-episode basis. Also, we need to be able to find the start of a new minimal window efficiently. It should be clear, then, that we need to keep more information than we did in Algorithm 3, where we just kept the number of events of each type currently in the sliding window, and for each episode, those event types for which there are enough events in the window to satisfy an occurrence anywhere in the sliding window.

Any newly discovered minimal window has to start *after* the start of the previous one, because:

1. we already know it ends after the start of the previous one, since the sliding window has advanced, and
2. one minimal window of α cannot be a subwindow of another.

So, given the previous minimal window $[a, b)$, all of the events that cannot contribute to a new minimal window are those with a timestamp $t \leq a$. All those with $t > a$ *can* contribute.

Algorithm 6 keeps a queue for each event type A, which contains the timestamps of the events of type A currently in the sliding window. The queues are referred to by Q in the pseudocode. When a new event (A, t) comes in, t is pushed onto $Q(A)$ (line 16). Then, for each episode α containing A, it is determined whether we have a new minimal window, by testing for each event type A in α whether $Q(A)$ contains enough events, excluding the events at the start of the most recently found minimal window (line 19). The exclusion of these events is realized by **consider_max**.

If we have determined that there are enough events in consideration to form a new minimal window, then the queues allow us to find the start of the window (line 21). If α contains n nodes of type A, then we look at the n-th most recently added timestamp

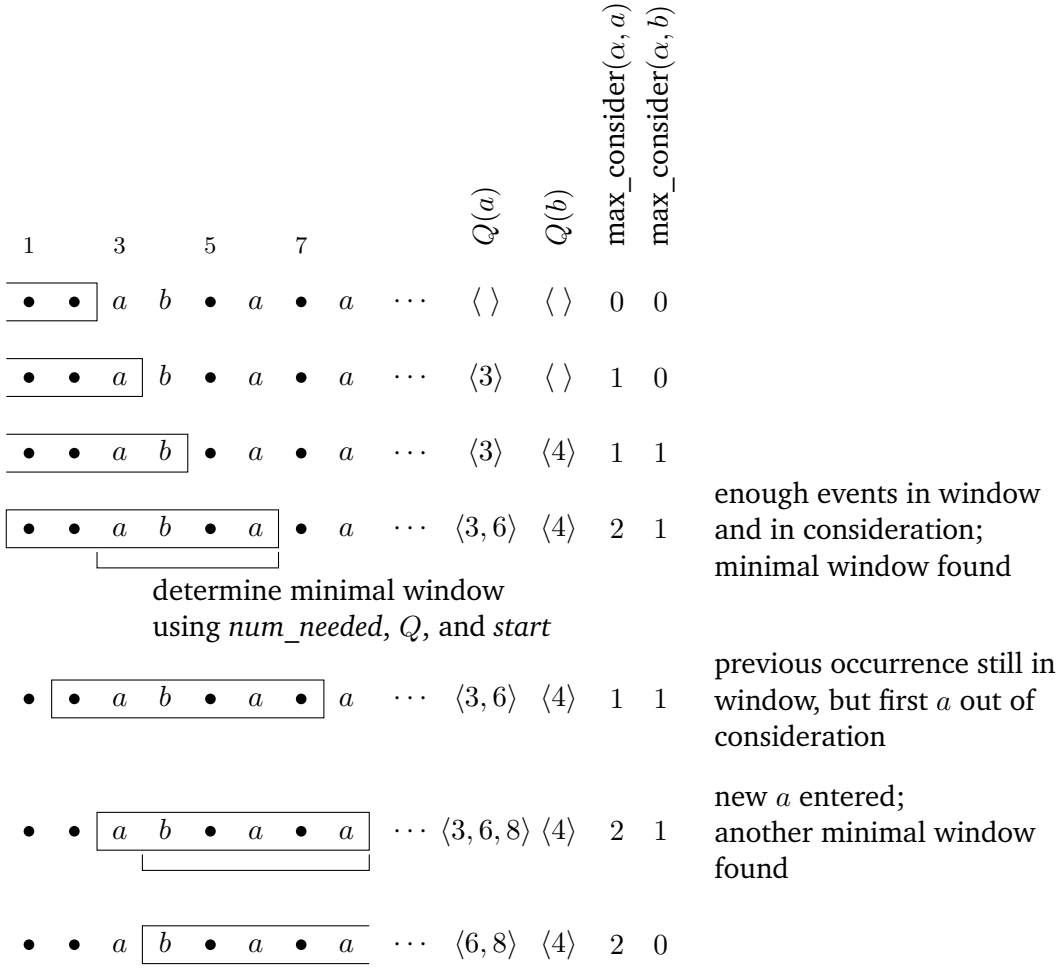


Figure 3.9.: Recognition of parallel episode $\alpha = \{a, a, b\}$, finding its minimal windows in the sequence. Black dots are timestamps that don't contain a or b.

to $Q(A)$. The earliest such timestamp, for all event types in α, is the start of the minimal window.

Figure 3.9 shows how the algorithm finds the minimal windows of an episode $\{a, a, b\}$ in an example sequence.

3.6.1. Notes on the implementation

We need random access into the queues in order to find the start of a minimal window (line 21). The queues should therefore be implemented with a circular array.

3.7. Finding all minimal windows of serial episodes

While finding the minimal windows of parallel episodes was slightly more complicated than finding their fixed-window frequency, finding the minimal windows of serial episodes is actually simpler than finding their fixed-window frequency, and can be accomplished with small changes to Algorithm 4.

Algorithm 7 accomplishes recognition of serial episodes using the minimal window frequency measure.

Once an automaton instance for an episode α has reached state $|\alpha|$, we know immediately where a minimal window starts and ends — it starts at the point where the automaton was initialized, and ends at the front of the current sliding window:

$$\boldsymbol{s}[\alpha.\text{initialized}[|\alpha|], \text{start} + \rho)$$

So, in contrast to the situation for fixed windows, now there is no second phase in which to keep track of the displacement of the sliding window, and therefore the *in_window* variable isn't needed.

In this adaptation, old events are removed before new ones are processed, because the situation explained in footnote 1 on page 30 would become a problem when the frequency isn't determined by the displacement of the sliding window anymore.

3.8. Determining the weighted-window frequency

For the weighted-window frequency, selecting the non-overlapping minimal windows such that total weight is maximal, is a bit more involved than selecting the largest possible number of non-overlapping minimal windows for the disjoint-window frequency (section 3.5).

In Algorithm 8 (from [4]), lines 2 through 5, for each minimal window $[a_i, b_i)$ the *next* window disjoint from $[a, b)$ is determined, and referred to by d_i. More precisely, d_i refers to the window $[a_j, b_j)$ with the smallest a_j such that $a_j \geq b_i$.

Then, in lines 6 through 8, the weighted-window frequency is determined through a backwards pass over the minimal windows. Each c_i is the optimal choice of windows i through n. If all c_j with $j > i$ are known, then c_i can be calculated as

$$c_i = \max((b_i - a_i)^{-1} + c_{d_i},\ c_{i+1})$$

where the leftmost argument to $\max$ represents selecting window i, and therefore "unselecting" all earlier selected windows that overlap with window i, and the rightmost argument to $\max$ represents not selecting window i and continuing with all previously selected windows.

In this way, the d_i essentially allow us to *backtrack* to undo earlier choices in favour of a better one.

Algorithm 7 Recognizing serial episodes using the minimal window frequency measure.
Input: An array of serial episodes $\mathcal{C}$, an event sequence $\boldsymbol{s} = (s, T_s, T_e)$, a window width ρ, and a frequency threshold *min_fr*.
Output: All episodes in $\mathcal{C}$, along with all minimal windows of each episode (including overlapping windows).

for all $\alpha \in \mathcal{C}$ **do** ▷ Initialization
  for $i \leftarrow 1$ to $|\alpha|$ **do**
    α.initialized$[i] \leftarrow$ *uninitialized*
    waits$(\alpha[i]) \leftarrow \emptyset$
for all $\alpha \in \mathcal{C}$ **do**
  waits$(\alpha[1]) \leftarrow$ waits$(\alpha[1]) \cup \{(\alpha, 1)\}$
  α.minimal_windows $\leftarrow \emptyset$
for $t \leftarrow T_s - \rho$ to $T_s - 1$ **do** begins_at$(t) \leftarrow \emptyset$
for start $\leftarrow T_s - \rho + 1$ to T_e **do** ▷ Recognition
  begins_at(start $+ \rho - 1) \leftarrow \emptyset$
  transitions $\leftarrow \emptyset$
  for all $(\alpha, l) \in$ begins_at(start $- 1$) **do**
    if $l < |\alpha|$ **then**
      waits$(\alpha[l+1]) \leftarrow$ waits$(\alpha[l+1]) \setminus \{(\alpha, l+1)\}$
    α.initialized$[l] \leftarrow$ *uninitialized*
  for all events (A, t) in s such that $t =$ start $+ \rho - 1$ **do**
    for all $(\alpha, j) \in$ waits(A) **do**
      if $j = 1$ **then**
        transitions $\leftarrow$ transitions $\cup \{(\alpha, 1,$ start $+ \rho - 1)\}$
      else
        transitions $\leftarrow$ transitions $\cup \{\alpha, j, \alpha$.initialized$[j-1]\}$
        begins_at(α.initialized$[j-1]) \leftarrow$
          begins_at(α.initialized$[j-1]) \setminus \{(\alpha, j-1)\}$
        α.initialized$[j-1] \leftarrow$ *uninitialized*
        waits$(A) \leftarrow$ waits$(A) \setminus \{(\alpha, j)\}$
  for all $(\alpha, j, t) \in$ transitions **do**
    if α.initialized$[j] \neq$ *uninitialized* **then**
      begins_at(α.initialized$[j]) \leftarrow$ begins_at(α.initialized$[j]) \setminus \{(\alpha, j)\}$
    α.initialized$[j] \leftarrow t$
    begins_at$(t) \leftarrow$ begins_at$(t) \cup \{(\alpha, j)\}$
    if $j = |\alpha|$ **then**
      α.minimal_windows $\leftarrow$
        α.minimal_windows $\cup \{[\alpha$.initialized$[j],$ start $+ \rho)\}$
    else
      waits$(\alpha[j+1]) \leftarrow$ waits$(\alpha[j+1]) \cup \{(\alpha, j+1)\}$

Algorithm 8 Computing the weighted frequency of an episode in a sequence.
Input: A list of minimal windows $V = \langle\, [a_1, b_1), \ldots, [a_n, b_n)\, \rangle$ of episode α.
Output: $fr_w(\alpha)$

```
1: j ← 1
2: for all [a_i, b_i) ∈ V do
3:     while j ≤ n ∧ b_i > a_j do
4:         j ← j + 1
5:     d_i ← j
6: c_{n+1} ← 0
7: for i ← n down to 1 do
8:     c_i ← max((b_i − a_i)^{−1} + c_{d_i}, c_{i+1})
9: output c_1
```

3.9. Removing episodes which have been found infrequent

At first sight, it seems a trivial task to discard episodes from a list of candidates which have turned out to be infrequent. But one thing needs to be taken into consideration, namely the auxiliary *block_start* variables, used in the candidate generation algorithm (section 3.2). Each episode α has such a *block_start* value, which denotes the array index of the first episode in the list with which it shares the first $|\alpha|-1$ elements in the episode's array representation. If we remove episodes without further consideration, these indices will be invalidated. They should not be invalidated, however, since the next round of the candidate generation step relies on them. Hence we need to keep track of the blocks while constructing the list of frequent episodes. Algorithm 9 achieves this.

The algorithm stores the frequent episodes in a new data structure $\mathcal{F}$. While iterating over the episodes in order, the array index of the new block start is kept in *new_block_start*. If for an index i, $\mathcal{C}$.block_start$[i] = i$, we know that a new block started at position i in $\mathcal{C}$, and so *new_block_start* gets updated in order to start a new block in $\mathcal{F}$ as well.

3.10. Finding association rules from frequent episodes: high-level algorithm

Now that we have described the full solution for mining frequent episodes, let's turn to association rules. Algorithm 10 (adapted from [9]) shows the high-level procedure for generating confident association rules after mining frequent episodes.

On line 6, to decide whether a newly constructed association rule $\alpha \Rightarrow \beta$ is confident, its confidence value must be computed. For the fixed-window confidence this is trivial:

Algorithm 9 Removing infrequent episodes from a collection of candidates $\mathcal{C}$ for which *freq_count* is known.
Input: A sorted array of candidates $\mathcal{C}$, including their *block_start* values, and their *freq_count* values with respect to some sequence, and a minimum frequency threshold *min_fr*.
Output: A sorted array $\mathcal{F}$ of those episodes in $\mathcal{C}$ which are frequent, along with consistent *block_start* values.

```
1: new_block_start ← 1
2: F ← empty array
3: F.block_start ← empty array
4: for i ← 1; i ≤ |C|; i ← i + 1 do
5:     if C.block_start[i] = i then
6:         /* Encountered new block in C */
7:         new_block_start ← |F|
8:     if α.freq_count < min_fr then
9:         /* Episode infrequent; discard */
10:        continue with next i
11:    append α to F
12:    F.block_start[|F|] ← new_block_start
13: output F
```

Algorithm 10 Finding confident association rules composed of frequent episodes.
Input: A window width ρ, a frequency/confidence measure Ψ, a frequency threshold *min_fr*, and a confidence threshold *min_conf*.
Output: $\{(\alpha \Rightarrow \beta, c(\alpha \Rightarrow \beta)) \mid \beta \subset \alpha \wedge fr(\beta) \geq \text{min\_fr} \wedge c(\alpha \Rightarrow \beta) \geq \text{min\_conf}\}$

```
1: /* Find frequent episodes (Algorithm 1) */
2: compute F(s, ρ, min_fr)
3: /* Generate rules */
4: for all β ∈ F(s, ρ, min_fr) do
5:     for all α ⊂ β do
6:         if c_Ψ(α ⇒ β) ≥ min_conf then
7:             output α ⇒ β and c_Ψ(α ⇒ β)
```

given the frequency of both episodes, compute

$$c_f(\alpha \Rightarrow \beta) = \frac{fr(\beta)}{fr(\alpha)}$$

For the minimal-window confidence and the weighted-window confidence, we use the minimal windows found while mining frequent episodes; in algorithms 11 (section 3.11) and 12 (section 3.12), respectively.

3.11. Determining the minimal-window confidence

Algorithm 11 Computing the minimal-window confidence of an association rule $\alpha \Rightarrow \beta$.
Input: List of minimal windows $V = \langle [a_1, b_1), \ldots, [a_n, b_n) \rangle$ of episode α, list of minimal windows $W = \langle [p_1, q_1), \ldots, [p_m, q_m) \rangle$ of episode β.
Output: $c_m(\alpha \Rightarrow \beta)$

```
1: u ← 0
2: j ← 1
3: for all i ← 1 to n do
4:     while j ≤ m ∧ q_j < b_i do
5:         j ← j + 1
6:     if j ≤ m ∧ p_j ≤ a_i then u ← u + 1
7: output u/n
```

Algorithm 11, adapted from [4], computes the minimal-window confidence of an association rule $\alpha \Rightarrow \beta$, given a list V of the minimal windows of α and a list W of the minimal windows of β, both lists ordered by the timestamps that define the windows. Both lists are iterated once, simultaneously.

For each minimal window $[a_i, b_i)$ of α, we need to determine if there exists a minimal window $[p_j, q_j) \supset [a_i, b_i)$ of β. If so, $ext_m([a, b), \alpha, \beta) = 1$. For each window in V, (line 3), W is traversed (line 4) until for a minimal window $[p_j, q_j)$ in W, a comparison of the end timestamps of both windows determine that $q_j \geq b_i$. Then the start timestamps are compared (line 6): if $p_j \leq a_i$, then we know that $[p_j, q_j) \supset [a_i, b_i)$. Note that W does not need to be traversed from the beginning for each window in V, since both lists are ordered by timestamp.

If it turns out that $ext_m([a, b), \alpha, \beta) = 1$ for a minimal window $[a, b)$, then this gets registered by incrementing variable u (line 6).

3.11.1. Adapting the original algorithm presented in [4]

In [4], there is a small mistake in Algorithm 11 (Algorithm 1 in [4]), regarding the comparison of the end timestamps of two windows. In the condition for the *while*-statement,

W is traversed until a window with an end timestamp *strictly* greater than the end timestamp of the window in V currently being considered. However, one window can be a subwindow of another if their end timestamps are equal. If implemented as written in [4], then a minimal window in W could be skipped erroneously.

3.12. Determining the weighted-window confidence

Algorithm 12 Computing the weighted-window confidence of an association rule $\alpha \Rightarrow \beta$.
Input: A list of the minimal windows $V = \langle [a_1, b_1), \ldots, [a_n, b_n) \rangle$ of episode α, and a list of the minimal windows $W = \langle [p_1, q_1), \ldots, [p_m, q_m) \rangle$ of episode β.
Output: $c_w(\alpha \Rightarrow \beta)$

```
 1: c ← 0
 2: i ← 1
 3: for k ← 1 to n do
 4:     while i ≤ m ∧ p_i < b_k do
 5:         i ← i + 1
 6:     j ← i
 7:     l ← +∞
 8:     while j ≤ m ∧ p_j ≤ a_k do
 9:         if q_j − p_j ≤ l then
10:             l ← q_j − p_j
11:             i ← j
12:         j ← j + 1
13:     if l ∈ ℕ then
14:         c ← c + (b_i − a_i)/l
15: output c/n
```

The calculation of the weighted-window confidence of an association rule can be done in a similar vein as the minimal-window confidence (section 3.11). The main difference is that now, not only do we have to determine whether or not there exists a window in W contains a given $[a_k, b_k) \in V$; if such windows exist, we have to find the *smallest* one in order to compute $ext_w([a_i, b_i), \alpha, \beta)$, given definition 18.

This is what happens in the second *while*-loop in Algorithm 12 (line 8, algorithm adapted from [4]). After the first minimal window $[p_i, q_i) \supset [a_k, b_k)$ has been found, all subsequent windows $[p_j, q_j) \supset [a_k, b_k)$ are checked for their length. The index j for which the weight $(q_j - p_j)^{-1}$ is maximal, is stored in i, and the corresponding window contributes towards the weighted-window confidence.

3.12.1. Adapting the algorithm presented in [4]

Algorithm 3 from [4] — which Algorithm 12 was based on — does not account for the possibility that there is no minimal window of β that contains the current minimal window of α. To that end, we update c only if at least one such window exists.

4. Experiments

In this chapter, we perform experiments to answer the research question we posed in the introduction. Concisely put, we want to know whether we can find patterns in long sequences in a reasonable time, and whether there are interesting patterns to be found.

After choosing a number of datasets and showing some of their basic characteristics, we assess the performance of the implementation by running it on datasets of different kinds, and varying the parameters.

First, we'll briefly discuss how we ensured the correctness of our implementation.

4.1. Correctness

A big point of comparison while implementing our algorithms was a closed episode miner[1]. It allowed us to verify the correctness of our implementation. The closed episode miner differs in a few ways from our implementation:

- It mines general episodes, not just parallel and serial episodes.
- It mines closed episodes. An episode is closed if there are no superepisodes of the same frequency. Mining only closed episodes helps reduce the amount of output, by excluding non-closed episodes. The closed episode miner has options to mine non-closed episodes as well, which allows us to compare the output of both implementations.
- While our implementation finds episodes of one class at a time, as specified by the user — parallel or serial — the closed episode miner finds episodes of all classes at once — parallel, serial and general.

Given the above, in order to compare the two implementations in terms of their output, we have to enable the options that cause the closed episode miner to find non-closed episodes, since our implementation does not exclude non-closed episodes. With those options enabled, our implementation finds a subset of the episodes that the closed episode miner finds; and we have to filter out those episodes which don't match the class of episodes we're currently mining.

We validated the output of our implementation with the output of the closed episode miner, running the closed episode miner alongside our implementation on the same

[1] http://adrem.ua.ac.be/mining-closed-strict-episodes

α	$fr_w(\alpha)$ (ours)	$fr_w(\alpha)$ (Closepi)
$\{b, d\}$	0.2 $(1/5)$	$0.166\ldots$ $(1/6)$
$\{a, b, d\}$	0.2 $(1/5)$	$0.142857\ldots$ $(1/7)$
$\{a, b, e\}$	0.25 $(1/4)$	$1.66\ldots$ $(1/6)$

Table 4.1.: Differing weighted-window frequency values between our implementation (*ours*) and the closed episode miner (*Closepi*), mining the example sequence from Figure 2.1.

datasets and parameter configurations, then parsing and comparing the results. Thanks to the closed episode miner we were able to eliminate quite a few errors in our implementation.

We did find, however, a possible bug in the closed episode miner. When mining parallel episodes in the example sequence (Figure 2.1), using the weighted-window frequency and with a window width of 8, the frequencies for a few episodes differed, as shown in Table 4.1. Observing the sequence, each of these episodes has two overlapping minimal windows, of which the second one has a greater weight. Our implementation seems to correctly select the window with the higher weight, while the closed episode miner seems to choose the first window.

Further distilling the issue, we tested a simpler example: the sequence $\langle(a, 1), (a, 3), (a, 4)\rangle$, episode $\{a, a\}$, and a window width of at least 3. The weighted-window frequency is clearly $1/2$, being the weight of the minimal window $[3, 5)$, but the closed episode miner reported $1/3$, so it indisputably selected $[1, 4)$.

4.2. Datasets

We will conduct experiments using a number of datasets:

- *abstract*: a dataset consisting of the first 739 NSF award abstracts from 1990, merged into one long sequence[2].
- *tolstoy*: Leo Tolstoy's novel *Anna Karenina*, part of the public domain, and available at Project Gutenberg[3].
- *trains*: a dataset consisting of departure times of delayed trains in a Belgian railway station, for trains with a delay of at least three minutes. This data is anonymized, so we won't be able derive any meaning from the patterns, but it is an interesting dataset nonetheless, because contrary to the textual datasets, *trains* contains sparse, real-time data. The time interval between two subsequent timestamps represents one second. Its timespan of over 2.66 million seconds converts to over 730 hours.

[2] `http://kdd.ics.uci.edu/databases/nsfabs/nsfawards.html`
[3] `https://www.gutenberg.org/ebooks/1399`

dataset	$\lvert\Sigma\rvert$	$\lvert s\rvert$	$T_e - T_s$	type
abstract	51 346	67 828	67 828	dense
tolstoy	95 623	124 627	124 627	dense
trains	1 280	10 115	2 662 667	sparse

Table 4.2.: Some properties of the datasets (s, T_s, T_e).

The textual datasets were preprocessed by lemmatizing using the Porter stemmer[4] and by removing stop words.

Table 4.2 shows some statistics about each dataset, where $|\Sigma|$ is the size of the alphabet, $|s|$ is the number of events, and $T_e - T_s$ is the time range of the sequence. For dense sequences, $T_e - T_s = |s|$. For sparse sequences, a window of ρ contains on average $|s|\frac{\rho}{T_e - T_s}$ events. For instance, a window of 900 in *trains* (15 minutes) contains approximately 3.42 events on average.

We see that all of the datasets have quite large alphabets. Figure 4.1 shows the number of occurrences of the most frequent event types, ordered by frequency, for each of the datasets. We observe a long tail for *abstract* and *tolstoy*, with a small number of the event types occurring a significant number of times, and the vast majority of event types occurring very rarely. This is common for natural-language texts. For *trains* the event types aren't uniformly distributed either, but the tail is not very long. Note that the graphs for the textual datasets don't show all event types, although for *abstract* the rightmost event types occur only once, and those for *tolstoy* occur fewer than 10 times.

Figure 4.2 shows the cumulative distribution of event types in the *trains* dataset. We see that some periods of time are noticeably more active than others, so some windows will contain more information than others. With some imagination, in Figure 4.2b one might be able to discern rush hour from quieter periods, and day from night — perhaps even weekdays from the weekend.

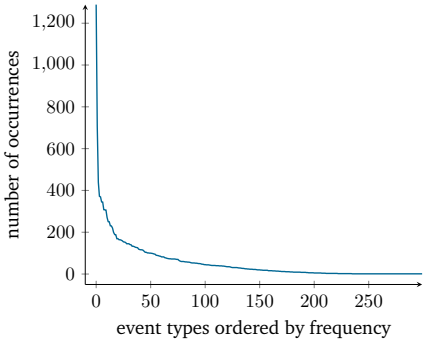
While we experiment on both dense and sparse datasets, results may differ for datasets with different event type distributions, or sparse datasets with events distributed differently through time. Our experiments will not generalize to all imaginable kinds of sequential data.

4.3. Performance

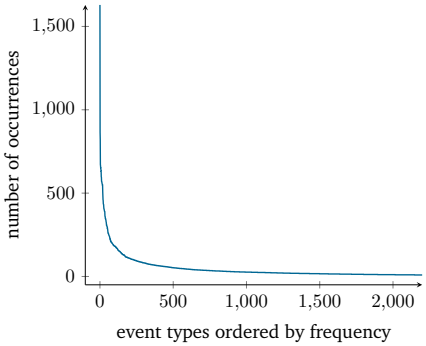
To assess the efficiency of the algorithm, we inspect the runtime for different input parameters: episode classes and frequency measures, over a range of window widths and thresholds.

The performance experiments were ran as follows. For specified episode classes, frequency measures, a list of window widths, and a range of frequency thresholds, an

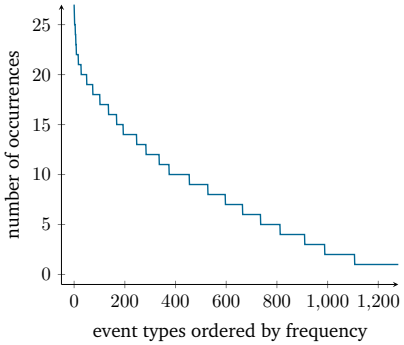
[4] `https://tartarus.org/martin/PorterStemmer`



(a) The 300 most frequent event types in *abstract*.



(b) The 2200 most frequent event types in *tolstoy*.



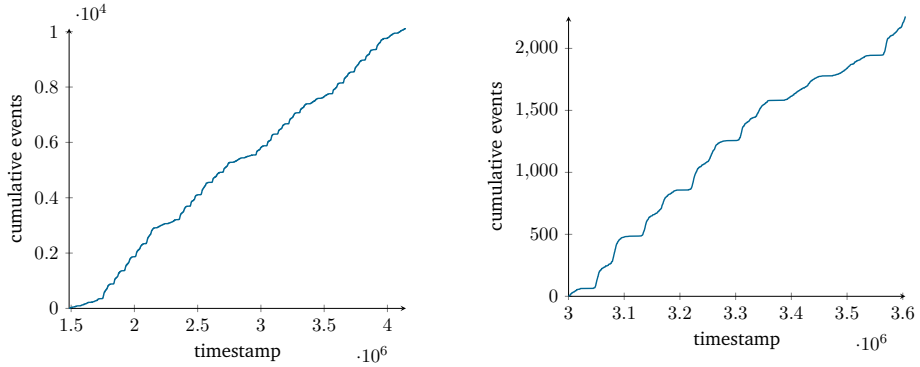
(c) The event types in *trains*.

Figure 4.1.: The frequency of event types in the datasets.

experiment would run the Cartesian product of all these parameters, within time and memory constraints.

For each configuration of episode class, frequency measure, and window width, the frequency threshold is lowered exponentially, using a multiplier $\in (0, 1)$. For example, with a multiplier of 0.9, the next threshold is always 10% smaller than the last. If memory runs out or the timeout is exceeded before reaching the lower bound, all lower frequency thresholds for that configuration are skipped, as they will take at least as much time and memory as the current threshold.

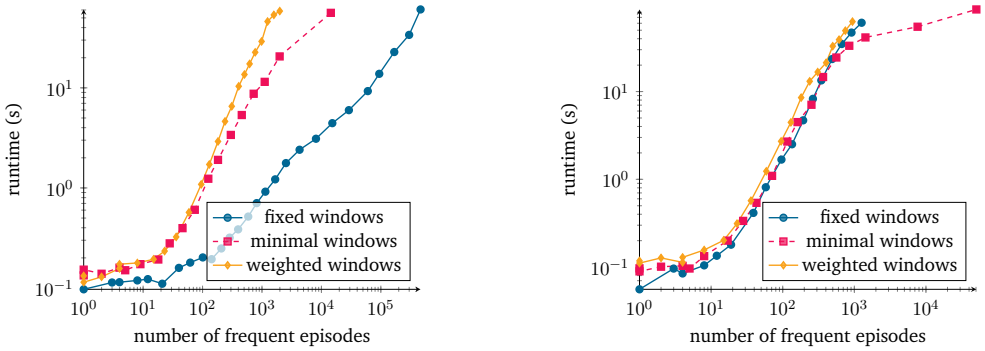
All performance experiments were run on the same machine; the full specifications of which can be found online[5].



(a) The whole sequence.

(b) A period of one week.

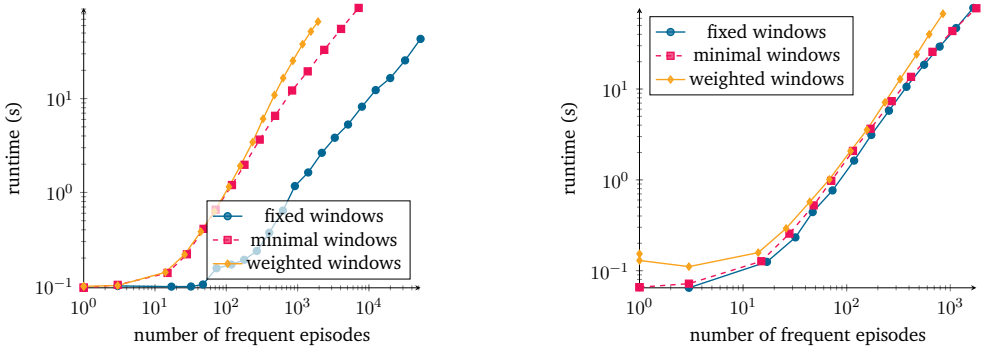
Figure 4.2.: The cumulative distribution of events in the sparse event sequence *trains*.



(a) parallel episodes

(b) serial episodes

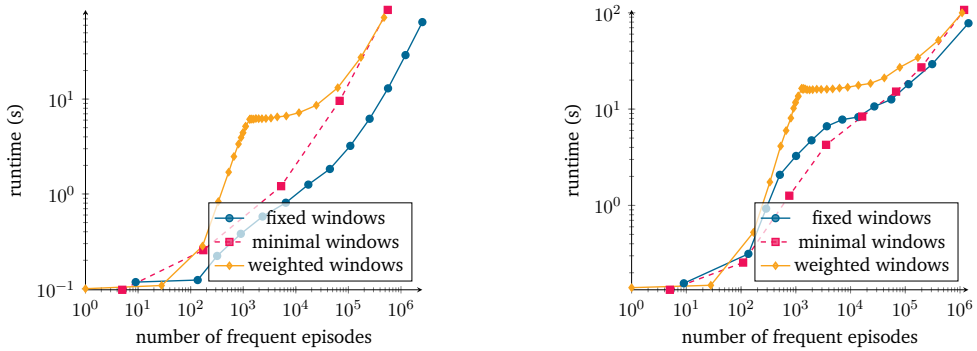
Figure 4.3.: Runtimes for finding episodes in dataset *abstract* using a window width of 15.



(a) parallel episodes

(b) serial episodes

Figure 4.4.: Runtimes for finding episodes in dataset *tolstoy* using a window width of 15.



(a) parallel episodes (b) serial episodes

Figure 4.5.: Runtimes for finding episodes in dataset *trains* using a window width of 900.

4.3.1. Episodes

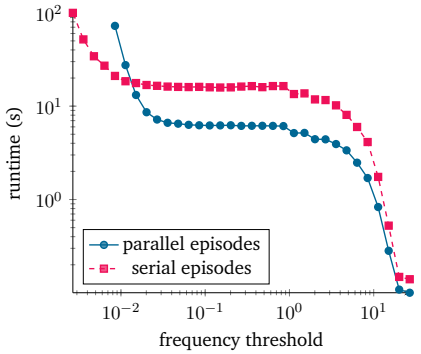
We would like to compare the efficiency for the different frequency measures across a range of frequency thresholds. However we should not compare the runtimes of the frequency measures based on the frequency thresholds directly, since the values for each of the measures are semantically different. Instead we can compare the runtimes as a function of the number of frequent episodes produced.

The graph in Figure 4.3 shows runtimes for mining episodes from *abstract*. We see that, for parallel episodes, the fixed-window frequency takes significantly less time to generate a given amount of episodes than the other measures. The disjoint-window frequency and the weighted-window frequency are very close to each other until approximately one thousand episodes are found; then they start to diverge, and the the weighted-window frequency takes more time than the other two.

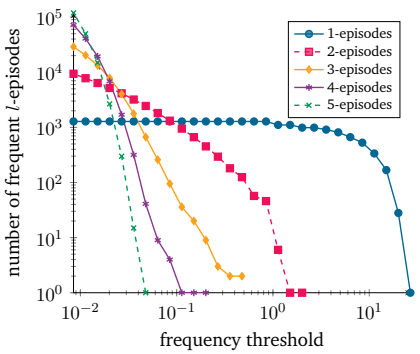
For serial episodes, the fixed-window frequency has much less of an advantage. In fact, the disjoint-window frequency seems to overtake the fixed-window frequency for large amounts of episodes produced. This could be partly due to the fact that the data pass algorithm that finds minimal windows of serial episodes is slightly simpler than the algorithm that determines the fixed-window frequency of serial episodes. Again the weighted-window frequency is slower, but the difference is smaller this time. Also note that for all measures, runtimes for serial episodes were generally higher — since the experiments were time-constrained, the graph stops around the order of 10^5 for parallel episodes, and around 10^3 for serial episodes.

Figure 4.4 shows the results for *tolstoy*. Though *tolstoy* is approximately twice as long as *abstract*, the progressions are very similar. Given these results, we think that it is likely

[5]The specifications can be found at `https://support.apple.com/kb/sp623`. 2.7 GHz model; memory manually upgraded to 12 GB. Running macOS 10.13.5. All C++ code compiled with clang-900.0.39.2 from LLVM 9.0. All Java code run with Java SE 1.8.



(a) Runtime as a function of the frequency threshold.



(b) Number of frequent l-episodes as a function of the frequency threshold. Not all episode sizes shown, and only parallel episodes.

Figure 4.6.: Some more plots as a subject of investigation for the curious runtimes for the weighted windows in experiment in Figure 4.5.

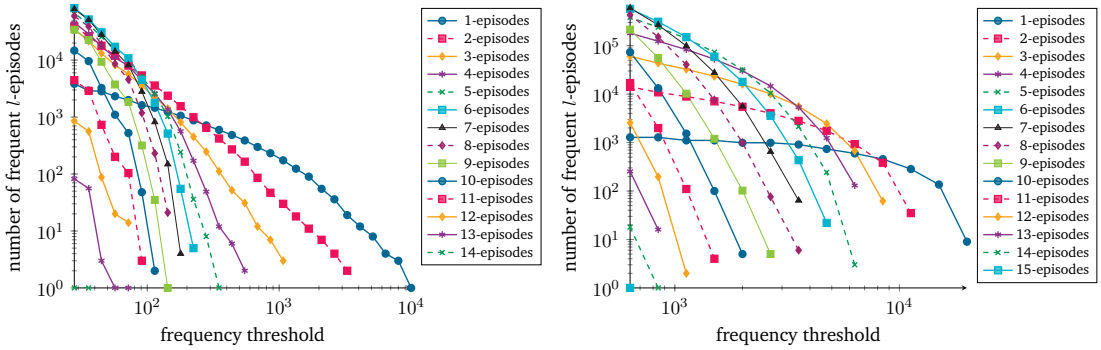
that most texts with a typical frequency distribution of words (as in Figure 4.1a and Figure 4.1b) will show a similar progression.

The results for the sparse sequence *trains* are a bit different (Figure 4.5). For parallel episodes, the fixed-window frequency is faster, like before. Curiously, the time consumption for the weighted-window frequency increases steadily, until around 10^3 episodes or so, when it plateaus, and eventually matches the disjoint-window frequency. For serial episodes, the weighted-window frequency shows a similar plateau.

Figure 4.6 provides some insight. If we plot the runtime as a function of the frequency threshold (Figure 4.6a), we see that the runtime plateaus around a frequency threshold of 1. On the right hand plot (Figure 4.6b), we see that for thresholds less than 1, the number of 1-episodes doesn't increase. That's because all possible 1-episodes (the entire alphabet) are now frequent. This means that the number of 2-candidates does not increase anymore. Further lowering the threshold, more of those 2-episodes start becoming frequent at no extra cost. Only after a number of 2-episodes starts becoming frequent, the runtime starts increasing significantly again; when a significant amount of greater-than-2-episodes starts becoming frequent.

For the other frequency measures, we don't see such a kink in the graph, and that's because those never reached the point where the full alphabet becomes frequent, since they are more forgiving towards larger episodes than the weighted-window frequency.

Note that significantly more episodes are found in the *trains* sequence — up to the order of 10^6. This is most likely due to the fact that *trains* is considerably smaller than *abstract* and *tolstoy*, in both alphabet size and number of events. Perhaps, if given more time, the progression would look similar for *abstract* and *tolstoy*.



(a) *abstract* (b) *trains*

Figure 4.7.: Number of parallel episodes found in *abstract* and *trains* for the fixed-window frequency across different thresholds, episodes grouped by size.

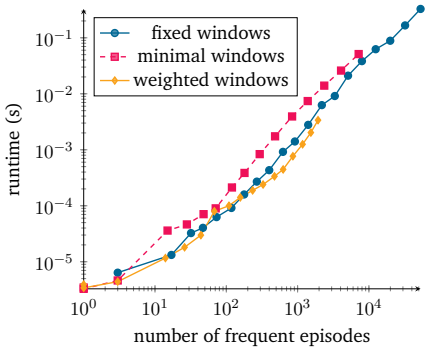
Besides runtime comparisons, it can also be useful to know how many episodes of each size are produced. In Figure 4.7 we see that for low thresholds lots of larger episodes become frequent. So if one is interested in large episodes, mining with a low threshold will help a lot. For those low thresholds, 6-episodes and 7-episodes are found in the largest quantities for both datasets. We find up to 15-episodes.

Plots for the other kinds of episodes and frequency measures showed similar results, though there were fewer large episodes. In part that can be explained by the fact that all experiments were run under the same time constraints: as we observed previously, the implementation that mines parallel episodes by the fixed-window frequency usually produces larger amounts of episodes within a given amount of time. It also depends on the characteristics inherent to the frequency measures, for instance: the weighted-window frequency is biased towards small minimal windows, so it will find more small episodes.

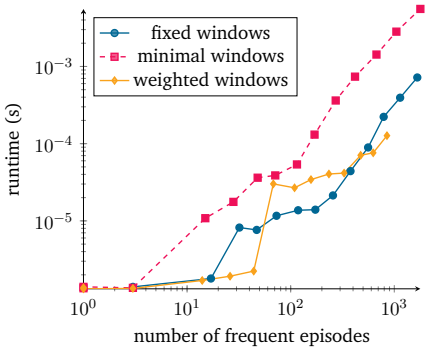
4.3.2. Association rules

Besides episodes, we would also like to have a sense of the runtimes for generating association rules. All of the results shown in this section were generated during the experiments from Section 4.3.1, so the same parameter configurations apply.

For a given set of episodes, the choice of the confidence threshold mostly depends on space requirements — time less so. That is because the number of association rules that needs to be considered does not depend on the chosen threshold, and the only time cost for choosing a low confidence threshold is in the fact that more episodes will need to be stored. The space cost, however, can increase dramatically. Experimentally, we found that it was feasible to store all association rules generated from the episodes found frequent for *tolstoy*, but not those for *trains*. So, for *tolstoy* we ran with a confidence threshold of 0 — further filtering could be applied in post-processing if desired — and

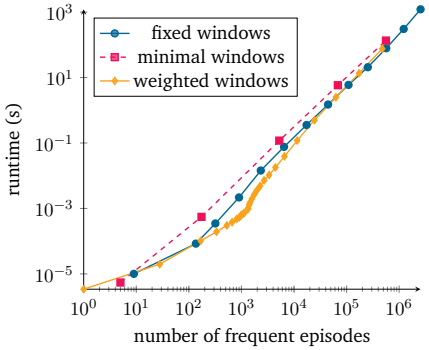


(a) rules consisting of parallel episodes

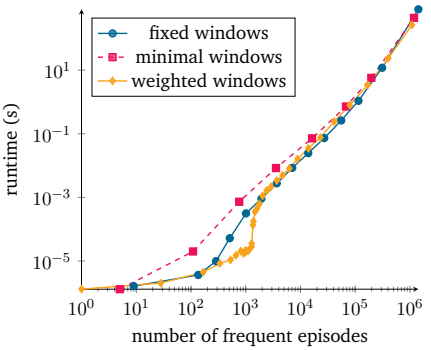


(b) rules consisting of serial episodes

Figure 4.8.: Runtimes for finding association rules using episodes generated from *tolstoy* as a function of the number of frequent episodes.



(a) rules consisting of parallel episodes



(b) rules consisting of serial episodes

Figure 4.9.: Runtimes for finding association rules using episodes generated from *trains* as a function of the number of frequent episodes.

for *trains* a threshold of 0.9 turned out to be reasonable in terms of storage.

As we saw in Algorithm 10, for each frequent episode β, all association rules $\alpha \Rightarrow \beta$ with $\alpha \subset \beta$ are considered. Assuming β has an injective lab-function (an event type appears at most once), the number of subepisodes of an episode is exponential in the size of the episode; and so the same holds for the number of association rules to be considered.

Figure 4.8 shows runtimes for generating association rules from *tolstoy* using different episode classes, as a function of the number of frequent episodes. Despite the concerns we expressed in the previous paragraph, the runtime seems to be polynomial — indicating that the number of frequent episodes remains the most important factor for these experiments.

Finding the association rules from parallel episodes in *trains* using the fixed-window frequency (Figure 4.9a) started to take significant amounts of time for the lowest frequency thresholds (largest quantities of episodes produced). However, given that the graphs in the log–log plot are mostly linear, we would judge the complexity to be polynomial.

On the contrary, for association rules from serial episodes in *trains*, the graph does curve upwards somewhat towards the end (Figure 4.9a), possibly due to the exponential relation described earlier. From the experiment of the Figure 4.7b the amounts of larger serial episodes seemed to rise more steeply than those for parallel episodes, which might contribute to these results.

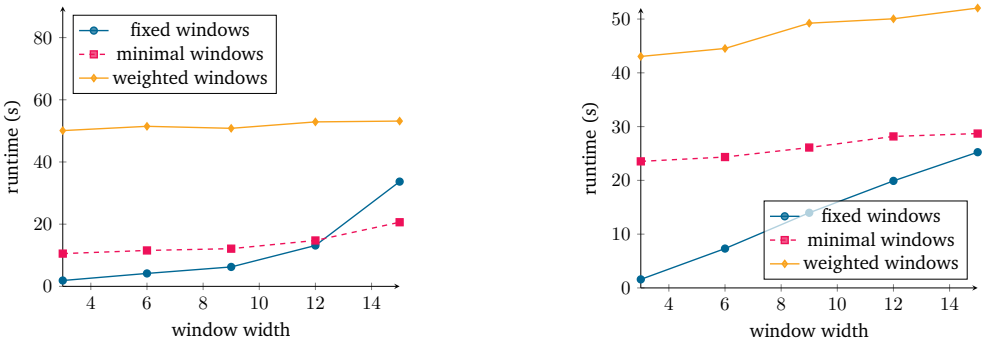
We mentioned earlier that computing the fixed-window confidence is trivial, given the frequency of the episodes; and yet for some of the experiments here, finding the confident association rules by the fixed-window confidence took longer than those for the weighted-window confidence. This can be explained by the overhead of storing the rules that turned out to be confident, requiring more time if there are more confident rules.

4.3.3. Window width

The window width places an upper bound on the maximal distance between two events that are part of an occurrence of an episode. Consequently this also limits the number of candidate patterns. We know that the number of candidate episodes is exponential in the number of nodes. So we would like to have a sense of how the window width affects the runtimes.

For each combination of episode class and frequency measure, we decided on a threshold for which the algorithm finishes in a reasonable amount of time for a given window width. We used a window width of 15 for *abstract* and 900 for *trains*. Then we varied the window width — 3, 6, 9, 12, 15 for *abstract* and 180, 360, 540, 720, 900 for *trains*. Because we selected the thresholds in this way, the absolute values are of no importance; only the progression is.

The results for *abstract* are shown in Figure 4.10. For parallel episodes, the fixed-window frequency seems to be affected the most. The weighted-window frequency doesn't seem



(a) parallel episodes

(b) serial episodes

Figure 4.10.: Runtimes for mining episodes from *abstract*, varying the window width.

to be affected much at all. This is likely due to the fact that it naturally produces fewer large episodes, given its preference for small minimal windows. As we mentioned before, in Section 2.3.3, weighted-windows frequency values stabilize as the window width grows.

Figure 4.11 shows the results for *trains*. Interestingly, the runtimes for parallel episodes and fixed-window frequency remain quite stable until a window width of 720, after which they jump upwards significantly. The number of frequent episodes grew over eight times as large. The particular characteristics of the *trains* sequence may contribute, although we have no plausible explanation at the moment. In the same interval, the runtimes for the disjoint-window frequency increased quite a bit as well, whereas it progresses in a more stable manner in the other graphs.

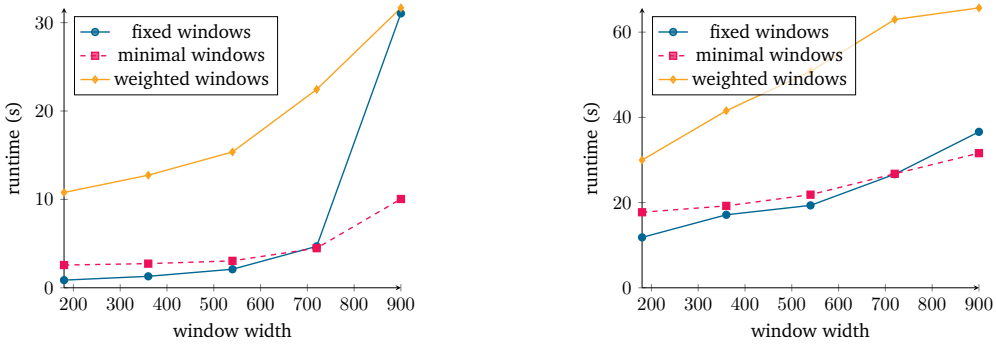
The general tendency for the fixed-window frequency to be more affected than the other measures might be partly attributable to the amplification effect described in Section 2.3.1.

While these results give an indication of the influence of the window width on the runtime, they are not conclusive, because since the experiments are time-consuming, they don't cover very large window widths, nor very low thresholds.

For reference, Table 4.3 shows the frequency thresholds that were used.

4.3.4. Sequence length

In sequential pattern mining, sequences are expected to be very long. Therefore we would like to know how the length of the sequence affects the performance. We measured the runtime for mining episodes from different-sized prefixes of the *tolstoy* sequence for all episode classes and frequency measures, using a window width of 15. The results for parallel episodes can be seen in Figure 4.12. The results for serial episodes were similar.



(a) parallel episodes (b) serial episodes

Figure 4.11.: Runtimes for mining episodes from *trains*, varying the window width.

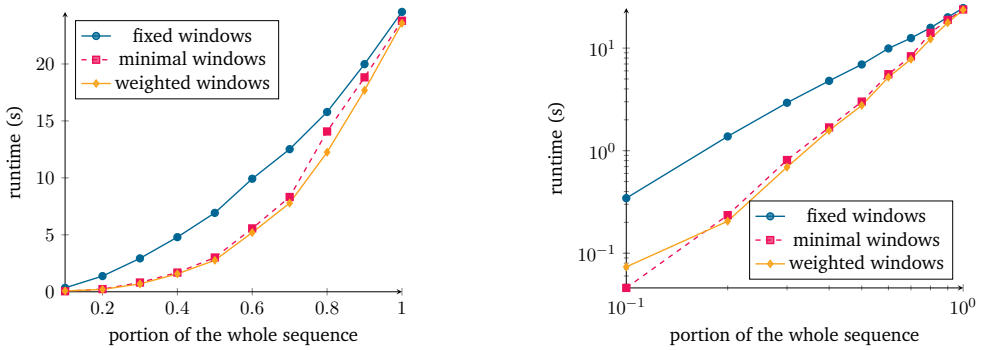
episode class	frequency measure	frequency threshold
parallel	fixed-window frequency	29
parallel	disjoint-window frequency	23
parallel	weighted-window frequency	4.11
serial	fixed-window frequency	246
serial	disjoint-window frequency	23
serial	weighted-window frequency	14.5

(a) *abstract*

episode class	frequency measure	frequency threshold
parallel	fixed-window frequency	843
parallel	disjoint-window frequency	9
parallel	weighted-window frequency	0.0113
serial	fixed-window frequency	632
serial	disjoint-window frequency	3
serial	weighted-window frequency	0.00358

(b) *trains*

Table 4.3.: The frequency thresholds used for the results of Figure 4.10 and Figure 4.11.



(a) linear scale

(b) log–log scale

Figure 4.12.: Runtimes for different portions of *tolstoy* using a fixed frequency threshold for each frequency measure.

episode class	frequency measure	frequency threshold
parallel	fixed-window frequency	74
parallel	disjoint-window frequency	55
parallel	weighted-window frequency	56.3
serial	fixed-window frequency	1000
serial	disjoint-window frequency	74
serial	weighted-window frequency	75.1

Table 4.4.: The frequency thresholds used for the results of Figure 4.12.

As in the previous section, frequency thresholds were chosen based on the runtime. So again, when studying the graphs, only the progressions are important.

The general progression is fairly similar for all of the measures, although the minimal-windows-based measures rise quicker than the fixed-window frequency. While unfortunately the complexity is worse than linear (Figure 4.12a), judging by the mostly straight lines in the log–log plot (Figure 4.12b) it does seem to be polynomial.

For reference, Table 4.4 shows the frequency thresholds that were used.

4.3.5. Performance comparison with closed episode miner

In section 4.1 we explained that our implementation computes a subset of the episodes that the closed episode miner mines, if the closed episode miner is configured not to restrict its output to closed episodes. Besides comparing the output of both implementations to ensure the correctness, we can also compare the runtimes.

We ran both implementations on the *tolstoy* sequence with a window width of 15. Thresholds were chosen based on a compromise between amount of output and time consumption. We should remark again that the closed episode miner finds parallel and

episode class	frequency measure	frequency threshold	runtime (s) ours	runtime (s) Closepi
parallel	fixed-window fr.	15	296	—
serial	fixed-window fr.	150	216	8.67
parallel	disjoint-window fr.	7	417	24.8
serial	disjoint-window fr.	10	417	15.0
parallel	weighted-window fr.	7	127	9.59
serial	weighted-window fr.	10	300	6.65

Table 4.5.: Runtimes for mining episodes with our implementation (*ours*) and the closed episode miner (*Closepi*).

serial episodes at the same time, as well as general episodes. So some of the configurations of the closed episode miner were equivalent, apart from the frequency threshold.

Table 4.5 shows the results. For one of the experiments, the closed episode miner ran out of memory (marked with a dash), when it used over 10 GB of memory after a couple of minutes. The other results are not a great look for our implementation. Even though we are at an advantage in the sense that we only mine one class of episodes at a time, the closed episode miner runs significantly faster.

We think that there must be opportunities left for improving the performance of our implementation. Although we tried to do our best, we are not experts at optimizing code for speed. That being said, we think that in most cases, our implementation produces adequate amounts of output in a reasonable time.

4.4. Quality

For the qualitative analysis of the output of our algorithm, we study the top-ranked results and compare across the different interestingness measures. After an introduction to cohesion-based measures, where the main focus shifts away from frequency, we run experiments using algorithms which implement those measures, and compare them to the results for our measures.

4.4.1. Analysis of episodes mined from *tolstoy* sequence

We ran the algorithm using a window width of 15, like we did before for the textual datasets. For reference, the frequency thresholds are we used are in Table 4.5.

As we would expect, if we rank the output by frequency, the top contains mostly just episodes of size 1 (Table 4.6). It does give us some information about the text, though not much more than when we simply count the occurrences of all words.

- We learn of many characters' names, as obviously those are mentioned often, but

#	fixed-window fr.	disjoint-window fr.	weighted-window fr.
1	{levin} (20913)	{levin} (1629)	{levin} (1629)
2	{vronski} (11165)	{vronski} (865)	{vronski} (865)
3	{anna} (10699)	{anna} (823)	{anna} (823)
4	{thought} (8994)	{kitti} (672)	{kitti} (672)
5	{time} (8948)	{thought} (663)	{thought} (663)
6	{kitti} (8826)	{time} (651)	{time} (651)
7	{hand} (8645)	{hand} (651)	{hand} (651)
8	{alexei} (8619)	{smile} (632)	{smile} (632)
9	{smile} (8549)	{alexei} (632)	{alexei} (632)
10	{face} (8315)	{face} (598)	{face} (598)
11	{ey} (8062)	{love} (595)	{love} (595)
12	{alexandrovitch} (7842)	{alexandrovitch} (571)	{alexandrovitch} (571)
13	{felt} (7753)	{alexei, alexandrovitch} (571)	{ey} (570)
14	{man} (7751)	{ey} (570)	{man} (565)
15	{feel} (7596)	{man} (565)	{feel} (561)

Table 4.6.: The top 15 parallel episodes found by our algorithm, with $\rho = 15$, and for the three frequency measures.

since we see mostly 1-episodes we can't connect many characters' first name and surname.

- Looking at the column for the disjoint-window frequency, we see that two names are frequently mentioned together: {alexei, alexandrovitch}. From this information it is likely that a character named *Alexei Alexandrovitch* appears often in the book. (This is indeed the case.) Moreover, we see that {alexei, alexandrovitch} is just as frequent as subepisode {alexandrovitch}. So wherever the last name *Alexandrovitch* is mentioned, the first name *Alexei* is mentioned nearby (within at most 15 words).
- Common words like *thought*, *smile*, *face*, *love*, *eye* (stemmed to *ey*), *feel* can give some indication of genre. At least we can agree that the sequence does not represent a research paper in computer science.

From studying Table 4.6 it is clear that simply ranking episodes by frequency is not a good strategy for getting the most out of our algorithm. We should at least filter out the 1-episodes, as those don't give any more information than counting the support of each word that appears in the text. Table 4.7 and Table 4.8 show the rankings of greater-than-1-episodes, for parallel and serial episodes respectively. Now we learn some more about the text:

- We find full names — Alexei Alexandrovitch, Stepan Arkadyevitch, Lidia Ivanovna, Darya Alexandrovna, Agafea Mihalovna — all are characters' full names. With serial episodes, we find their order as well — *alexei* usually precedes *alexandrovitch* closely, so alexei $\rightarrow$ alexandrovitch is rated more highly than alexandrovitch $\rightarrow$ alexei.

#	fixed-window fr.	disjoint-window fr.	weighted-window fr.
1	{alexei, alexandrovitch} (7416)	{alexei, alexandrovitch} (571)	{alexei, alexandrovitch} (286)
2	{stepan, arkadyevitch} (7117)	{stepan, arkadyevitch} (547)	{stepan, arkadyevitch} (274)
3	{sergei, ivanovitch} (3763)	{levin, levin} (395)	{sergei, ivanovitch} (146)
4	{levin, levin} (3348)	{sergei, ivanovitch} (291)	{darya, alexandrovna} (102)
5	{darya, alexandrovna} (2739)	{darya, alexandrovna} (205)	{levin, levin} (61.6)
6	{levin, kitti} (2039)	{levin, kitti} (202)	{lidia, ivanovna} (54)
7	{anna, vronski} (1942)	{stepan, levin} (199)	{anna, vronski} (45.1)
8	{arkadyevitch, levin} (1896)	{arkadyevitch, levin} (197)	{smile, levin} (41.3)
9	{stepan, levin} (1887)	{stepan, arkadyevitch, levin} (195)	{levin, kitti} (41.1)
10	{stepan, arkadyevitch, levin} (1784)	{vronski, vronski} (191)	{room, draw} (40.6)
11	{smile, levin} (1778)	{anna, vronski} (180)	{countess, lidia} (39.4)
12	{vronski, vronski} (1722)	{smile, levin} (171)	{thought, levin} (39.3)
13	{time, levin} (1597)	{anna, anna} (170)	{love, love} (38.2)
14	{brother, levin} (1558)	{time, levin} (159)	{arkadyevitch, levin} (37.2)
15	{anna, anna} (1531)	{good, levin} (153)	{agafea, mihalovna} (37)

Table 4.7.: The top 15 parallel episodes found by our algorithm, excluding 1-episodes, with $\rho = 15$, and for the three frequency measures.

- We also find the most important couples, since naturally their names are often mentioned close to each other — Kitty and Levin, Anna and Vronski.
- The weighted-window frequency finds $\{\text{draw}, \text{room}\}$ and draw $\rightarrow$ room at positions 10 and 7, respectively. The fixed-window confidence ranked $\{\text{draw}, \text{room}\}$ at position 30, and the disjoint-window frequency ranked it at position 90. The weighted-window frequency ranks it highly because of the noun *drawing room*. Those two words often follow each other directly, making for many minimal windows of great weight.

Removing the 1-episodes proved to be useful, since they dominated the top 15. This is a typical phenomenon of monotonically decreasing frequency measures — they put larger episodes at a disadvantage, since an episode is never more frequent than any of its subepisodes. $(l + 1)$-episodes generally score lower than l-episodes, and as a consequence, larger episodes tend to be ranked lower than smaller ones. Some measures prefer small minimal windows over large ones. And though, as we stated before, small occurrences are more likely to be significant, larger episodes naturally require a larger time span.

#	fixed-window frequency	disjoint-window fr.	weighted-window fr.
1	alexei → alexandrovitch (7401)	alexei → alexandrovitch (571)	alexei → alexandrovitch (286)
2	stepan → arkadyevitch (7106)	stepan → arkadyevitch (547)	stepan → arkadyevitch (274)
3	sergei → ivanovitch (3758)	levin → levin (395)	sergei → ivanovitch (146)
4	levin → levin (3348)	sergei → ivanovitch (291)	darya → alexandrovna (102)
5	darya → alexandrovna (2734)	darya → alexandrovna (205)	levin → levin (61.6)
6	vronski → vronski (1722)	vronski → vronski (191)	lidia → ivanovna (54)
7	anna → anna (1531)	anna → anna (170)	draw → room (40.5)
8	lidia → ivanovna (1438)	alexandrovitch → alexei (145)	countess → lidia (39.1)
9	love → love (1411)	kitti → kitti (144)	love → love (38.2)
10	arkadyevitch → levin (1245)	love → love (140)	agafea → mihalovna (37)
11	kitti → kitti (1200)	arkadyevitch → levin (137)	levin → felt (32)
12	levin → kitti (1160)	levin → kitti (136)	vronski → vronski (31.3)
13	vronski → anna (1137)	stepan → levin (132)	good → humor (29.1)
14	levin → felt (1130)	stepan → arkadyevitch → levin (132)	anna → arkadyevna (28.5)
15	draw → room (1126)	kitti → levin (131)	vronski → anna (28.1)

Table 4.8.: The top 15 serial episodes found by our algorithm, excluding 1-episodes, with $\rho = 15$, and for the three frequency measures.

Nevertheless, there are ways to surface larger episodes present in the output, for example:

- As we did in this section, we can filter out smaller episodes.
- Grouping episodes by size before ranking them will highlight larger episodes even more.
- One could also boost the score of larger episodes in some way, so that they have more value in the rankings.

4.4.2. Analysis of association rules mined from *tolstoy* sequence

The association rules that we show here, were generated from the episodes in Section 4.4.1. Because the rules require lots of space, we refer to Appendix A for the top 15 of each of the episode classes and frequency measures.

Table 4.9 shows the top 5 association rules consisting of parallel episodes using the fixed-window confidence.

For the fixed-window confidence, there are a lot of rules of confidence 1. In fact, in this experiment, there were 129 association rules consisting of serial episodes that had a confidence of 1, and $114\,230$ such rules for parallel episodes (out of $131\,464$ rules with a confidence threshold of 95%). For minimal windows, the situation wasn't much better: $2\,218$ association rules had a confidence of 1.

This poses a problem for top-k techniques. The top 5 lists we show here are just a small sample of equally-confident association rules — these rules happened to be on top.

In the top 5 for the fixed-window confidence, all of the rules have the same tail:

$$\{\text{stepan}, \text{arkadyevitch}, \text{ey}, \text{love}, \text{youth}, \text{gallant}, \text{steed}, \text{token}, \text{declaim}\}$$

In fact, the output contains 63 of these rules, all with a confidence of 1. This episode originates from a quote that is mentioned twice in the book. A non-stemmed quotation:

> "I know a gallant steed by tokens sure,
> And by his eyes I know a youth in love,"
> declaimed Stepan Arkadyevitch.

The heads of these association rules all include the two event types that are outermost in the occurrence, *gallant* and *arkadyevitch*, and leave out different words from the middle of the quote: *stepan*, *ey*, *love*, and so on. All of those subepisosdes are covered by the same fixed windows as the tail, resulting in a fixed-window confidence of 1. Needless to say, including all of these rules is not very informative. Here, restricting the output to closed association rules would be very helpful. The association rule

$$\{\text{arkadyevitch}, \text{gallant}, \text{declaim}\} \Rightarrow$$
$$\{\text{stepan}, \text{arkadyevitch}, \text{ey}, \text{love}, \text{youth}, \text{gallant}, \text{steed}, \text{token}, \text{declaim}\}$$

#	association rule (fixed-window confidence)
1	{arkadyevitch, ey, love, youth, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
2	{stepan, arkadyevitch, love, youth, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
3	{arkadyevitch, love, youth, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
4	{arkadyevitch, ey, youth, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
5	{stepan, arkadyevitch, youth, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)

Table 4.9.: Top 5 parallel association rules, by the fixed-window confidence.

has a minimal head in order to have a confidence of 1, and the heads of all other 62 rules are superepisodes of this minimal head. So the other 62 rules provide no additional information.

The top association rules for serial episodes seemed to suffer less from this (see Appendix A.2), though that may be because the relevant episodes were mined with a higher frequency threshold, such that fewer of these large episodes with many subepisodes occur.

Table 4.10 shows the top 5 association rules consisting of parallel episodes using the weighted-window confidence, and Table 4.11 shows the same for serial episodes.

From these top fives we mostly learn that for certain characters, the full names are always used.

- Parallel #1: Whenever two occurrences of surname *arkadyevitch* are close together, the character's first name *stepan* can be found nearby — or more precisely, in between the two occurrences, since a weighted-window confidence of 1 implies that the minimal windows of head and tail are equal.
- Parallel and serial #2: The same for *Alexei Alexandrovitch*.
- Parallel #3 is analogous #1, but from first name to surname instead.
- Serial #3 and #4 give similar information, when another another word follows the name (*levin*, *smile*).
- Serial #1: sergei → levin leads consistenly to sergei → ivanovitch → levin. Sergey Ivanovitch Levin is one character's full name, so it makes sense that whenever we find the outermost parts, the inner part is in between.

For the weighted-window confidence, there were far fewer association rules in the top 15 with the same confidence value, which we would expect.

Before we conclude, we should note that this comparison between the different confidence measures was not entirely fair — the sets of episodes upon which we built the as-

#	association rule (weighted-window confidence)
1	{arkadyevitch, arkadyevitch} ⇒ {stepan, arkadyevitch, arkadyevitch} (1)
2	{alexandrovitch, alexandrovitch} ⇒ {alexei, alexandrovitch, alexandrovitch} (1)
3	{stepan, stepan} ⇒ {stepan, stepan, arkadyevitch} (1)
4	{countess, ivanovna} ⇒ {countess, lidia, ivanovna} (0.943)
5	{arkadyevitch, alexei} ⇒ {stepan, arkadyevitch, alexei} (0.937)

Table 4.10.: Top 5 parallel association rules, by the weighted-window confidence.

#	association rule (weighted-window confidence)
1	sergei → levin ⇒ sergei → ivanovitch → levin (1)
2	alexandrovitch → alexandrovitch ⇒ alexandrovitch → alexei → alexandrovitch (1)
3	stepan → levin ⇒ stepan → arkadyevitch → levin (1)
4	stepan → smile ⇒ stepan → arkadyevitch → smile (1)
5	alexei → alexandrovitch → alexandrovitch ⇒ alexei → alexandrovitch → alexei → alexandrovitch (1)

Table 4.11.: Top 5 serial association rules, by the weighted-window confidence.

sociation rules depended both on the inherent characteristics of the respective frequency measures, and on computational constraints of the algorithms implementing them. For instance, we saw that the weighted-window frequency favoured smaller episodes more so than the disjoint-window frequency, since it values small minimal windows higher than larger ones. Also, during the performance experiments it became clear that the fixed-window frequency was able to find more episodes in the same amount of time.

If we wanted to make a better qualitative comparison between the different confidence measures, perhaps we should decide on a set of episodes, and generate confidence rules based on those. While not hard to realize, this would require some effort to modify the implementation, and could be subject of future work. Because of this, it is hard to make definitive statements about how the confidence measures compare in quality.

We saw that many association rules provided very similar information, so we think that some redundancy elimination techniques are required to increase the usefulness of a ranking of association rules by confidence.

4.4.3. Comparison with non-frequency-based methods

So far, in this thesis, we have only studied frequency-based interestingness measures. In this section, we'll look into a different interpretation of interestingness: *cohesion*. The cohesiveness of an episode expresses how closely the events of its occurrences are to each other. An episode is deemed cohesive in regard to a sequence if the events that constitute occurrences are close to each other; if, in other words, its minimal windows are generally small.

All frequency measures we have studied so far had a cohesion angle as well. This is

natural in the context of a long event sequence, where events that are far apart, are unlikely to have anything to do with each other. The width of the sliding window was not only a means to make the algorithms computationally feasible, it also placed a cutoff beyond which events were deemed uncorrelated. For the disjoint-window frequency, this cutoff was quite harsh — all minimal windows within a certain width ρ were considered equally valuable, while any windows larger than ρ were judged uncorrelated.

The weighted-window frequency takes into account the width of the minimal windows more carefully — it assigns a weight to each minimal window (the inverse of its width) and sums the weights. So, the weighted-window frequency places importance on the cohesiveness of patterns, but it remains a measure based on frequency.

With cohesion-based interestingness measures, the main focus shifts towards the cohesion of patterns.

In [3], Cule et al. define the cohesion of an episode α as

$$C(\alpha) = \frac{|\alpha|}{\overline{W}(\alpha)}$$

where $\overline{W}(\alpha)$ is the average width of the minimal windows of α. Further details on the exact definition can be found in [3].

This definition makes an effort not to disadvantage larger patterns — as anti-monotonic frequency measures inherently do for computational reasons — by defining the cohesiveness of an episode to be proportional to the size of the episode. The paper describes a method for mining episodes according to this non-monotonic measure, using other techniques of pruning the search space, which are beyond the scope of this thesis.

Also, to prevent including patterns that are too infrequent to bear any kind of significance, [3] places a frequency constraint as well: the event types that make up a cohesive episode should occur often enough by themselves, given some support threshold. Event types that don't have enough support won't appear in any pattern, but there are no frequency requirements for episodes as a whole.

The mining algorithm from [3], called FCI_{SEQ}, mines parallel episodes with an injective lab-function, meaning that each event type appears at most once. The algorithm takes the following parameters:

- The minimal support that an event type must have in order to be considered at all.
- The maximal size of patterns to generate.
- The minimal cohesion of the patterns.

In [3], the cohesion of a pattern is defined using the mean width of the minimal windows. The mean has a number of downsides, though, one of which is that the mean of a distribution is unstable if there are outliers. So if one window is significantly larger or smaller than the others, the cohesion may be greatly affected. In [5], Feremans et al. propose a quantile-based approach. Here, the quantile-based cohesion is defined as the

#	mean-based cohesion	quantile-based cohesion (cohesion, support)
1	{agafea, mihalovna} (1)	stepan → arkadyevitch (0.998, 1096)
2	{char, banc} (1)	alexei → alexandrovitch (0.95, 1203)
3	{pinc, nez} (1)	sergei → ivanovitch (0.965, 603)
4	{bell, soeur} (1)	agafea → mihalovna (1, 148)
5	{stepan, arkadyevitch} (0.915)	darya → alexandrovna (0.967, 424)
6	{nativ, tribe} (0.523)	char → banc (1, 14)
7	{lizaveta, petrovna} (0.408)	pinc → nez (1, 12)
8	{ivanovna, lidia} (0.137)	bell → soeur (1, 10)
9	{alexandrovitch, alexei} (0.05)	lidia → ivanovna (0.977, 221)
10	{sergei, ivanovitch} (0.0428)	lizaveta → petrovna (0.98, 49)
11	{darya, alexandrovna} (0.0363)	nativ → tribe (0.966, 29)
12	{bezzubov, landau} (0.0351)	mashkin → upland (0.833, 12)
13	{gladiat, frou} (0.0326)	grand → duchess (0.714, 14)
14	{partnership, ryezunov} (0.0303)	liza → merkalova (0.706, 34)
15	{bridal, lectern} (0.0281)	marya → nikolaevna (0.673, 98)

Table 4.12.: The top 15 patterns mined from *tolstoy* using cohesion (FCI_{SEQ}, minimum support 5, maximal size 5) and quantile-based cohesion (QCSP, minimum support 5, maximal size 5, minimal-window width threshold 2).

percentage of minimal occurrences that are smaller than a given threshold. The threshold is proportional to the size of the episode, again in an effort not to disadvantage larger patterns.

Their mining method QCSP mines serial episodes, and takes the following parameters:

- The minimal support that an event type must have in order to be considered at all.
- The maximal size of patterns to generate.
- A minimal-window width threshold (as mentioned above, which gets multiplied by the size of each episode being considered).
- k. The algorithm reports only the k patterns which are the most cohesive.

Again we refer to the paper for more details [5].

For both algorithms we ran with a minimum support of 5, and a maximal pattern size of 5. We ran FCI_{SEQ} with a minimal cohesion of 0.014 (a lower threshold started to take significantly more time) and QCSP with a window-width threshold of 3 and $k = 150$.

Table 4.12 shows the 15 most interesting episodes by the two different definitions of cohesion. The implementation of the quantile-based cohesion sorts episodes according to a linear combination of the cohesion and the support; therefore we also show the support for QCSP results The support of an episode is defined as the sum of supports of the constituent event types.

Some of these patterns are both frequent and cohesive — we see some of the characters' names reappear. But we also see many patterns that we haven't seen before, including the names of some characters which feature less prominently in the book, such as

Lizaveta Petrovna.

The algorithms find some maximally cohesive patterns consisting of multiple-word nouns, for which the constituent words don't appear on their own anywhere else, such as *char-à-banc* (7 occurrences), *pince-nez* (6 occurrences), and *belle-soeur* (5 occurrences).

These are all quite infrequent, and we expect that the frequency-based measures do not value them highly. Indeed: for the disjoint-window frequency, $\{\text{char}, \text{banc}\}$ appears at position $30\,203$ in the rankings.

Interestingly, there are no 3-episodes in the top 15, despite the efforts not to disadvantage larger episodes. However, for the mean-based cohesion, the first 3-episode appears at position 17, and the first 4-episode at position 21. Further down the rankings, many larger episodes start to appear.

The mean-based cohesion values start to decrease very quickly. The phrase *Native Tribes* occurs 14 times in the book, and the word *tribe* occurs only once on its own. Intuitively, given that one lone occurrence, we would still consider the episode to be quite cohesive — if that occurrence wasn't there, the cohesion of $\{\text{nativ}, \text{tribe}\}$ would equal 1. Yet, $\{\text{nativ}, \text{tribe}\}$ has a mean-based cohesion of 0.523. It seems like the mean-based cohesion punishes any unrelated occurrences quite harshly; likely because of the instability to outliers that the mean suffers, as we mentioned earlier. For the quantile-based cohesion, episode nativ $\rightarrow$ tribe is valued 0.966, which is closer to what we would intuitively expect.

One more thing we noticed for the mean-based cohesion, is that beyond the 30 or so most cohesive episodes, we see the same words appearing over and over again, all with similarly low cohesion values; among others: *time* (present in 56% of episodes), *hand* (49%), *eye* (46%), *feel* (39%). To illustrate our point, Table 4.13 shows ten consecutive episodes, which are quite representative of the following 100 episodes.

While QCSP found fewer large episodes than FCI_{SEQ}, in our opinion the episodes it did find were more varied and interesting throughout the top 150. In Table 4.14 we give a selection of patterns from the top 150 in which we recognize nouns, expressions, or otherwise related words. At least for this dataset, we think the quantile-based cohesion measure is the better option.

Many of the patterns we find here we wouldn't have found at the top of the output of any frequency-based measures, for they are quite infrequent. Nevertheless, they are valid patterns — because they appear close together, they are highly likely to be semantically related. Indeed, we often recognize nouns and phrases.

In conclusion, we can say that the cohesion-based frequency measures certainly have their merit. Using the algorithms we were able to find some interesting patterns that weren't ranked anywhere near the top of the frequency-based rankings. Therefore we think that the cohesion-based methods are a valuable complement to the more traditional frequency-based measures.

#	mean-based cohesion (cohesion, support)
40	{ey, face, smile, hand} (0.0164)
41	{feel, ey, face, smile, hand} (0.0164)
42	{feel, ey, smile, time, hand} (0.0161)
43	{ey, smile, time, hand, thought} (0.0161)
44	{chapter, ey, smile, time, hand} (0.0161)
45	{chapter, felt, feel, time, thought} (0.0161)
46	{chapter, face, smile, time, hand} (0.016)
47	{troth, bridal} (0.016)
48	{ey, smile, hand} (0.016)
49	{face, smile, hand} (0.0159)

Table 4.13.: Ten consecutive patterns from the top 50 patterns by FCI_{SEQ} (mean-based cohesion).

#	quantile-based cohesion (cohesion, support)
27	bridal → pair (0.5, 20)
45	wag → tail (0.4, 20)
46	unbutton → waistcoat (0.3, 20)
62	lightn → thunder (0.364, 11)
67	hall → porter (0.344, 128)
91	hawk → hover (0.308, 13)
84	district → council (0.325, 77)
110	butter → salt (0.294, 17)
116	lime → tree (0.282, 71)
122	summer → villa (0.281, 64)
131	carpent → harrow → repair (0.25, 32)

Table 4.14.: A selection of patterns from the top 150 by QCSP (quantile-based cohesion).

5. Conclusion

After an introduction to sequential pattern mining, and after defining various measures of interestingness of patterns, ranging from long-established approaches to more novel ones, we devised and implemented an algorithm for mining frequent episodes and confident association rules. While the general strategy for mining episodes was consistent across all episode classes and frequency measures — breadth-first approach, Apriori-style pruning — quite a few specialized subalgorithms were needed. Many of these subalgorithms were based on preexisting algorithms, though most needed some adaptation for our purposes.

We saw that each of the interestingness measures have advantages and disadvantages. The implementation of the fixed-window frequency for parallel episodes performed the best, since it requires less bookkeeping in a database pass than the other approaches. Furthermore, computing the fixed-window confidence of an association rule is trivial, where the minimal-windows-based confidence measures require some more work. The fixed-window frequency values are counterintuitive, and depend highly on the chosen window width. Therefore the values should always be interpreted within the context of the chosen window width.

The minimal-windows-based measures produced more natural frequency values. The disjoint-window frequency values minimal windows of all sizes equally, at the risk of finding meaningless patterns if the window width is too large. The weighted-window frequency values smaller minimal windows more highly, which makes it less likely to find spurious patterns, but causes it to be less favourable towards potentially meaningful, larger episodes.

For all frequency-based measures, it was clear that simply ranking the episodes by frequency is not sufficient to discover all interesting patterns. We need techniques surface larger episodes, for instance by excluding smaller episodes — 1-episodes dominate the top of the rankings, yet they aren't usually considered to be the most interesting.

Nevertheless, we were able to find some interesting patterns in Tolstoy's *Anna Karenina* by constructing the top 15 episodes after filtering out the 1-episodes.

The association rules were in need of redundancy elimination. In particular, the rankings of the fixed-window confidence contained way too many association rules of confidence 1, with many giving no additional information beyond a more general rule also present.

The experiments with the cohesion-based measures showed that they are a valuable complement to the frequency-based measures. They found some interesting cohesive patterns that weren't very frequent — episodes which a frequency-based measure would

not score highly, but which, due to their minimal windows being small, were clearly related semantically.

Possible avenues for future work include further optimizing the performance of the implementation — while our implementation is most often able to find adequate output in reasonable amounts of time, the closed episode miner often completed a task in significantly less time, despite the fact that we have a computational advantage in that the search space is more restricted. Furthermore, it could be useful to conduct analysis with more varied datasets, to get an even better picture of the capabilities and the limitations of the measures and of the algorithm. Finally, finding more techniques of increasing the value of the output may be rewarding, such as preventing larger episodes from being inherently at a disadvantage, or exploring options to reduce redundancy, particularly for association rules.

6. Nederlandstalige samenvatting

Het doel van datamining is doorgaans om nuttige informatie uit grote datasets te halen – datasets die te groot zijn om manueel in detail door te kammen. In dit werk spitsen we ons toe op sequential pattern mining, waarbij we patronen zoeken in sequentiële gegevens, meer specifiek gegevens die voor te stellen zijn als een *event sequence*.

Na het uiteenzetten van een formeel kader met verschillende soorten patronen en manieren om te kwantificeren hoe interessant zo'n patroon is, beschrijven we een algoritme dat, gegeven een event sequence, alle patronen vindt die volgens een gegeven drempelwaarde interessant genoeg zijn.

Vervolgens testen we de efficiëntie van eem implementatie van het algoritme onder verschillende event sequences en parameters, en bestuderen we of we daarmee werkelijk interessante patronen vinden.

Bibliography

[1] Rakesh Agrawal and Ramakrishnan Srikant. “Mining sequential patterns”. In: *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*. IEEE. 1995, pp. 3–14.

[2] Rakesh Agrawal, Ramakrishnan Srikant, et al. “Fast algorithms for mining association rules”. In: *Proc. 20th int. conf. very large data bases, VLDB*. Vol. 1215. 1994, pp. 487–499.

[3] Boris Cule, Len Feremans, and Bart Goethals. “Efficient discovery of sets of co-occurring items in event sequences”. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2016, pp. 361–377.

[4] Boris Cule, Nikolaj Tatti, and Bart Goethals. “Marbles: Mining association rules buried in long event sequences”. In: *Statistical Analysis and Data Mining: The ASA Data Science Journal* 7.2 (2014), pp. 93–110.

[5] Len Feremans, Boris Cule, and Bart Goethals. “Mining Top-k Quantile-based Cohesive Sequential Patterns”. In: *Proceedings of the 2018 SIAM International Conference on Data Mining*. SIAM. 2018, pp. 90–98.

[6] Srivatsan Laxman, PS Sastry, and KP Unnikrishnan. “A fast algorithm for finding frequent episodes in event streams”. In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2007, pp. 410–419.

[7] Srivatsan Laxman, PS Sastry, and KP Unnikrishnan. “Discovering frequent episodes and learning hidden markov models: A formal connection”. In: *IEEE Transactions on Knowledge and Data Engineering* 17.11 (2005), pp. 1505–1517.

[8] V. C. Liao and M. Chen. “An efficient sequential pattern mining algorithm for motifs with gap constraints”. In: *2012 IEEE International Conference on Bioinformatics and Biomedicine*. 2012, pp. 1–1. DOI: 10.1109/BIBM.2012.6392660.

[9] Heikki Mannila, Hannu Toivonen, and A Inkeri Verkamo. “Discovery of frequent episodes in event sequences”. In: *Data mining and knowledge discovery* 1.3 (1997), pp. 259–289.

[10] Nikolaj Tatti and Boris Cule. “Mining closed strict episodes”. In: *Data Mining and Knowledge Discovery* 25.1 (2012), pp. 34–66.

Appendices

A. Most confident association rules from *tolstoy* sequence

The top 15 association rules mines from the *tolstoy* dataset, for each of the episode classes and confidence measures. Generated for analysis in Section 4.4.2.

A.1. Parallel, fixed-window confidence

#	association rule (fixed-window confidence)
1	{arkadyevitch, ey, love, youth, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
2	{stepan, arkadyevitch, love, youth, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
3	{arkadyevitch, love, youth, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
4	{arkadyevitch, ey, youth, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
5	{stepan, arkadyevitch, youth, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
6	{arkadyevitch, youth, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
7	{arkadyevitch, ey, love, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
8	{stepan, arkadyevitch, love, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
9	{arkadyevitch, love, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
10	{stepan, arkadyevitch, ey, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
11	{arkadyevitch, ey, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
12	{stepan, arkadyevitch, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
13	{arkadyevitch, gallant, steed, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)

14	{arkadyevitch, ey, love, youth, gallant, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)
15	{stepan, arkadyevitch, love, youth, gallant, token, declaim} ⇒ {stepan, arkadyevitch, ey, love, youth, gallant, steed, token, declaim} (1)

A.2. Serial, fixed-window confidence

#	association rule (fixed-window confidence)
1	countess → ivanovna → alexei ⇒ countess → lidia → ivanovna → alexei (1)
2	sergei → ivanovitch → ivanovitch ⇒ sergei → ivanovitch → sergei → ivanovitch (1)
3	levin → stepan → levin ⇒ levin → stepan → arkadyevitch → levin (1)
4	alexei → alexandrovitch → alexandrovitch ⇒ alexei → alexandrovitch → alexei → alexandrovitch (1)
5	stepan → arkadyevitch → alexandrovitch ⇒ stepan → arkadyevitch → alexei → alexandrovitch (1)
6	stepan → alexandrovitch ⇒ stepan → arkadyevitch → alexei → alexandrovitch (1)
7	stepan → stepan → arkadyevitch ⇒ stepan → arkadyevitch → stepan → arkadyevitch (1)
8	stepan → arkadyevitch → arkadyevitch ⇒ stepan → arkadyevitch → stepan → arkadyevitch (1)
9	vronski → alexandrovitch ⇒ vronski → alexei → alexandrovitch (1)
10	sergei → brother ⇒ sergei → ivanovitch → brother (1)
11	sergei → put ⇒ sergei → ivanovitch → put (1)
12	fellow → arkadyevitch ⇒ fellow → stepan → arkadyevitch (1)
13	alexandrovitch → ivanovna ⇒ alexandrovitch → lidia → ivanovna (1)
14	alexandrovitch → alexandrovitch ⇒ alexandrovitch → alexei → alexandrovitch (1)
15	alexei → ivanovna ⇒ alexei → lidia → ivanovna (1)

A.3. Parallel, minimal-window confidence

#	
1	{countess, countess, ivanovna, ivanovna} ⇒ {countess, countess, lidia, lidia, ivanovna, ivanovna} (1)
2	{ey, alexei, alexandrovitch, countess, lidia} ⇒ {ey, alexei, alexandrovitch, countess, lidia, ivanovna} (1)
3	{ey, alexei, countess, lidia} ⇒ {ey, alexei, alexandrovitch, countess, lidia, ivanovna} (1)

4	{stepan, alexei, alexei, alexandrovitch, alexandrovitch} ⇒ {stepan, arkadyevitch, alexei, alexei, alexandrovitch, alexandrovitch} (1)
5	{stepan, brother, alexei, alexandrovitch, law} ⇒ {stepan, arkadyevitch, brother, alexei, alexandrovitch, law} (1)
6	{stepan, arkadyevitch, brother, alexandrovitch, law} ⇒ {stepan, arkadyevitch, brother, alexei, alexandrovitch, law} (1)
7	{stepan, arkadyevitch, brother, alexei, law} ⇒ {stepan, arkadyevitch, brother, alexei, alexandrovitch, law} (1)
8	{stepan, stepan, arkadyevitch, arkadyevitch, alexandrovitch} ⇒ {stepan, stepan, arkadyevitch, arkadyevitch, alexei, alexandrovitch} (1)
9	{anna, alexandrovitch, countess, lidia, ivanovna} ⇒ {anna, alexei, alexandrovitch, countess, lidia, ivanovna} (1)
10	{anna, alexei, alexandrovitch, countess} ⇒ {anna, alexei, alexandrovitch, countess, lidia, ivanovna} (1)
11	{anna, alexandrovitch, countess} ⇒ {anna, alexei, alexandrovitch, countess, lidia, ivanovna} (1)
12	{countess, lidia, ivanovna, ivanovna} ⇒ {countess, lidia, lidia, ivanovna, ivanovna} (1)
13	{countess, countess, lidia, lidia} ⇒ {countess, countess, lidia, lidia, ivanovna} (1)
14	{kitti, agafea, agafea, mihalovna} ⇒ {kitti, agafea, agafea, mihalovna, mihalovna} (1)
15	{kitti, agafea, agafea} ⇒ {kitti, agafea, agafea, mihalovna, mihalovna} (1)

A.4. Serial, minimal-window confidence

#	minimal-window confidence
1	countess → ivanovna → alexei ⇒ countess → lidia → ivanovna → alexei (1)
2	sergei → ivanovitch → ivanovitch ⇒ sergei → ivanovitch → sergei → ivanovitch (1)
3	levin → stepan → levin ⇒ levin → stepan → arkadyevitch → levin (1)
4	alexei → alexandrovitch → alexandrovitch ⇒ alexei → alexandrovitch → alexei → alexandrovitch (1)
5	stepan → arkadyevitch → alexandrovitch ⇒ stepan → arkadyevitch → alexei → alexandrovitch (1)
6	stepan → alexandrovitch ⇒ stepan → arkadyevitch → alexei → alexandrovitch (1)
7	stepan → stepan → arkadyevitch ⇒ stepan → arkadyevitch → stepan → arkadyevitch (1)
8	stepan → arkadyevitch → arkadyevitch ⇒ stepan → arkadyevitch → stepan → arkadyevitch (1)
9	vronski → alexandrovitch ⇒ vronski → alexei → alexandrovitch (1)
10	sergei → brother ⇒ sergei → ivanovitch → brother (1)
11	sergei → put ⇒ sergei → ivanovitch → put (1)

12	fellow → arkadyevitch ⇒ fellow → stepan → arkadyevitch (1)
13	alexandrovitch → ivanovna ⇒ alexandrovitch → lidia → ivanovna (1)
14	alexandrovitch → alexandrovitch ⇒ alexandrovitch → alexei → alexandrovitch (1)
15	alexei → ivanovna ⇒ alexei → lidia → ivanovna (1)

A.5. Parallel, weighted-window confidence

#	association rule (weighted-window confidence)
1	{arkadyevitch, arkadyevitch} ⇒ {stepan, arkadyevitch, arkadyevitch} (1)
2	{alexandrovitch, alexandrovitch} ⇒ {alexei, alexandrovitch, alexandrovitch} (1)
3	{stepan, stepan} ⇒ {stepan, stepan, arkadyevitch} (1)
4	{countess, ivanovna} ⇒ {countess, lidia, ivanovna} (0.943)
5	{arkadyevitch, alexei} ⇒ {stepan, arkadyevitch, alexei} (0.937)
6	{stepan, arkadyevitch, alexandrovitch} ⇒ {stepan, arkadyevitch, alexei, alexandrovitch} (0.937)
7	{stepan, alexei, alexandrovitch} ⇒ {stepan, arkadyevitch, alexei, alexandrovitch} (0.935)
8	{stepan, alexandrovitch} ⇒ {stepan, alexei, alexandrovitch} (0.934)
9	{stepan, alexandrovitch} ⇒ {stepan, arkadyevitch, alexandrovitch} (0.934)
10	{stepan, good} ⇒ {stepan, arkadyevitch, good} (0.933)
11	{arkadyevitch, alexandrovitch} ⇒ {arkadyevitch, alexei, alexandrovitch} (0.933)
12	{stepan, alexei} ⇒ {stepan, arkadyevitch, alexei} (0.931)
13	{arkadyevitch, alexei, alexandrovitch} ⇒ {stepan, arkadyevitch, alexei, alexandrovitch} (0.927)
14	{alexandrovitch, ivanovna} ⇒ {alexandrovitch, lidia, ivanovna} (0.926)
15	{arkadyevitch, alexei} ⇒ {arkadyevitch, alexei, alexandrovitch} (0.925)

A.6. Serial, weighted-window confidence

#	association rule (weighted-window confidence)
1	sergei → levin ⇒ sergei → ivanovitch → levin (1)
2	alexandrovitch → alexandrovitch ⇒ alexandrovitch → alexei → alexandrovitch (1)
3	stepan → levin ⇒ stepan → arkadyevitch → levin (1)
4	stepan → smile ⇒ stepan → arkadyevitch → smile (1)
5	alexei → alexandrovitch → alexandrovitch ⇒ alexei → alexandrovitch → alexei → alexandrovitch (1)
6	stepan → stepan ⇒ stepan → arkadyevitch → stepan (1)
7	countess → ivanovna ⇒ countess → lidia → ivanovna (1)
8	levin → arkadyevitch ⇒ levin → stepan → arkadyevitch (1)

9	arkadyevitch → arkadyevitch ⇒ arkadyevitch → stepan → arkadyevitch (1)
10	alexei → alexei → alexandrovitch ⇒ alexei → alexandrovitch → alexei → alexandrovitch (0.984)
11	alexei → alexei ⇒ alexei → alexandrovitch → alexei (0.898)
12	alexandrovitch → alexei → alexandrovitch ⇒ alexei → alexandrovitch → alexei → alexandrovitch (0.846)
13	alexandrovitch → alexandrovitch ⇒ alexei → alexandrovitch → alexei → alexandrovitch (0.846)
14	alexandrovitch → alexandrovitch ⇒ alexei → alexandrovitch → alexandrovitch (0.846)
15	alexei → alexandrovitch → alexei ⇒ alexei → alexandrovitch → alexei → alexandrovitch (0.833)