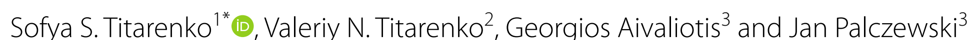
Titarenko *et al. J Big Data (2019) 6:37*
https://doi.org/10.1186/s40537-019-0200-9

Journal of Big Data



METHODOLOGY **Open Access**

Fast implementation of pattern mining algorithms with time stamp uncertainties and temporal constraints

Sofya S. Titarenko[1*], Valeriy N. Titarenko[2], Georgios Aivaliotis[3] and Jan Palczewski[3]

*Correspondence: S.Titarenko@hud.ac.uk
[1] School of Computing and Engineering, University of Huddersfield, Queensgate, Huddersfield HD1 3DH, UK
Full list of author information is available at the end of the article

Abstract

Pattern mining is a powerful tool for analysing big datasets. Temporal datasets include time as an additional parameter. This leads to complexity in algorithmic formulation, and it can be challenging to process such data quickly and efficiently. In addition, errors or uncertainty can exist in the timestamps of data, for example in manually recorded health data. Sometimes we wish to find patterns only within a certain temporal range. In some cases real-time processing and decision-making may be desirable. All these issues increase algorithmic complexity, processing times and storage requirements. In addition, it may not be possible to store or process confidential data on public clusters or the cloud that can be accessed by many people. Hence it is desirable to optimise algorithms for standalone systems. In this paper we present an integrated approach which can be used to write efficient codes for pattern mining problems. The approach includes: (1) cleaning datasets with removal of infrequent events, (2) presenting a new scheme for time-series data storage, (3) exploiting the presence of prior information about a dataset when available, (4) utilising vectorisation and multicore parallelisation. We present two new algorithms, FARPAM (FAst Robust PAttern Mining) and FARPAMp (FARPAM with prior information about prior uncertainty, allowing faster searching). The algorithms are applicable to a wide range of temporal datasets. They implement a new formulation of the pattern searching function which reproduces and extends existing algorithms (such as SPAM and RobustSPAM), and allows for significantly faster calculation. The algorithms also include an option of temporal restrictions in patterns, which is available neither in SPAM nor in RobustSPAM. The searching algorithm is designed to be flexible for further possible extensions. The algorithms are coded in C++, and are highly optimised and parallelised for a modern standalone multicore workstation, thus avoiding security issues connected with transfers of confidential data onto clusters. FARPAM has been successfully tested on a publicly available weather dataset and on a confidential adult social care dataset, reproducing results obtained by previous algorithms in both cases. It has been profiled against the widely used SPAM algorithm (for sequential pattern mining) and RobustSPAM (developed for datasets with errors in time points). The algorithm outperforms SPAM by up to 20 times and RobustSPAM by up to 6000 times. In both cases the new algorithm has better scalability.

Keywords: Pattern mining, Temporal data, Uncertainty, Optimisation, OpenMP

Springer Open

© The Author(s) 2019. This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

Introduction

Many datasets are produced in science and technology these days. Thus there is demand to efficiently extract useful information from these data and be able to predict certain events in the future. This may help a person, organisation or society to optimise their time and resources. Statistical methods can provide solutions in many cases. But when the quantity of data is too large, they may become too slow to be used in practical applications. In this case data mining methods are used. Data mining includes many fields, such as clustering, classification, outlier analysis and frequent pattern mining. As datasets become ever larger and new computing architectures emerge, researchers need to adapt existing algorithms to be used in a more efficient way.

A particular problem arises with datasets related to personal data such as health records [1, 2]. Great care in processing of sensitive data is imperative. In some cases, when use of a network is unavoidable, a combination of isolating the network and data analysis of previous cyber-attacks can be a good solution [3]. While modern remote supercomputers allow rapid solution of problems, their use may be prohibited where data confidentiality is paramount. Therefore it is important to develop software utilising all the available resources of a standalone (and not necessarily high-end) workstation. To achieve that, two main issues should be addressed: (1) efficient methods of database storage (see, for example, [4]) and (2) algorithm optimisation.

In this paper we focus on the problem of frequent pattern mining, which was first formulated in the early 1990s [5, 6]. It includes several classes of mining problems applied to sequential or temporal patterns, frequent itemset mining, and association rules. Mining through sequential datasets and itemsets, the corresponding problem of association rules, are relatively straightforward to implement. Such problems have been well studied and include algorithms developed for serial implementation (sequential pattern mining (SPADE, SPAM, FreeSpan, PrefixSpan) [7–10], constraint-based sequential pattern mining (CloSpan, Bide) [11, 12] and mining for frequent itemsets and for association rules [5, 13]). Most of the serial algorithms mentioned above have been modified to run on high performance computers. For example, pSPADE [14] is a parallelised version of SPADE with the use of a shared memory interface, and PrefixSpan has been parallelised using MPI instructions [15]. In other cases, new parallel algorithms have been proposed, for example using hybrid OpenMP-MPI [16], and parallel sequential pattern mining applied to so-called massive trajectory data [17]. For more examples of pattern mining algorithms and a list of platforms for which they have been adapted see [18–22].

Particular problems in pattern mining arise with temporal data, which are widely collected for various purposes in social, health, consumer, environmental, and medical areas, communications, and financial monitoring [2, 23–32]. These data include time as a parameter, either as a set of discrete time points or, in more sophisticated problems, timestamps. Timestamps could include either instantaneous events, such as temperature measurements, or events which happen over a period of time.

Existing temporal pattern mining algorithms usually include either interval based representations (which typically exploit Allen-type relationships, e.g. [25]) or time-point representations, reducing an interval between start and end time-point events. The first approach more fully describes frequent patterns, but is very expensive in CPU time and storage space due to the large number of relations that need to be checked for each

candidate pattern. Temporal reasoning may help to optimise performance and reduce storage requirements [25]. An example of a method for discovering frequent temporal *interval arrangements* is presented in [33], and mining for association rules in [34]. Unfortunately, these algorithms are not publicly available. The second approach allows to apply sequential pattern mining, or time series pattern mining techniques (for example SPAM). This approach has the advantages of using less storage space and being easier to implement, e.g. [34–37]. Another approach to mining through timestamped events is relevance weight pattern mining [38]. This has been applied to building an activity detection model, based on assigning relevance weights to the recorded activities.

It is often desirable to put additional constraints on patterns to be found. For example, if a pattern is met more frequently than a predefined threshold it is called *frequent*. Another example is considering patterns which took place in the last *n* years, and have good predictivity [2]. Other examples of possible constraints include *item constraints*, *model-based constraints*, *length* or *temporal length* constraints. For example, an abnormal blood pressure measurement can be associated with a stroke that took place in the following week but it might be hard to associate it with a stroke taking place a decade later. Examples of constrained pattern mining can be found in [2, 35, 39–44].

In time-series datasets and streamed data the time of the recorded event can often contain an error. This may happen due to faults in sensors, errors in signal sampling, etc. The use of standard algorithms designed for error-free data may lead to incorrect results, therefore an appropriate probabilistic model as well as suitable pattern mining algorithm should be used. A sliding window algorithm has been proposed in [45]. In [29, 46] a model assigns a certain probabilities to the events.

There also can be uncertainty with regards to the time stamps of temporal data, and hence the sequence of events. For example, suppose we have two events A and B which are likely to happen during time intervals $[t_A^s, t_A^e]$ and $[t_B^s, t_B^e]$. If these intervals overlap, it means that there could be a probability of event A happening before event B as well as event B happening before event A. If this possibility is not taken into account a mining algorithm will mine this record only for one type of pattern. This will lead to incorrect estimates of how frequent the patterns AB and BA are in the dataset. For example, the algorithm RobustSpam [37] takes into account the possibility of inaccuracies in the way timestamps are recorded. This approach is focused on using time points instead of intervals and fitting probabilistic models for the errors in the time stamps around these time points. Intervals were represented as a start and end points with the possibility of different errors around those points. RobustSpam also allows for deliberate introduction of uncertainty to protect patient confidentiality. Other examples of mining datasets with uncertainties can be found in [47–49].

In this paper we present an integrated approach to the optimisation of pattern mining. After data cleaning, we first remove infrequent events, then arrange data storage to optimise searching, use prior information where possible, and optimise calculation speeds using OpenMP and vectorisation. We present two new algorithms, FARPAM (FAst Robust PAttern Mining) and FARPAMp (when prior information is used), which are applicable to a wide range of datasets recorded with time stamps. A new formulation of the pattern searching function is implemented which reproduces and extends the outputs of existing algorithms (such as SPAM

Table 1 Algorithms tested in the paper

Name	Language	Source	Algorithm
Apriori	C++	Present paper	PM with uncertainty intervals and temporal length restriction
Apriori + bitmap			
Apriori + bitmap + openMP			
FARPAM			
FARPAMp			
RobustSpam	Java	[37]	PM with uncertainty intervals
SPAM	Java	[8]	SPAM

Table 2 Optimisation details of the tested algorithms

Name	Bitmap ID lists	OpenMP	New way of storage	Prior knowledge
Apriori	No	No	No	No
Apriori + bitmap	Yes	No	No	No
Apriori + bitmap + openMP	Yes	Yes	No	No
FARPAM	Yes	Yes	Yes	No
FARPAMp	Yes	Yes	Yes	Yes
RobustSpam	No	No	No	No
SPAM	Yes	No	No	No

and RobustSPAM), and also allows significantly faster calculation. The algorithms also include the option of temporal restrictions in patterns, which is included neither in SPAM nor in RobustSPAM. In practice, uncertainty intervals for event time-stamps are estimated using field expertise and are, in general, different for different events. The algorithm FARPAM covers such a general case. However, in some cases there is prior information allowing us to reduce the number of possible pairwise relations between uncertainty intervals. For instance, we may know that uncertainty intervals for all events are identical. In this case the searching function can be further optimised resulting in faster computing times. This is implemented in the algorithm FARPAMp. Both algorithms include a full range of optimisation measures (removing infrequent events, a new way of efficiently storing datasets, using binary ID lists and multithreading). The algorithms are coded in C++, and are highly optimised and parallelised for a modern standalone multicore workstation, thus avoiding security issues with transfer of confidential data onto a cluster. They are profiled for applications to (1) itemsets using the existing and highly optimised SPAM code, (2) time-series recorded with errors in timestamps using RobustSPAM. We show that they are both faster and better scalable than SPAM and RobustSPAM. Table 1 presents a summary of the algorithms we have developed (FARPAM and FARPAMp), their intermediate versions (*Apriori*, *Apriori* + bitmap and *Apriori* + bitmap + openMP) and the methods we used for profiling and verifying outputs (RobustSPAM and SPAM). Table 2 shows the levels of optimisation used in the algorithms (fully described in "Steps in constructing and optimising FARPAM" section). All algorithms presented in the table produce identical outputs (the list of frequent patterns found for a given level of support, and the frequency and ID

of entries where they have been found) for the same problems. The algorithms are applied to a confidential social care dataset and to a publicly accessible meteorological dataset.

Methodology

Definitions and notations

Suppose we have m unique items and denote them as i_j, $j = 1, \dots, m$. According to [6] an *itemset* is a non-empty set of items and a *sequence* is an ordered list of itemsets. We may denote an itemset I by $(i_1, i_2, \dots, i_n)$, where i_j is an item, and a sequence S by $\langle s_1, s_2, \dots, s_k \rangle$, where s_j is an itemset. If a sequence consists of n itemsets we call it *sequential pattern of length n* or *n-pattern*. It is possible that the same itemset may appear several times in a pattern, e.g. $\langle s_1, s_2, s_1, s_3, s_1, s_1 \rangle$. Note that the order of items within an itemset is not important while the order of itemsets matters and sequences $\langle s_1, s_2, s_1, s_3, s_1, s_1 \rangle$ and $\langle s_1, s_1, s_1, s_1, s_3, s_2 \rangle$ are different.

A database $\mathcal{D}$ is formed of several records (sets of sequences). Each record is allowed to have a different number of sequences. All these records can be enumerated as ρ_k, $k = 1, \dots, r$. For the k-th record of $\mathcal{D}$ we can check if a given pattern p is a subpattern of ρ_k. In such a way we may count the number of records containing the given pattern p. This number divided by the total number of records in the database is called the *support* for the pattern p. By a *frequent pattern* we mean a pattern whose support is not less than a given threshold σ (*minimum support*). Our problem is to find all possible frequent patterns in the database $\mathcal{D}$.

In this paper we consider datasets consisting of *events* rather than items or sequences of itemsets. Each event is a triple $\{e, t_s, t_e\}$, where e is a coded item, $[t_s, t_e]$ is an interval during which the event is equally probable to occur (*uncertainty interval*).

Let a customer regularly buy grocery in a supermarket. Each transaction contains a set of purchased items and represents a single row in the dataset. If the exact time of purchase is not relevant we can apply a standard sequential pattern algorithm like SPAM to search for frequent patterns. However, in some problems the time relation between the transactions can be important.

A sequence $\langle e^{j_1}, e^{j_2}, \dots, e^{j_s} \rangle$ of s events from database $\mathcal{D}$ is called *ordered* if

$$\forall i = 1, \dots, s \qquad t_s^{j_i} \leq t_s^{j_{i+1}} \tag{1}$$

An ordered sequence $\langle e^{j_1}, e^{j_2}, \dots, e^{j_s} \rangle$ of s events from $\mathcal{D}$ is called a pattern of length s or $\boldsymbol{s}$-pattern if

$$\forall i = 2, \dots, s \qquad \tau \equiv \sup_{k=1,\dots,i-1} t_s^{j_{i-1}} \leq t_e^{j_i}. \tag{2}$$

In the case of no uncertainty, the size of interval $[t_s^j, t_e^j]$ is reduced to 0. Therefore $t_s^j = t_e^j \equiv t^j$ and pattern can be defined as an *ordered* sequence of events $\langle e^{j_1}, e^{j_2}, \dots, e^{j_s} \rangle$. If the exact time t^j is not important we can sort the events in each row (see Definition 1) and apply traditional SPAM. FARPAM and FARPAMp suggested in this work are also applicable.

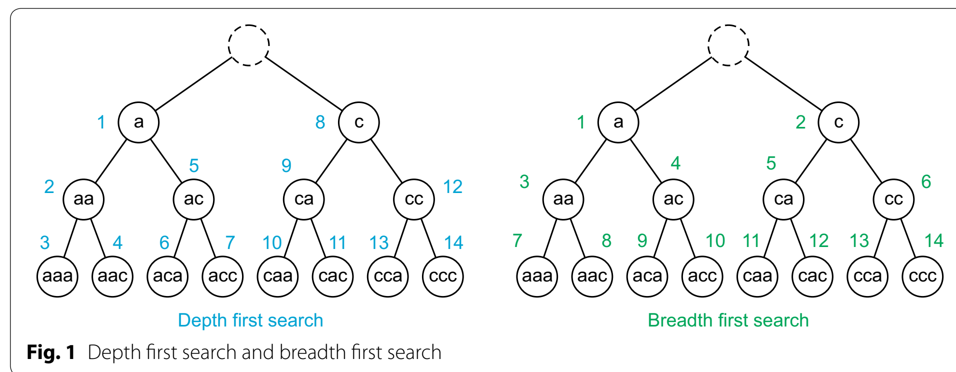


Fig. 1 Depth first search and breadth first search

Sequential patterns

There are two main groups of sequential pattern mining algorithms: breadth first search (BFS) and depth first search (DFS). They come from methods used in artificial intelligence, e.g. [50], and depend on how a search tree is constructed. To explain the difference we consider an example in Fig. 1.

BFS methods suppose that all frequent patterns of a given length k are known, including where $k = 0$. We then search for $(k + 1)$-patterns and once all those frequent patterns are found we search for $(k + 2)$-patterns, and so on until all frequent patterns have been found. Classical examples of BFS methods can be *Apriori*-like algorithms first suggested in [5, 6] and based on ideas from the so-called *Apriori* Principle [51]: "All non-empty subsets of a frequent itemset must also be frequent". BFS-like algorithms generate all k-patterns in each k-th iteration and move to the next $k + 1$ step only after exploring the entire k-th search space. This idea was later extended for the frequent itemset algorithm in [52–54] and for the case of constrained itemsets, in [55–57].

DFS methods, in contrast, construct the search tree by finding all frequent extensions of the current pattern before exploring other frequent patterns at the same level of the search tree, i.e. of the same length. The first DFS-like algorithm was suggested by introducing the FP-grow method [13]. Subsequently, several improvements have been developed, for example vertical representation of the database, or introduction of so called "TID-array" to link frequent itemsets to arrays of transaction IDs [58, 59]. In application to sequential pattern mining problems similar ideas were suggested in SPADE [7], SPAM [8], FreeSpan [9] and PrefixSpan [10]. Generally DFS methods are more difficult to parallelise.

The main disadvantage of BFS methods is related to memory usage, as a user needs to store all frequent k-patterns if any $(k + 1)$-pattern is to be found. While memory requirements for DFS algorithms can be less demanding compared to BFS, we may still need to store data related to all previous patterns, which can be challenging, especially when patterns are long. In this paper we only consider BFS methods.

Temporal data

It is often important to know not only the fact that an event e^j took place but also the time t^j at which it happened. We talk about a *time series* database when a sequence of events $\{e^j\}$ is ordered according to their times t^j. Temporal data may also have an *interval-based*

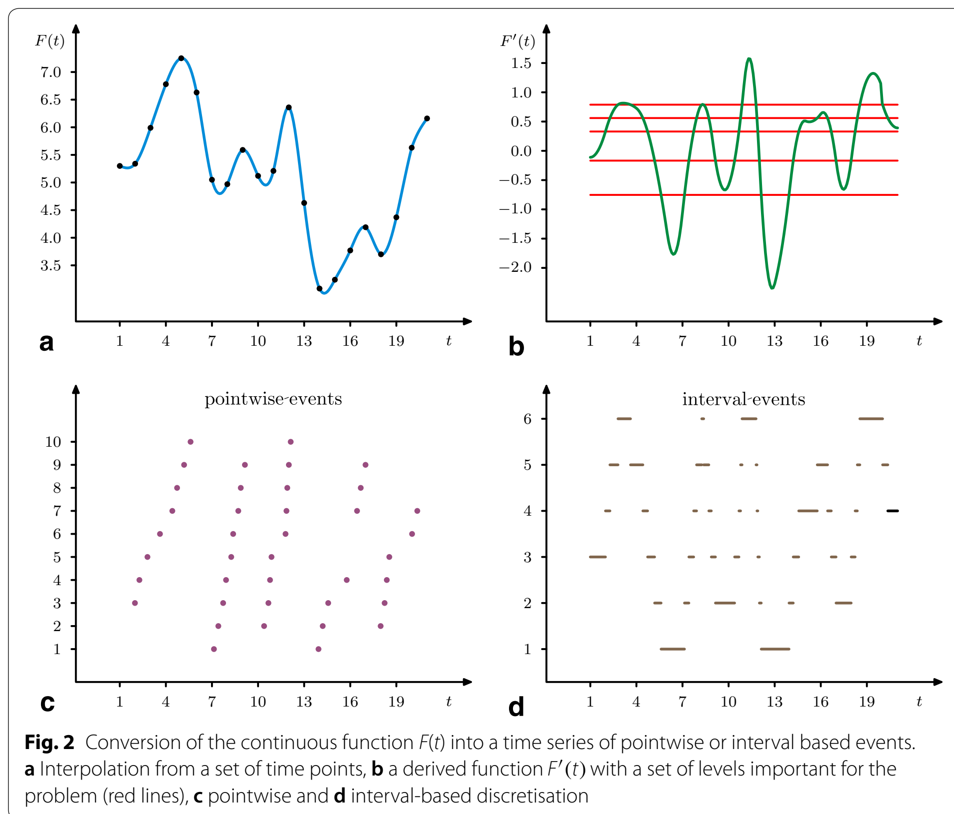


Fig. 2 Conversion of the continuous function *F*(*t*) into a time series of pointwise or interval based events. **a** Interpolation from a set of time points, **b** a derived function $F'(t)$ with a set of levels important for the problem (red lines), **c** pointwise and **d** interval-based discretisation

structure when an event happens between start t_s and end t_e time points. Naturally, there may be other types of temporal datasets, for example continuous functions of time. There are several ways to convert those functions into temporal datasets.

The following conversion procedure was used to generate events for the weather dataset (see "Evaluation" section for a full description). Suppose a variable F from the database can be written as a function $F(t)$ of time. If the function is known at discrete time points only, e.g. as shown in Fig. 2a, we can use various interpolation techniques such as cubic spline interpolation. Let there be n time points t_i, $i = 1, \ldots, n$, with known values of $F(t)$. We may use a cubic spline approximation $\hat{F}(t)$ of $F(t)$ given by functions $F_i(t)$ for each $t \in [t_i, t_{i+1}]$, $i = 1, \ldots, n-1$:

$$\hat{F}_i(t) \equiv a_i(t - t_i)^3 + b_i(t - t_i)^2 + c_i(t - t_i) + d_i. \tag{3}$$

As the values of $F(t)$ are known at t_i, we get $\hat{F}_i(t_i) = F(t_i)$ and $\hat{F}_i(t_{i+1}) = F(t_{i+1})$, $i = 1, \ldots, n-1$. We also require $\hat{F}(t)$ to have its first and second derivatives continuous at t_i, $i = 1, \ldots, n-2$: $\hat{F}'_i(x_{i+1}) = \hat{F}'_{i+1}(x_{i+1})$, $\hat{F}''_i(x_{i+1}) = \hat{F}''_{i+1}(x_{i+1})$. To have a unique solution we also put the following condition at the endpoints $\hat{F}''_1(x_1) = \hat{F}''_{n-1}(x_n) = 0$. Solving the system of linear equations we find the approximation $\hat{F}(t)$ everywhere on $[t_1, t_n]$.

It is often necessary to preprocess the original data by some procedure. For instance, each patient may have their unique normal/abnormal values depending on their gender, age, or race. Therefore the original function should be scaled in order to get meaningful functions across many patients. In other cases we may use some values derived from the

original function. As an illustration, we may measure wind speed at several locations. Clearly, some locations may be very windy, there can be strong seasonal variations and each place has its own wind rose showing how wind speed and direction are usually distributed. So a normalisation procedure should be applied to the data, in order to know if wind speed has normal or extreme values for the chosen place and season. If we process temperature records, then it may be better to consider the first derivative $F'(t)$ rather than absolute values $F(t)$, see Fig. 2b. Now we need to convert the derived function into a set of temporal events, which can be done by introducing specific level values for the derived function [60]. We find time points when $F'(t)$ intersects the given levels and take into account if the function is decreasing or increasing at those time points. Thus for 5 levels shown in Fig. 2b we get $2 \cdot 5 = 10$ events (5 for decreasing and 5 for increasing values) in Fig. 2c. On the other hand, we may also have interval based events when the function is between two levels (or above the highest/below the lowest levels), see $5 + 1 = 6$ events in Fig. 2d.

Interval pattern mining problems can be challenging and take up a lot of time and data storage resources to solve, as various relations between any two time intervals should be considered. A simplified approach is to convert any interval-based event e^j taking place within an interval $[t_s^j, t_e^j]$ into two events $\{e_s^j, t_s^j\}$ and $\{e_e^j, t_e^j\}$, where the superscripts s and e denote synthetic events corresponding to the beginning and end of the interval for event e^j. In this way we should solve a the problem of pointwise time series events, see examples in [34–37]. A required sequential or time series pattern mining algorithm can be applied afterwards.

Uncertainty in data

There may be cases when we are not sure when a given event took place. For example, some health related measurements concerning a patient may be taken several times per day, while other procedures like CT/MRI/ultrasound can be performed less frequently due to their potential health hazards, costs or availability. The time taken to process and record results processing results may also differ, meaning that they may not be recorded in the order the actual measurements took place. Finally, there may simply be errors in transcribing manually collected data. All these issues introduce time uncertainty into the record. In the case of weather data some weather stations may have older equipment and report only daily values. These daily values allow us to roughly estimate when a given event took place, e.g. gale force winds started at 9 p.m., however there will be some uncertainty in our estimation, e.g. 3 h. Therefore we are given an approximate time t^j with some uncertainty $\beta^j \geq 0$, so we are sure the event took place between t_s^j and t_e^j time points where $t_s^j \equiv t^j - \beta^j$ and $t_e^j \equiv t^j + \beta^j$. Parameter β^j is allowed to be zero when we know precisely when an event took place. Examples of databases with uncertainties can be found in [29, 37, 47].

Steps in constructing and optimising FARPAM

In this paper, we present a novel fast algorithm that enables classical temporal pattern mining, but can also accommodate much more complicated problems like patterns with temporal length constraints. It can potentially be used to solve problems when the uncertainty interval depends on a particular event. The efficiency of the proposed

approach is based on a number of algorithmic advances (bitmap ID-lists, a special way of database representation, etc.) as well as multi-thread processing. To the best of our knowledge there is currently no parallelised algorithm that takes into account complexity introduced by uncertainty in datasets and temporal length restrictions on patterns.

We test our algorithms on two datasets: a weather dataset and an adult social care dataset. In the application to the adult social care dataset, we may want to look for frequent patterns where events happen over a certain period of time (for example, a few years). This restriction may help to reduce the number of frequent patterns used for the future classification or predictive models.

This section shows in details the modification steps we apply to the *Apriori* algorithm to achieve the final highly optimised FARPAM and FARPAMp. "Apriori principle" section describes the main principles of the *Apriori* algorithm (Algorithm 1). "Binary vectors" section shows modifications needed in order to achieve first-step optimisation. "Sorting frequent patterns" section talks about the way to accelerate the searching function by sorting found frequent patterns. The implementation of "ordering" the array of frequent s-patterns is described in "Forming a list of candidate patterns" section (Algorithm 2), its parallel implementation is shown on Fig. 4 and discussed in "Multithreading" section. "Events with uncertainty intervals" section explains our approach to storing data and demonstrates how it works if applied to the datasets with uncertainty intervals. This is a universal method and can be applied to any dataset. Algorithms 4 and 5 present the method for searching for patterns in a dataset recorded in the *our new format*. This can be especially efficient for datasets with many repeated events. "Data with the same uncertainty for alike events" section shows how *prior* information can be used for further optimisation (implemented in FARPAMp). The searching function algorithm is provided in Algorithms 6. "Time restriction" section explains how time restriction is put on a frequent pattern (Algorithm 7). The time restriction condition is implemented in both FARPAM and FARPAMp.

***Apriori* principle**

For a standard *Apriori*-like approach each $(s-1)$-pattern p is extended by one event e, so a new candidate s-pattern $\langle p, e\rangle$ is formed (see for example [5]). Then for each record in the database $\mathcal{D}$ we check if the given s-pattern is a subpattern of the record. See Algorithm 1 as an example algorithm to find frequent patterns. One can see that to find new patterns of length s we need to consider $n_{s-1} \cdot n_1$ combinations, and for each combination, all r records from database $\mathcal{D}$ should be considered. This can be time-consuming, so we want to use the *Apriori* Principle in order to reduce the number of checks. Suppose a candidate s-pattern is a sequence $\langle e^{j_1}, e^{j_2}, \ldots, e^{j_s}\rangle$. This pattern can be a subpattern of a given i-th record only when all its subpatterns are also subpatterns of the i-th record. By removing one event from the pattern $\langle e^{j_1}, e^{j_2}, \ldots, e^{j_s}\rangle$ we may form s subpatterns of length $(s-1)$, i.e. $\langle e^{j_2}, e^{j_3}, \ldots, e^{j_s}\rangle$, $\langle e^{j_1}, e^{j_3}, \ldots, e^{j_s}\rangle$, $\langle e^{j_1}, e^{j_2}, e^{j_4}, \ldots, e^{j_s}\rangle, \ldots, \langle e^{j_1}, e^{j_2}, e^{j_3}, \ldots, e^{j_{s-1}}\rangle$. According to the *Apriori* Principle all these $(s-1)$-patterns must be subpatterns of the i-th record and also belong to the set $\mathcal{F}_{s-1}$ in order for the candidate s-pattern to be a subpattern of the i-th record and have a chance of being frequent.

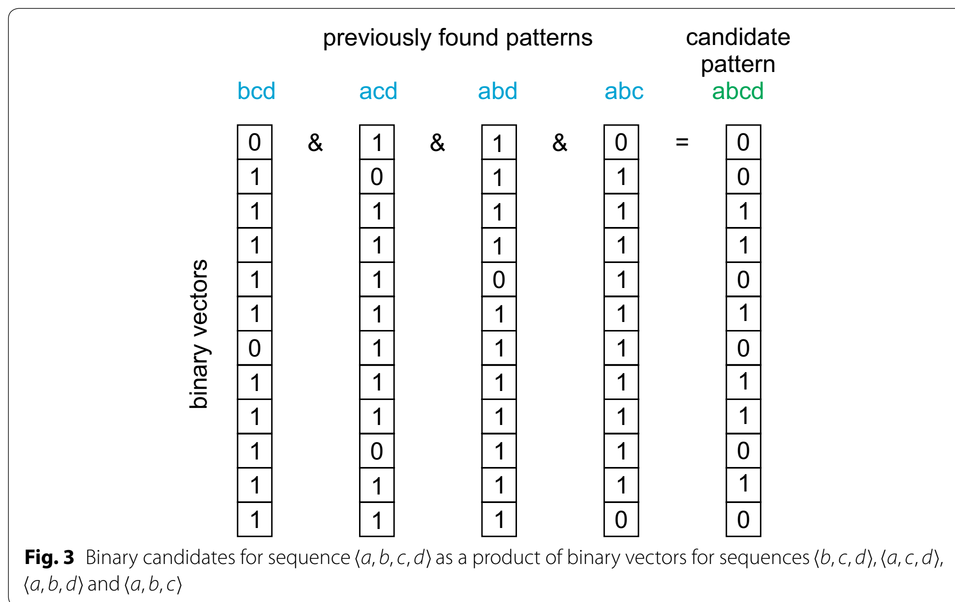


Fig. 3 Binary candidates for sequence $\langle a, b, c, d \rangle$ as a product of binary vectors for sequences $\langle b, c, d \rangle$, $\langle a, c, d \rangle$, $\langle a, b, d \rangle$ and $\langle a, b, c \rangle$

```
input  : A database D
input  : A set F_{s-1} of (s-1)-patterns, n_{s-1} patterns in total
input  : A set F_1 of 1-patterns, n_1 patterns in total
input  : The minimum support σ
output: A set F_s of s-patterns, n_s patterns in total
set F_s to the empty set;
n_s ← 0;
foreach pattern p from F_{s-1} do
    foreach element e from F_1 do
        newCandidatePattern ← ⟨p, e⟩;
        κ ← findSupport(newCandidatePattern, D);
        if κ ≥ σ then
            add newCandidatePattern to F_s;
            n_s ← n_s + 1;
        end
    end
end
```

Algorithm 1: *Apriori* algorithm for breadth first search

Binary vectors

In order to use the *Apriori* principle we introduce a binary vector v of length r for each pattern p. An i-th element of this vector is 1 when the pattern p is a subpattern of the i-th record, otherwise $v^i = 0$. Suppose a new candidate s-pattern p is given. By p_k, $k = 1, \ldots, s$ we denote all its s subpatterns of length $(s-1)$ and the corresponding binary vectors are v_k. If p is a subpattern of an m-th record of database $\mathcal{D}$, then all elements p_k^m must be 1. Therefore we form a candidate binary vector $\tilde{v}$ such that $\tilde{v}^m = v_1^m \wedge v_2^m \wedge \ldots v_s^m$ where $\wedge$ is the binary AND operator, see Fig. 3 for an example.

If the number of non-zero elements of vector $\tilde{v}$ is less than the minimum support σ, then the candidate pattern p will not be a frequent one. However, if the number of non-zero elements of $\tilde{v}$ is greater or equal to σ, then we need to check all records corresponding to $\tilde{v}^m = 1$. For example, pattern $\langle b, a, c, d, a, b, d, c \rangle$ contains subpatterns $\langle b, c, d \rangle$, $\langle a, c, d \rangle$, $\langle a, b, d \rangle$ and $\langle a, b, c \rangle$, i.e. $\langle \mathbf{b}, a, \mathbf{c}, \mathbf{d}, a, b, d, c \rangle$, $\langle b, \mathbf{a}, \mathbf{c}, \mathbf{d}, a, b, d, c \rangle$, $\langle b, \mathbf{a}, c, d, a, \mathbf{b}, \mathbf{d}, c \rangle$ and $\langle b, \mathbf{a}, c, d, a, \mathbf{b}, \mathbf{d}, c \rangle$, however it does not contain pattern $\langle a, b, c, d \rangle$.

Suppose we have N candidate records, i.e. records with $\tilde{v}^m = 1$. We start processing them and count the number μ of records where p is not a subpattern of the record. Once $1 - \mu/N < \sigma$, then there is no need to check the remaining records as the total number of records with p as a subpattern will be less than σN and the candidate pattern p is not a frequent one. Otherwise, after checking all the records we form the new binary vector v.

For numerical implementation it is important to remember that a logical variable usually requires at least 1 byte (8 bits) of memory. The use of logical variables for the vector v will require 8 times more memory than is really needed. Therefore for a given number r of records we can always assume that r is divisible by 32, otherwise we may create extra empty records to fulfil this assumption. So we can store a binary vector v in an $r/32$ long vector of 32-bit numbers. If $m = 32 \cdot M + i$, where $0 \leq i < 32$, then the element v^m is the i-th bit of the M-th 32-bit element in a storage system (we use C/C++ notations where the first element of a vector is stored at the 0-th position in memory). Then a bitwise AND operator can be applied to the corresponding M-th elements of vectors $v_1, \ldots, v_s$. The use of bitwise operators for 32-bit numbers instead of a logical operator for 8-bit variables helps us not only to reduce the size of memory for data storage but also allows a CPU to issue 32 times less instructions. This idea resembles the bitmap ID list storage idea, first proposed in [8], and can be easily extrapolated to the case of sequential pattern mining.

If the number of records is large, then further steps for optimisation can be used. For instance, so called SIMD intrinsics (Single Instruction for Multiple Data) can be called when the same bitwise AND operator can be applied to an 128-, 256- or 512-bit number in one instruction

```
_mm_and_si128 (__m128i a, __m128i b),
_mm256_and_si256 (__m256i a, __m256i b),        (4)
_mm512_and_si512 (__m512i a, __m512i b),
```

thus possibly saving processing time by up to a further 4, 8 or 16 times depending on the CPU type. However, in many cases there may be no need for low level optimisation as modern C/Fortran compilers may optimise a code themselves if they are aware of the number r of records for each vector v and the alignment of the vector in global memory.

Sorting frequent patterns

Suppose we have constructed the set $\mathcal{F}_{s-1}$ of frequent patterns and a new candidate pattern p of length s is formed. By p_k, $k = 1, \ldots, s$ we denote subpatterns of p such that p_k is a pattern p without the k-th element. To apply the procedure described above we need to find the corresponding binary vectors for all s subpatterns p_k. If at least one of the subpatterns does not belong to $\mathcal{F}_{s-1}$ the candidate pattern p does not belong to $\mathcal{F}_s$. It may happen that in a real application the number of patterns in $\mathcal{F}_{s-1}$ is relatively large, i.e. thousands or millions of patterns. Therefore we need a smarter approach to finding the position of p_k within the set $\mathcal{F}_{s-1}$.

We aim to order patterns in $\mathcal{F}_s$ using lexicographical ordering. If $s = 1$, then all patterns can be ordered according to the index of each event, i.e. $e^i < e^j$ if $i < j$. For $s > 1$ two different patterns $p = \langle e^{i_1}, e^{i_2}, \ldots, e^{i_s} \rangle$ and $q = \langle e^{j_1}, e^{j_2}, \ldots, e^{j_s} \rangle$ can also be ordered.

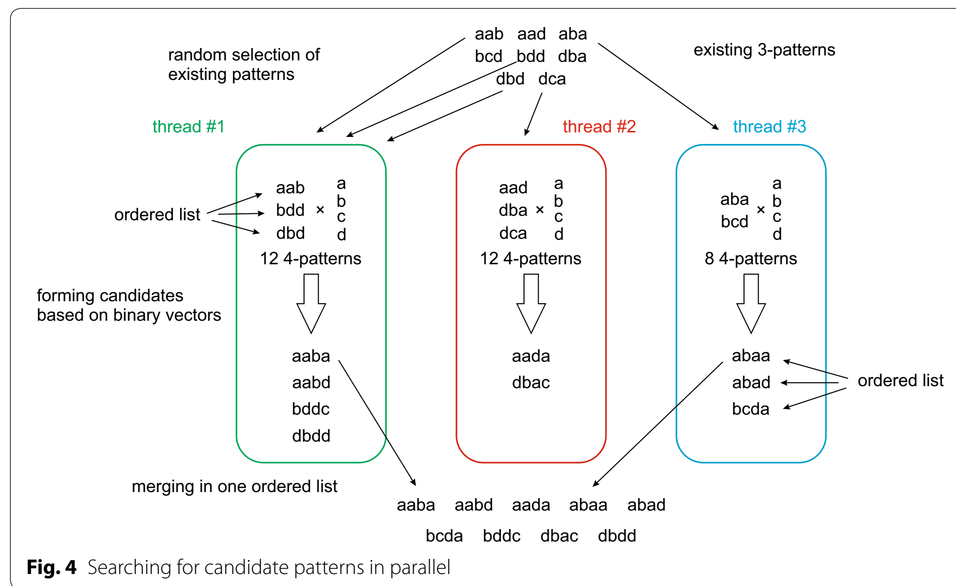


Fig. 4 Searching for candidate patterns in parallel

We denote $p < q$ if there is a number $m \geq 1$ such that $e^{i_k} = e^{j_k}$ for $k = 1, \ldots, m-1$ and $e^{i_m} < e^{j_m}$ (or in an equivalent manner $i_m < j_m$). Suppose all patterns in the original sets $\mathcal{F}_1$ and $\mathcal{F}_{s-1}$ are ordered in Algorithm 1. Then the search for a pattern p in $\mathcal{F}_{s-1}$ can be significantly accelerated if bisection-type algorithms are used. Other types of searching algorithms which can also be used are described in [61].

Multithreading

Modern CPUs offer the possibility to run on several threads at a time in parallel. To adapt the above algorithm to this situation, we propose two steps to find a new set $\mathcal{F}_s$ of frequent s-patterns. Firstly we generate possible candidate patterns and corresponding binary vectors as shown in Fig. 3. Secondly we check if a given candidate pattern is really a frequent pattern. The reason for splitting the original algorithm is to allow all threads to have a roughly uniform load for the search and checking steps. It may happen that some events occur more often than others, therefore a new candidate $\langle p, e \rangle$ formed from a "more frequent" pattern p (with higher support values) may have a greater chance to be also a frequent pattern compared to a "less-frequent" pattern p (with values close the minimum support σ).

The simplest way to achieve multithreading is to give each thread a pattern p from $\mathcal{F}_{s-1}$ and allow it to form all various combinations $\langle p, e \rangle$ where e is a frequent 1-pattern from $\mathcal{F}_1$. Once a thread has processed all these possible patterns $\langle p, e \rangle$, then a new $(s-1)$-pattern p is taken from the ordered set $\mathcal{F}_{s-1}$ (see an example in Fig. 4 and details in Algorithm 2). As result each thread forms an ordered subset of $\mathcal{F}_s$ of candidate patterns. Due to parallel jobs we cannot put the candidate patterns in one set as this set may not be the ordered one. However, merging the resulting subsets into one can be done easily after all the threads finish their jobs because the subsets are ordered (see Algorithm 3).

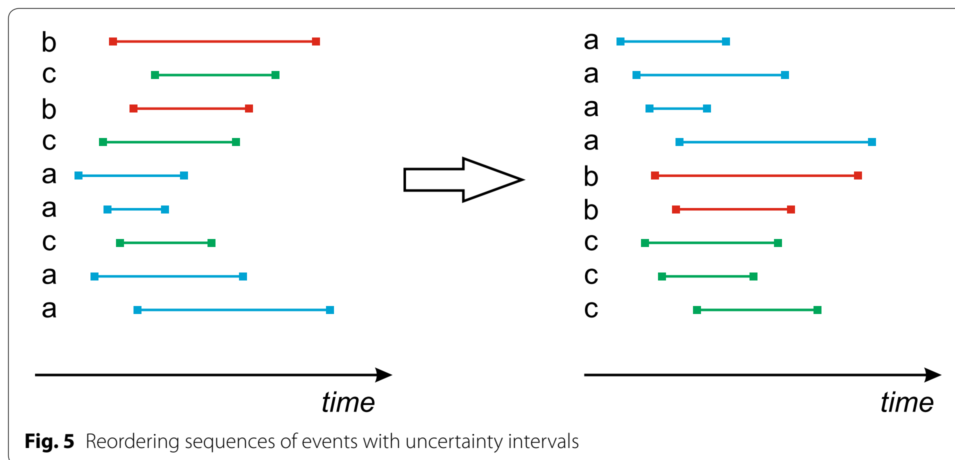


Fig. 5 Reordering sequences of events with uncertainty intervals

Once the final set of candidate patterns has been formed, the process of checking if a given pattern is a frequent one is straightforward to parallelise by giving each thread one or several patterns from the set of candidate patterns.

Events with uncertainty intervals

Suppose that each event is equally likely to happen in time interval $[t_s, t_e]$, where t_s and t_e are start and end points of the interval respectively. Thus each record becomes a list of triples $\{e^k, t_s^k, t_e^k\}$ (in the interests of brevity we denote them as E^k). We may say that a list $\{e^k, t_s^k, t_e^k\}$, $k = 1, \ldots, n$ forms an interval based sequence if there are n time points $t^k \in [t_s^k, t_e^k]$ such that $t^k \leq t^{k+1}$, $k = 1, \ldots, n-1$.

If there are two triples $E^k \equiv \{e^k, t_s^k, t_e^k\}$ and $E^m \equiv \{e^m, t_s^m, t_e^m\}$, we may order these events:

$$E^k < E^m \qquad \text{if} \qquad \left[\begin{array}{l} e^k < e^m, \\ e^k = e^m \text{ and } t_s^k < t_s^m, \\ e^k = e^m \text{ and } t_s^k = t_s^m \text{ and } t_e^k < t_e^m. \end{array}\right. \tag{5}$$

For our algorithms (FARPAM and FARPAMp) we decided to reorder all records according to the above definition. An example of such reordering is shown in Fig. 5. So 9 triples

$$\begin{array}{c} \{b, 15.6, 23.4\}, \{c, 17.2, 21.9\}, \{b, 16.4, 20.8\}, \{c, 15.2, 20.3\}, \{a, 14.3, 18.4\}, \\ \{a, 15.5, 17.6\}, \{c, 15.9, 19.4\}, \{a, 14.9, 20.6\}, \{a, 16.5, 24.0\} \end{array} \tag{6}$$

can be reordered to form $\{a, 14.3, 18.4\} < \{a, 14.9, 20.6\} < \{a, 15.5, 17.6\} < \{a, 16.5, 24.0\} < \{b, 15.6, 23.4\} < \{b, 16.4, 20.8\} < \{c, 15.2, 20.3\} < \{c, 15.9, 19.4\} < \{c, 17.2, 21.9\}$. For the implementation of the algorithm we store the following data for each record:

1. A total number of different events n (for the above example we have three different events e^j, i.e. a, b and c);
2. A list of ordered events or their indexes, i.e. (a, b, c) or (1, 2, 3);
3. An array of start times, i.e. (14.3, 14.9, 15.5, 16.5, 15.6, 16.4, 15.2, 15.9, 17.2);
4. An array of end times, i.e. (18.4, 20.6, 17.6, 24.0, 23.4, 20.8, 20.3, 19.4, 21.9);

5. An $(n + 1)$-array ξ^k of indexes to know what elements of start/end times are related to a given event (in our case it is $\xi = (1, 5, 7, 10)$; so times for the second event (which is b) are from $\xi^2 = 5$ and till $\xi^3 - 1 = 7 - 1 = 6$).

Data with the same uncertainty for alike events

Suppose a given pattern contains n *alike* events (events, which are coded by the same symbol or number) and a given record has m entries (for example, in Fig. 5 shows 4 events a, 3 events c and 2 events b). The procedure above requires us to check n permutations of m, i.e. $m!/(m - n)!$. For long records with many entries of same elements this can be very time-consuming. However, for some problems we may have extra prior information about time intervals. Suppose that for each event e any two time intervals $[t_s^i, t_e^i]$ and $[t_s^j, t_e^j]$ related to this event are ordered in the following way:

$$\text{if} \qquad \mathrm{t_s^i > t_s^j}, \qquad \text{then} \qquad \mathrm{t_e^i > t_e^j}. \tag{7}$$

For example if alike events have the same uncertainty β, then we know for sure an event takes place in the interval $[\tilde{t}^i - \beta, \tilde{t}^i + \beta]$, so if we set $t_s^i \equiv \tilde{t}_i - \beta$ and $t_e^i \equiv \tilde{t}_i + \beta$, then any two alike events can be reordered so to fulfil statement (7).

Assumption (7) leads to the fact that only permutations with $t_s^j > t_s^i$ for $j > i$ should be considered. Suppose we considered only ordered time intervals (i.e. satisfying 7) and have tried and failed to find a pattern in a record. If we permute any two entries of the alike event, then τ [as defined in (2)] for the new combination will be greater for some elements than τ found for the original combination and the requirement for $\tau \le t_e^j$ may not be true.

In the case of ordered events we need to check $m!/((m - n)!n!)$ patterns, i.e. the number of n combinations from a set of m elements. Thus we need to process $n!$ times less patterns compared to a general case. The procedure is shown in Algorithm 6.

Time restriction

For some practical problems it is important to not only find a given pattern for each client but also to be sure that all these events took place within a given time interval. This may help to exclude distant events which are not related to each other. If we set a time restriction interval ΔT, then a list $\{e^k, t_s^k, t_e^k\}$, $k = 1, \ldots, n$ forms an interval based sequence with time restriction ΔT if there are n time points $t^k \in [t_s^k, t_e^k]$ such that $t^k \le t^{k+1}$, $k = 1, \ldots, n - 1$ and $t^n - t^1 \le \Delta T$. Only small modifications of general and ordered events in Algorithms 5 and 6 are required, see for instance Algorithm 7 for ordered events.

Main algorithms used in FARPAM

Forming a list of candidate patterns

Let there be a database with n_1 records. Suppose on a previous $(s - 1)$-step we have found r frequent patterns of length $(s - 1)$. We keep these frequent patterns in $r \times (s - 1)$-matrix `allPreviousPatterns`. In fact this 2D matrix is stored as a 1D array by concatenating neighbouring rows of the matrix.

```
input  : r: number of (s − 1)-patterns found on the previous step
input  : n: number of frequent events
input  : σ: minimum support
input  : allPreviousPatterns: an array storing all (s − 1)-patterns
input  : tempPattern: an s × (s − 1)-matrix to store (s − 1)-subpatterns of a temporary pattern
         for the given thread
input  : indexOfPreviousPatterns: an s-array to store position of (s − 1)-subpatterns
input  : binPrevious: an array with all binary vectors found on the previous step
input  : binTemp: a temporary array to store binary vectors for the given thread
output: q: number of candidate patterns found for the given thread
output: binCandidate: an array with all new binary vectors found for the given thread
output: candPattern: an array with candidate patterns found for the given thread
q ← 0;
for i from 1 to r do
    /* all numbers from 1 to r are distributed between all threads                */
    /* Prefilling tempPattern array                                                */
    copy i-th (s − 1)-array from allPreviousPatterns to the last row of tempPattern;
    for k from 1 to s − 1 do
        copy 1, . . . , k − 1, k + 1, . . . s − 1 elements of the last row of tempPattern to 1, . . . s − 2
        elements of k-th row respectively;
    end
    /* by the above procedure we have filled in all elements of tempPattern matrix
       except last in row elements for the top s − 1 rows                          */
    for j from 1 to n do
        /* considering all frequent events                                         */
        set all last in row elements (for the top (s-1) rows) of tempPattern to j;
        for m from 1 to s do
            indexOfPreviousPatterns[m] ← position of m-th row of tempPattern within
            allPreviousPatterns;
            if indexOfPreviousPatterns[m] = 0 then
                /* the next frequent element to be considered                      */
                break m-loop and go the j-loop;
            end
        end
        copy indexOfPreviousPatterns[1]-th binary vector from binPrevious to binTemp;
        for m from 2 to s do
            binTemp ← bitwise AND for binTemp and indexOfPreviousPatterns[m]-th binary
            vector from binPrevious;
            n_b ← numberOfNonZeroBits(binTemp);
            if n_b < σr then
                break m-loop and go the j-loop;
            end
        end
    end
    /* The new candidate pattern is found                                          */
    q ← q + 1;
    save the last row of tempPattern as the q-th pattern of candPattern;
    save binTemp to q-th vector of binCandidate;
end
```

Algorithm 2: Forming a list of candidate patterns (parallel implementation)

For each pattern we keep a binary vector of length n_1, so each for each i-th record the corresponding element of the binary vector is 1 when the given pattern is present in the record, otherwise we set it to 0. In order to exploit SIMD features of modern processors it is better to assume that the number n_1 of records is divisible by 32. We may always do so by adding extra (empty) records in the database. If we denote $n_{32} = n_1/32$, then each binary vector can be stored as an n_{32}-vector of 32-bit unsigned integer numbers. We place all binary vectors for r patterns in an array `binPrevious` of 32-bit unsigned integer numbers. The size of the array is $n_{32} \times r$.

Now we want to find all possible candidate patterns of length s. For this purpose we take each frequent patterns of length $(s - 1)$ and add an extra event to it. In order to the new s-pattern to be frequent all its s subpatterns of length $(s - 1)$ should also be

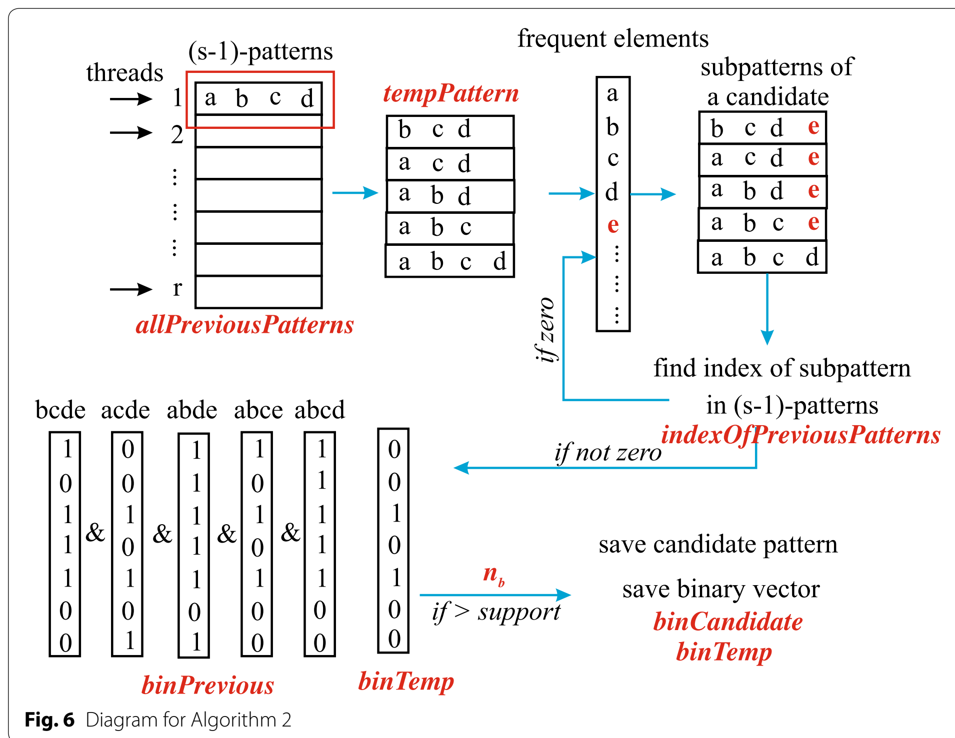


Fig. 6 Diagram for Algorithm 2

frequent. Therefore we create a template $s \times (s-1)$-matrix with each row consisting of elements of those $(s-1)$-subpatterns. In this matrix only the last element of first $(s-1)$ rows are changed when we vary the extra event.

Once the template matrix is filled in for a given extra event, we check if all rows of the matrix are among the frequent patterns found on the previous step. If at least one of the patterns is missed, then the new s-pattern cannot be frequent and we should take another extra element and reiterate the procedure. In case of all rows of the matrix to be frequent, we find the corresponding binary vectors in `binPrevious` array. Those s binary vectors are logically multiplied and the new binary vector for the candidate s-pattern is formed. By counting the number of non-zero elements of the new binary vector we check if it is more or equal to σr where σ is the minimum support. Otherwise the candidate pattern cannot be frequent.

If there are several threads available, then all r patterns from the previous steps can be processed concurrently, e.g. with `parallel for` loop from OpenMP. The algorithm and its example diagram are presented in Fig. 6 and Algorithm 2.

We store $(s-1)$-subpatterns to form a candidate pattern in $s \times (s-1) \times t$-matrix `tempPattern`, where t is the number of threads. Every subpattern of a candidate can be described by its position in `allPreviousPatterns`. We store these positions in matrix `indexOfPreviousPatterns`. The size of matrix is $t \times s$. When calculating the binary vector for a candidate we store it in `binTemp`. The size of it is n_{32}. After doing all the steps in Algorithm 2 we store the binary vector of a candidate and the candidate pattern itself in matrices `binCandidate` and `candPattern`. The size of `binCandidate` is $n_{32} \times 100,000$. Size of `candPattern` is $(s-1) \times 100,000$. We chose 100, 000 as an upper limit of maximum number of candidates (can be changed if it is needed).

Merging candidate patterns

Suppose that *n* OpenMP threads are used and *m* frequent candidate *s*-patterns are found. Each thread stores all candidate patterns it has found in its own block of memory. In principle, the corresponding binary vectors can be stored in a common block of memory thus avoiding extra data copying. In our algorithm we prefer to store all patterns as a lexicographically ordered list. This type of storage allows us to check in a faster way if a given pattern belongs to the list. Otherwise one needs to check all patterns in the list when searching for $(s-1)$-patterns. As each thread processes $(s-1)$-patterns from the ordered list by concatenating with single events also from the lexicographically ordered list of 1-patterns, then the candidate patterns found by this thread will also form a lexicographically ordered list of *s*-patterns. Thus we just need to merge *n* ordered lists found by *n* threads.

```
input  : s: size of patterns
input  : n: number of arrays of s-patterns
input  : v_k: the k-th ordered array of s-patterns
input  : q_k: size of v_k
input  : Ψ: a large integer number (more than the number of frequent events)
output: w: output ordered array of s-patterns
output: m: size of the output array
m ← 0;
for k from 1 to n do
    m ← m + q_k; η_k ← 1;          /* index of the current pattern in the k-array */
end
for p from 1 to m do
    μ ← 0;
    for i from 1 to s do
        u^i ← Ψ;     /* set all elements of the ‘‘smallest’’ pattern to the largest
        number */
    end
    for j from 1 to n do
        if η_j > q_j then
            /* all elements from the j-th array have been considered; we should
               consider the next array                                          */
            continue;
        end
        isBetter ← true;
        β ← the pointer to the q_j-th pattern of the j-th array;
        for l from 1 to s do
            if β^l > u^l then
                isBetter ← false;
                break;
            end
            if β^l < u^l then break;
        end
        if isBetter = false then continue;
        μ ← j;
        for l from 1 to s do
            u^l ← β^l;
        end
    end
    add u to the array w;
    η_μ ← η_μ + 1;
end
```

Algorithm 3: Merging n ordered lists of s-patterns without common s-patterns

We define an array of frequent patterns found by a single OpenMP k-th thread as v_k and the corresponding size of the array as q_k. Algorithm 3 shows how found frequent patterns are merged into output array ω of frequent s-patterns (all resultant patterns become sorted).

The total number of candidate patterns is $m = \sum_{k=1}^{n} q_k$. For each n list of ordered patterns we store the index of the smallest pattern (not yet merged to the final list), we denote it as η_k and set to 1 initially.

We introduce a working s-vector u. For each new ordered s-pattern to be found from the given n lists we initially set all elements of u to Ψ (a number larger than the number of events). Then for each n lists we consider smallest patterns not yet merged, i.e. η_k-th pattern for the k-th list. By comparing the working vector u with corresponding n smallest patterns we find the index μ of the smallest of them. Then we put the found pattern to the merged list and increment η_μ by 1.

Note that due to the process of formation of candidate patterns each n lists of patterns does not have any common s-patterns with the other $n-1$ lists. This allows us to slightly reduce the number of checks compared to a general case of merging patterns with possibly common patterns.

Checking if a subpattern belongs to a record (initialisation step)

The Algorithm 4 shows the initialisation part of a searching function (used in FARPAM). Each event is defined as a triple of event index, start and end times. Suppose a record consists of m events. All those events can be sorted according to rule (5). Therefore we get n distinctive events ($n \leq m$) and two m-arrays for the start/end times of the events. We may introduce an $(n+1)$-vector of start positions for the events, i.e. the data related to the first distinctive event are for indices from ξ^1 till $\xi^2 - 1$. Then for each i-th element of the given pattern p we find its first and last occurrence within the record and denote them as `vFirst[i]` and `vLast[i]` respectively. This means that for that element p^i we should only check start/end times for the indices between `vFirst[i]` and `vLast[i]`. For example, we have a record *aabbbc* and want to find patterns {*abbcb*} in it, then we get `vFirst = {1,3,3,6,3}`, `vEnd = {2,5,5,6,5}`.

```
input : p: the given s-pattern
input : n: number of distinctive events in a record
input : q: an array of the distinctive events
input : ξ^k: an array of start positions for all events related to q^k event
input : t_s/t_e: an array of start/end times for each event
output: true/false: true if the given pattern is found
for i from 1 to s do
    τ ← false;
    for j from 1 to n do
        if p^i ≠ q^j then continue;
        τ ← true;
        vFirst[i] ← ξ^j; vLast[i] ← ξ^{j+1} − 1;
        /* start/end positions of data related to p^i event in the given record */
        break;
    end
    if τ = false then return false;
end
/* setting the start position for each event */
for i from 1 to s do
    vCurrent[i] ← vFirst[i];
end
/* finding the index of same previous/next events if they exist, otherwise 0 */
for i from 1 to s do
    previousIndex[i] = 0;
    nextIndex[i] = 0;
end
for i from 2 to s do
    for j from i − 1 to 1 by −1 do
        if vFirst[i] = vFirst[j] then
            previousIndex[i] = j; nextIndex[j] = i; break;
        end
    end
end
/* in case of several same events we need to adjust start positions */
for i from 2 to s do
    if previousIndex[i] = 0 then continue;
    vCurrent[i] = vCurrent[previousIndex[i]] + 1;
    if vCurrent[i] > vEnd[i] then return false;
end
```

Algorithm 4: Checking if a given pattern is a subpattern for a given record (initialisation step)

We aim to consider all possible combinations of events. For the event p^i we keep its position within the record as an index `vCurrent[i]` which can have values between `vFirst[i]` and `vLast[i]`. It is clear that in case of two same events p^i and p^j they cannot point to the same event in the record. Therefore we want to avoid cases when two positions `vCurrent[i]` and `vCurrent[j]` are the same. Therefore in the example above we cannot set `vCurrent = {1,3,3,6,3}` as the second, third and fifth events point to the third element. Thus for the initialisation step in this algorithm for each element we first find previous/next alike elements `previousIndex` and `nextIndex` if they exist and reset current values based on values of previous elements. It is clear that the index `vCurrent[i]` should not exceed the index of the last alike element. The algorithm takes into account cases when there are more alike events in a given pattern than in a record (and returns `false`).

Consider an example. Suppose we search for pattern $\langle c, c, c \rangle$ in the record from Fig. 5. We aim to check all possible patterns. We get `vFirst` $= \{7, 7, 7\}$ and `vLast` $= \{9, 9, 9\}$. During the initialisation step shown in Algorithm 4 we get `vCurrent` $= \{7, 8, 9\}$. Let us see what possible sequences of vectors `vCurrent` should be considered during the iteration step.

- Initial values: $\{7, 8, 9\}$.
- Now we increment the third element by one and get $\{7, 8, 10\}$, this element is greater than `vLast[3]` $= 9$, so we set $\{7, 9, 7\}$. The first and the last elements are the same, so we set $\{7, 9, 8\}$.
- By incrementing the third element by one we get $\{7, 9, 9\}$, so by correcting out-of-range and identical values we get this procedure $\{7, 9, 9\} \rightarrow \{7, 9, 10\} \rightarrow \{7, 10, 7\} \rightarrow \{8, 7, 7\} \rightarrow \{8, 7, 8\} \rightarrow \{8, 7, 9\}$.
- In a similar way $\{8, 7, 10\} \rightarrow \{8, 8, 7\} \rightarrow \{8, 9, 7\}$.
- $\{8, 9, 8\} \rightarrow \{8, 9, 9\} \rightarrow \{8, 9, 10\} \rightarrow \{8, 10, 7\} \rightarrow \{9, 7, 7\} \rightarrow \{9, 7, 8\}$.
- And the final vector $\{9, 7, 9\} \rightarrow \{9, 7, 10\} \rightarrow \{9, 8, 7\}$.

So we have to check $3!/(3-3)! = 3!/0! = 6/1 = 6$ patterns.

Of course, checking all $m!/(m-n)!$ is the worst case scenario and is not always the case (since all our events are rearranged according to rule 5). However, pattern $\langle cacc \rangle$ in the record

$$\{a, 5, 6\}, \{c, 1, 8\}, \{c, 2, 7\}, \{c, 3, 4\}$$

will require to check all the possible combinations, starting from `vCurrent` $= \{2, 1, 3, 4\}$, and ending with `vCurrent` $= \{4, 1, 3, 2\}$ as a solution.

Checking if a subpattern belongs to a record (iteration step)

For the second (iteration) step we suppose a valid combination of `vCurrent[i]` is given, i.e. `vCurrent[i]` $\neq$ `vCurrent[j]` for any $i \neq j$ (for example, `vCurrent` $= \{1, 1, 1\}$ is not valid while `vCurrent` $= \{1, 3, 2\}$ is). Let us have a record with $E^i \equiv \{e^i, t^i_s, t^i_s\}$ and $E^j \equiv \{e^j, t^j_s, t^j_s\}$, and want to check if pattern (e^i, e^j) can be a subpattern in the record. We need that $t^i \leq t^j$ where $t^i \in [t^i_s, t^i_e]$ and $t^j \in [t^j_s, t^j_e]$. This may happen only if $t^i_s \leq t^j_e$. The minimum possible value of t^i is then t^i_s and t^j may vary from $\max(t^i_s, t^j_s)$ to t^j_e. So if we introduce $\tau = t^1_s$ for the first element in the pattern, then we should check that for each following element of the pattern $\tau \leq t^j_e$ and re-define τ as $\max(\tau, t^j_s)$. If we succeed to fulfil these requirements for a given s-vector of indexes `vCurrent[i]`, then the subpattern is found, otherwise we need to check the next permitted combination of indexes `vCurrent[i]`, see Algorithm 5.

```
repeat
    τ ← vStartTime[vCurrent[1]]; J ← 0;
    for i from 2 to s do
        I ← vCurrent[i];
        if τ > vTimeEnd[I] then
            J ← I; break;
        end
        τ ← max(τ, vStartTime[I]);
    end
    if J = 0 then return true;
    i_s ← vFirst[J]; i_e ← vLast[J];
    /* create a list of previous alike events                    */
    u ← previousIndex[J]; numberOfPreviousEvents ← 0;
    while u > 0 do
        poistionOfPrevious[numberOfPreviousEvents] ← vCurrent[u];
        numberOfPreviousEvents ← numberOfPreviousEvents + 1;
        u ← previousIndex[u];
    end
    N ← 1;
    repeat
        b = false;
        vCurrent[J] ← vCurrent[J] + 1;
        if vCurrent[J] > i_e then
            if previousIndex[J] = 0 then return false;
            J ← previousIndex[J]; N ← N + 1; b ← true;
        else
            for i from N to numberOfPreviousEvents do
                if vCurrent[J] = positionOfPrevious[i] then
                    b ← true; break;
                end
            end
        end
    until b = false;
    u ← previousIndex[J];
    N ← 1;
    positionOfPrevious[1] ← vCurrent[J];
    while u > 0 do
        positionOfPrevious[N] ← vCurrent[u];
        N ← N + 1; u ← positionOfPrevious[iu];
    end
    u ← nextIndex[J];
    while u > 1 do
        vCurrent[u] ← i_s;
        repeat
            b ← false;
            for i from 1 to N do
                if vCurrent[u] = poistionOfPrevious[i] then
                    vCurrent[u] ← vCurrent[u] + 1; b ← true;
                    break;
                end
            end
        until b = false;
        positionOfPrevious[N] ← vCurrent[u];
        N ← N + 1; u ← nextIndex[u];
    end
until true;
return false;
```

Algorithm 5: Checking if a given pattern is a subpattern for a given record (iteration step)

Ordered events

Suppose we have prior information for alike events, i.e. they are ordered according to (7). Then the corresponding algorithms can be simplified as it was discussed in "Data with the same uncertainty for alike events" and "Time restriction" section, see Algorithms 6 and 7.

```
/* finding the index of the same previous event if it exists, otherwise 0 */
for i from 1 to s do
    previousIndex[i] ← 0;
end
for i from 2 to s do
    for j from i − 1 to 1 by −1 do
        if vCurrent[i] = vCurrent[j] then
            previousIndex[i] ← j; break;
        end
    end
end
τ ← vTimeStart[vCurrent[1]];
for i from 2 to s do
    b ← false;
    if previousIndex[i] > 0 then vCurrent[i] ← vCurrent[previousIndex[i]] + 1;
    repeat
        J ← vCurrent[i];
        if vTimeEnd[J] ≥ τ then
            τ ← max(τ, vTimeStart[J]);
            b ← true; break;
        end
        vCurrent[i] ← vCurrent[i] + 1;
        if vCurrent[i] ≤ vLast[i] then break;
    until b = true;
end
return true;
```

Algorithm 6: Checking if a given pattern is a subpattern for a given record (iteration step) for ordered events

```
while vCurrent[1] ≤ vLast[1] do
    τ1 = vTimeStart[vCurrent[1]];
    τ2 = vTimeEnd[vCurrent[1]];
    for i from 2 to s do
        vCurrent[i] ← vFirst[i];
    end
    for i from 2 to s do
        b ← false;
        if previousIndex[i] > 0 then vCurrent[k] ← vCurrent[previousIndex[i]] + 1;
        while vCurrent[i] ≤ vLast[i] do
            J ← vCurrent[i];
            if vTimeEnd[J] > τ1 then
                τ1 ← max(τ1, vStartTime[J]);
                τ2 ← min(τ2, vEndTime[J]);
                t ← true;
                break;
            end
            vCurrent[i] ← vCurrent[i] + 1;
        end
        if τ1 > τ2 + ΔT then break;
        if b = false then return false;
        if i = s then return true;
    end
    vCurrent[1] ← vCurrent[1] + 1;
end
return false;
```

Algorithm 7: Checking if a given pattern is a subpattern for a given record (iteration step) for ordered events with time restriction

Evaluation

The algorithms have been evaluated on two datasets: an adult social care database (data provided by a large local authority) and a weather temperature measurements dataset (open access).

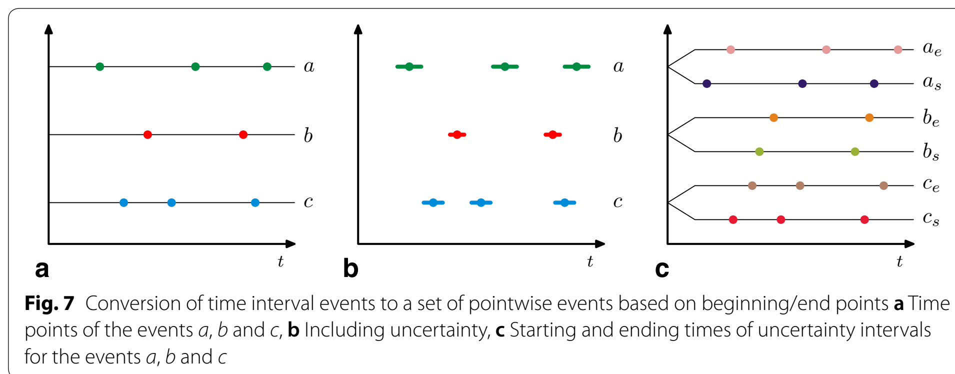


Fig. 7 Conversion of time interval events to a set of pointwise events based on beginning/end points **a** Time points of the events *a*, *b* and *c*, **b** Including uncertainty, **c** Starting and ending times of uncertainty intervals for the events *a*, *b* and *c*

Adult social care database

The database consists of approximately 100,000 adult social care records over a period of 15 years (also used in [37]). The records cannot be processed outside the secure facilities of Leeds Institute for Data Analytics (LIDA) at the University of Leeds. After removing records with fewer than three events and selecting people within a certain age range the dataset was reduced to $\approx 25,000$ records. All the events have been categorised and coded, where each code (event label) represents one of the following four event types: a referral to adult social care, an assessment, a service (including reablement activity) or a review. Referral event codes are composed of three parts: source (who made the referral)—2 categories; reason (why the service is needed)—12 categories and outcome (decision for assessment)—15 categories. The assessment activity is coded using only one variable; eligibility with 16 categories. The service activity is coded with 116 categories and the review activity with 17 categories. The final number of unique codes (unique existing combinations of categories for all codes) is 301.

We consider an event of interest to be the use of a form of intensive, high cost care provision, a permanent residential or nursing placement, called for simplicity "Expensive Care" (EC in short), with the alternative event being called "Non Expensive Care" (NEC), which we use to split the clients into two groups: EC, with 6128 clients with 111,736 events in total and 252 unique events and NEC, with 18,518 clients, 286,201 total number of events, 258 unique events). It is clear that the average number of events per client in the EC group is much higher and we expect that the two groups will provide an interesting test bed for the performance of our algorithms. The output from our algorithm can be further used for risk stratification (see [37] for initial results using RobustSpam).

Each event is an interval that is represented with a starting time point and ending time point (together with the event label as discussed in "Methodology" section, see Fig. 7), the length of the interval varies and is even reduced to single points on some occasions. We allow an uncertainty β on the time stamp of the starting and ending point events. β could depend on the type of event (for example, services could be recorded with better accuracy than referrals) although here we consider uniform uncertainty. In principle, a frequent pattern mining algorithm can use prior information as in (7). However, in order to illustrate how prior information

can improve performance we apply both the general and ordered versions of the algorithm to this dataset (Algorithms FARPAM and FARPAMp correspondingly).

Weather dataset

To see how performance of the proposed algorithm may depend on various hardware parameters we decided to use a publicly available dataset. The European Commission provides access to weather data via Agri4Cast Resources Portal of the Joint Research Centre (http://agri4cast.jrc.ec.europa.eu). Gridded Agro-Meteorological database contains meteorological parameters from weather stations interpolated on a 25×25 km grid. Meteorological data are available on a daily basis from 1975 to the last calendar year completed, covering the EU Member States, neighbouring European countries, and the Mediterranean countries. The following variables can be accessed:

- Maximum/minimum/mean temperature,
- Mean daily wind speed at 10 m (m/s),
- Vapour pressure (hPa),
- Sum of precipitation (mm/day),
- Potential evaporation from a free water/crop canopy/moist bare soil surface (mm/day),
- Total global radiation (kJ/m^2/day),
- Snow depth.

We chose a mean daily temperature to generate patterns according to the procedure described in "Temporal data" section and shown in Fig. 2. As temperature has seasonal variation we consider its first derivative $T'(t)$. Suppose that for each grid point where data were interpolated from weather stations we have chosen n_{level} levels T'_j, $j = 0, \ldots, n_{level-1}$. Then j-th event happens at a time τ_k when $T'(\tau_k) = T'_j$ and $T''(\tau_k) \geq 0$ while $(j + n_{level})$-th even happens if $T'(\tau_k) = T'_j$ and $T''(\tau_k) < 0$. So for each grid point we have a sequence $2n_{level}$ events. We chose T'_j values in such a way, so $T'(t) \leq T'_j$ for $(j+1)t_{max}/n_{level}$, $j = 0, \ldots, n_{level-1}$, time where t_{max} is the total period of time the given variable is known (we used 20 years). Note that each grid point has its own values for T'_j.

The weather data allows us to form various datasets. If we choose places in a relatively small region (one country like the UK), then we should expect a high level of correlation between temperature rise and fall for neighbouring places, thus we expect to have a lot of patterns even for large values of minimum support, e.g. $\sigma = 0.99$. On the other hand, temperature variations in the Mediterranean and Baltic countries may often behave independently. At the same time we may also control the length of patterns we aim to find. Each record corresponds to events that took place within a given time period. It is clear that during a 5-day period we get fewer events compared to a 25-day period. In this way we may control the maximum length of patterns for the given minimum support level.

The aim of the paper is not to provide a comprehensive study related to possible weather (or adult social care) datasets but to use them as test data to show performance improvements of the proposed algorithms.

Table 3 Run times (in seconds) for algorithms with zero uncertainty $\beta = 0$ for the weather dataset (L3-D3-T14) measured over 14 places in the UK

Support	Max length	No. patterns	Apriori	Apriori + bitmap	SPAM	FARPAM	FARPAMp
0.5	3	8332	120.1	18.7	14.7	1.19	1.21
0.4	4	46,848	5942.1	51.9	50.4	3.66	3.87
0.3	5	157,536	7519.8	120.4	219.4	7.80	7.85

Hardware

Two workstations were used to get performance results:

- WS4. CPU: Intel Core i7-4790 (codename Skylake, 4 cores, processor base frequency 3.6 GHz), RAM: 16 GB, operating system: 64-bit Windows 8.1. This workstation is a part of LIDA of University of Leeds where the private social care records can be processed.
- WS6. CPU: Intel Core i7-3930K (codename Sandy Bridge E, 6 cores, processor base frequency 3.2 GHz), RAM: 32 GB, operating system: 64-bit Windows 10. This workstation was used to process the weather dataset.

All codes were compiled with an Intel C++ compiler (part of Intel Parallel Studio XE 2016), in release mode, with maximum optimisation (favour speed, /O2 flag). For multithreaded versions of the codes, OpenMP was used.

Results

We evaluate the efficiency of the proposed optimisation by comparing results with other existing algorithms where possible as well as by varying some parameters of the problems.

Sequential pattern mining

If the uncertainty parameter β is set to zero (and with the assumption that for each record no two events happen at the same time), then the problem becomes a classical sequential pattern mining problem. Therefore we are able to compare results and performance of our optimised algorithms with other publicly available sequential pattern mining algorithms. We decided to use a SPAM code from SPMF (http://www.philippe-fournier-viger.com/spmf/, an open-source data mining library written in Java, [62]). This is considered to be one of the most efficient codes for sequential pattern mining problems and according to [8] it outperforms such algorithms as SPADE and PrefixSpan. Our codes are written in C and compiled with an Intel C compiler. Of course, it is not fully correct to compare codes written in different languages, however our aim is simply to provide the reader with an idea of possible improvements. It is likely that direct conversion (without any optimisation) of a Java code to C or Fortran languages will provide a user with shorter run times (both implementations should have similar dependence on parameters of a problem, e.g. number of patterns or minimum support). In order to give a more realistic comparison for the C code we have implemented a naive version of the *Apriori*-like algorithm with bitmaps similar to the method in [8].

Table 4 Run times (in seconds) for algorithms with zero uncertainty $\beta = 0$ for expensive and non-expensive datasets

EC/NEC	Support	Max length	No patterns	SPAM	Apriori + bitmap + OMP	FARPAM	FARPAMp
NEC	0.10	7	491	1.0	1.6	0.42	0.29
NEC	0.05	9	3414	3.6	3.8	1.45	0.47
NEC	0.03	11	15,413	9.3	12.0	7.60	0.84
NEC	0.02	12	51,413	20.9	36.2	3.47	1.82
NEC	0.01	14	427,239	88.5	301.0	21.92	10.16
EC	0.10	8	3416	1.3	1.7	0.56	0.24
EC	0.05	11	32,507	5.9	5.1	3.00	0.54
EC	0.03	12	163,949	18.5	27.3	9.73	1.67
EC	0.02	14	567,200	43.2	107.9	18.59	4.56
EC	0.01	16	479,3176	191.1	–	67.45	32.77

Table 3 demonstrates the difference in times between the naive Algorithm 1, its bitmap version and the SPAM Java function. For the illustration we used the L3-D3-T14 weather dataset. The dataset was abstracted from data collected at 14 places in the UK (roughly uniformly distributed) and is for 3 levels for the first derivative of temperature, each record is for events within a 3-day period. The corresponding results for the social care data (non-expensive and expensive care datasets) are shown in Table 4. For both tables FARPAM stands for the optimised version of the proposed algorithm with no assumption for robustness intervals for alike events, while FARPAMp is for the case when those intervals can be ordered according to statement (7).

While the relative performance of the two proposed algorithms depends on the datasets they are applied to, the main result is that the proposed algorithms outperform the SPAM code: the times for the new algorithms are sufficiently less (roughly 15 times faster for weather datasets and 4 times faster for the social care dataset). For the weather dataset the relative improvement increases with the number of patterns mined. For the adult social care dataset the improvement decreases at the largest number of patterns mined, but is still significant. This is maybe partly due to different programming languages used, but also to multithreading and differences in the algorithms. FARPAMp gives a significant improvement for the adult social care dataset, but not for the weather dataset. This is because the weather dataset does not have many repeated events. However we expect that if the Java version is directly translated into its C version (without any further optimisation) the ratio of run times may decrease, but dependence on the number of patterns should not change very much. We can see that our approaches can be used even in the case of sequential pattern mining and we have some sort of linear behaviour for run times provided by SPAM and the proposed algorithms. The algorithms proposed in the paper deal with uncertain timestamps thus solve an inherently harder problems than SPAM. Hence, seeing them running faster than SPAM is one of their advantages.

Non-zero uncertainty

In case of non-zero uncertainty β, the SPAM algorithm cannot be used, however we may consider the RobustSPAM algorithm introduced in [37] and written in Java. The results shown in Table 5 demonstrate a dramatic reduction in run times by using the new approach.

Table 5 Run times (in seconds) for algorithms with uncertainty $\beta = 60$ for expensive and non-expensive datasets, pattern lengths 3, 4 and 5

EC/NEC	Support	RobustSPAM	Apriori	Apriori + bitmap + OMP	FARPAM	FARPAMp
NEC	0.10	715.9	20.9	2.3	0.74	0.40
NEC	0.05	2370.7	91.7	8.1	1.06	1.05
NEC	0.03	5984.1	256.3	14.1	1.66	1.58
NEC	0.02	13,045.1	602.2	26.8	2.30	1.88
EC	0.10	355.6	10.4	1.4	0.39	0.25
EC	0.05	1160.6	46.8	2.7	0.48	0.50
EC	0.03	2908.1	128.5	5.1	0.76	0.58
EC	0.02	6189.3	275.1	10.0	1.05	0.85

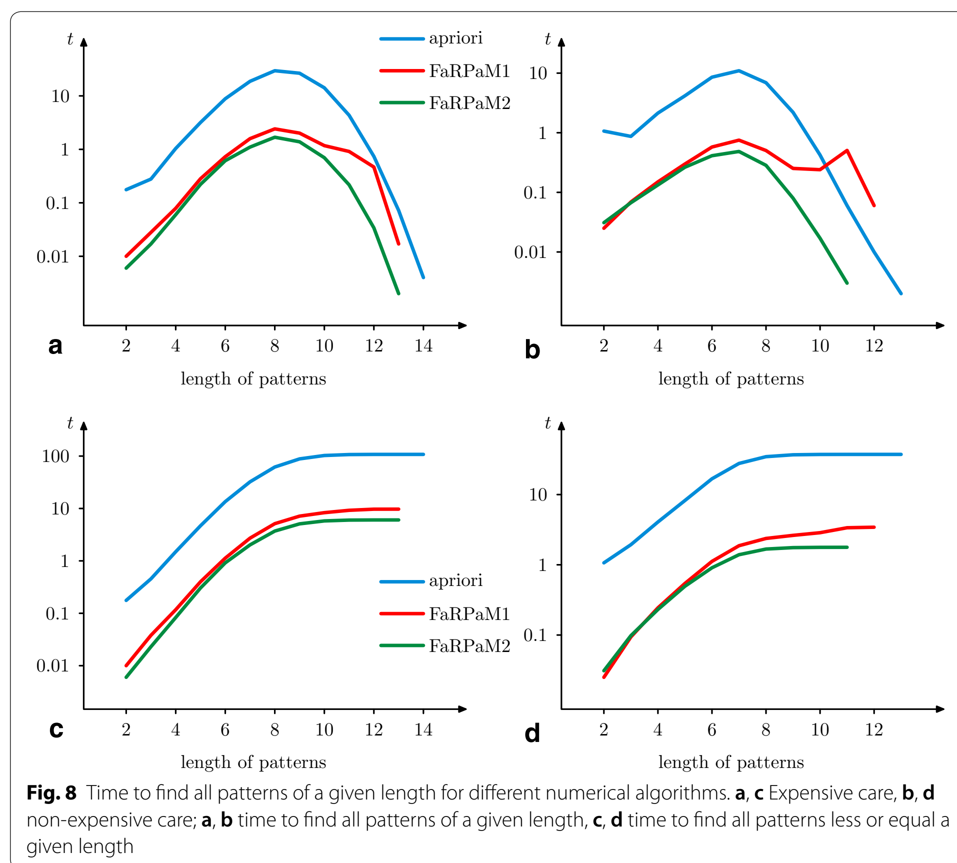


Fig. 8 Time to find all patterns of a given length for different numerical algorithms. **a**, **c** Expensive care, **b**, **d** non-expensive care; **a**, **b** time to find all patterns of a given length, **c**, **d** time to find all patterns less or equal a given length

It is also interesting to see how the new approaches perform for test datasets. For a given minimum support σ one may try to find patterns of all possible lengths. Thus we may plot run time as a function of maximum pattern length. Due to a limited number of events for each record the maximum length of a pattern is also limited. In Fig. 8 we show results found with the naive *Apriori* + bitmap + OpenMP approach and the two proposed algorithms. One may see that for the social care datasets NEC and EC curves obtained with the naive approach and the proposed approach for ordered uncertainty intervals for alike events (FARPAMp) behave in a similar way even though the second

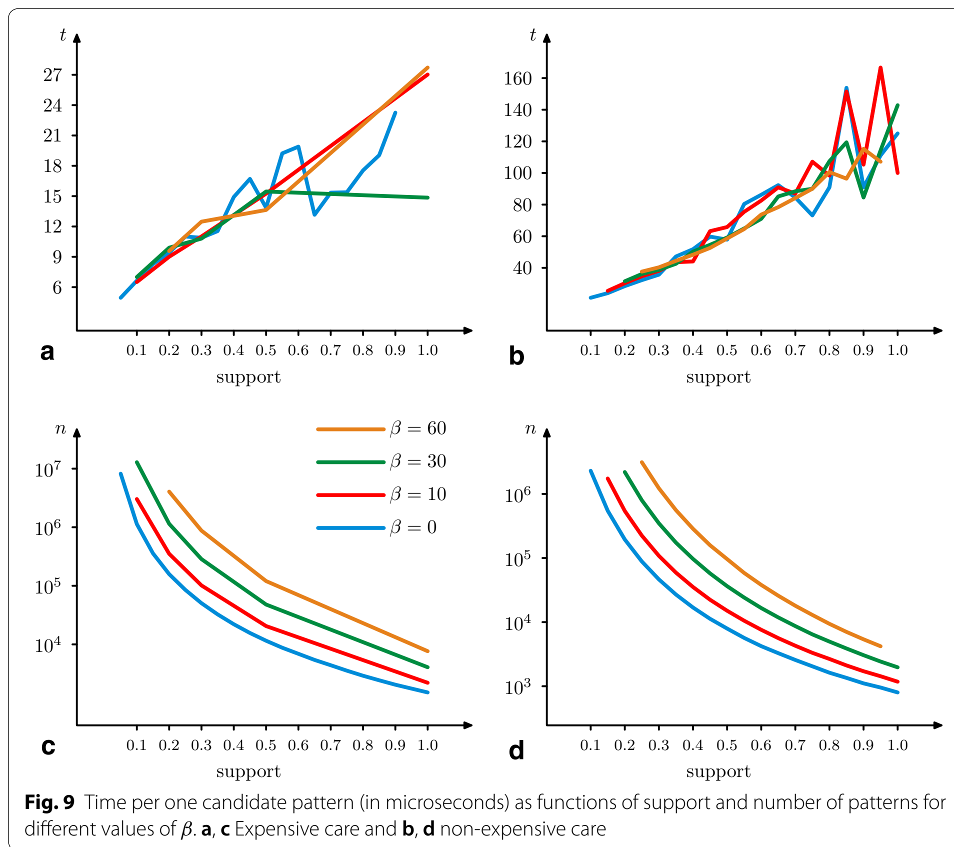


Fig. 9 Time per one candidate pattern (in microseconds) as functions of support and number of patterns for different values of β. **a**, **c** Expensive care and **b**, **d** non-expensive care

approach works much faster. On the other hand, if prior information related to alike events is not embedded in the algorithm for longer patterns (FARPAM) and small run times, the performance of the general approach may in some cases be even slower than the naive one. This is due to the fact that the general approach has to process more possible pattern combinations and is not allowed to discard some of them. Thus it is clear that if such prior information exists one needs to embed it in the algorithm.

Our algorithms can readily be used to provide a sensitivity analysis by varying the uncertainty factor beta and the level of support sigma. For example, the uncertainty parameter β may sometimes be known or roughly estimated from data collection procedures. However, in many cases we may need to vary its value within a range and see if we get meaningful results. Ideally we want to estimate the number of patterns we may find for a given support σ and uncertainty β. In Fig. 9 curves found for different values of β behave in a similar way. Thus we may find a number of patterns for a relatively big support, e.g. $\sigma = 0.7$, for a range of β, then find the number of patterns for smaller support for one value of β and based on the values found we may estimate the number of patterns for various values of β for a given level of support.

Time restrictions

The algorithms we propose here can also be used when additional time restrictions are given, so all events in a pattern should take place within a given time interval Δt, see "Multithreading" section. In Fig. 10 we show results found for the weather dataset.

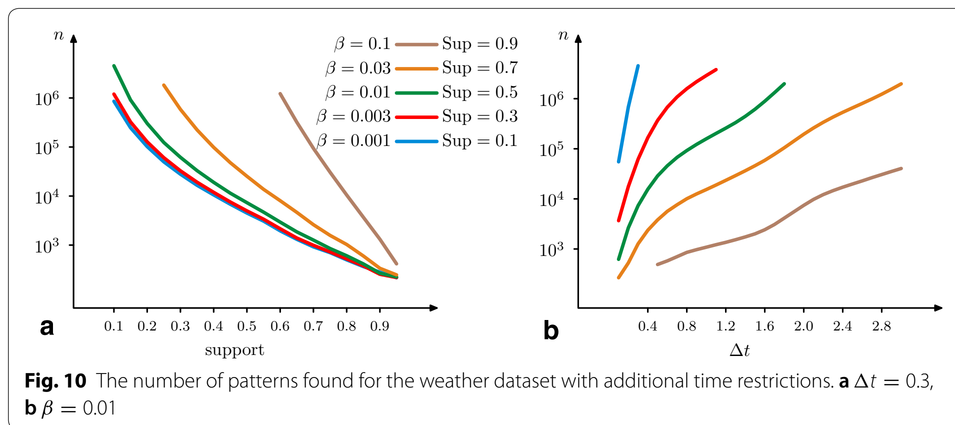


Fig. 10 The number of patterns found for the weather dataset with additional time restrictions. **a** $\Delta t = 0.3$, **b** $\beta = 0.01$

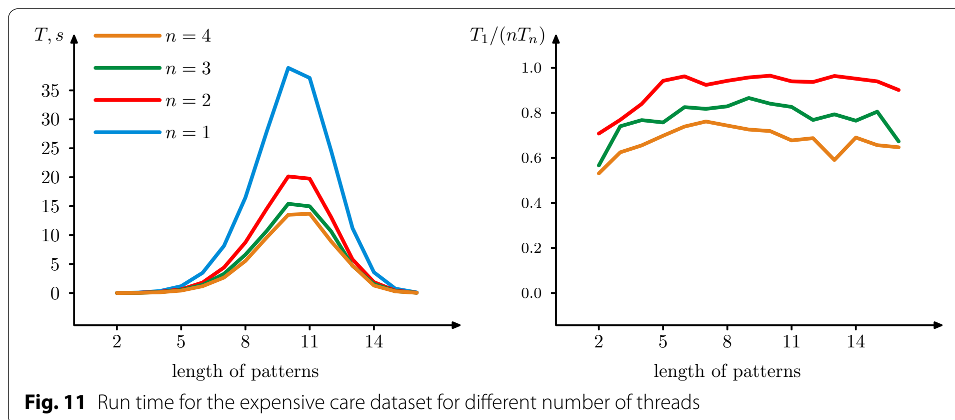


Fig. 11 Run time for the expensive care dataset for different number of threads

By setting Δt to 0.3 we plot the number of patterns found for various uncertainties β, see Fig. 10a. In the case of small β the results are not very sensitive to values of β as the number of patterns found by the algorithm tends to be similar. On the other hand, when values of uncertainty β become of the same order as the time restriction Δt we see a dramatic increase of the number of patterns when we increase uncertainty β. For Fig. 10b we fix the uncertainty and show how the number of patterns increases with Δt for given support values.

Multithreading

Absolute numbers for run times of algorithms are important, however it is also crucial to see how an algorithm behaves in a multithreading environment. The current tendency in CPU development is to increase the number of threads able to run parallel jobs. We need to be sure an approach is scalable in the ideal case, run times are inversely proportional to the number of threads used by an application. Figures 11 and 12 show performance of our algorithm in the multithreading environment. There is certainly room for further optimisation of the codes to achieve better results, however this may require low level programming and taking into account the hardware parameters of an existing CPU. We

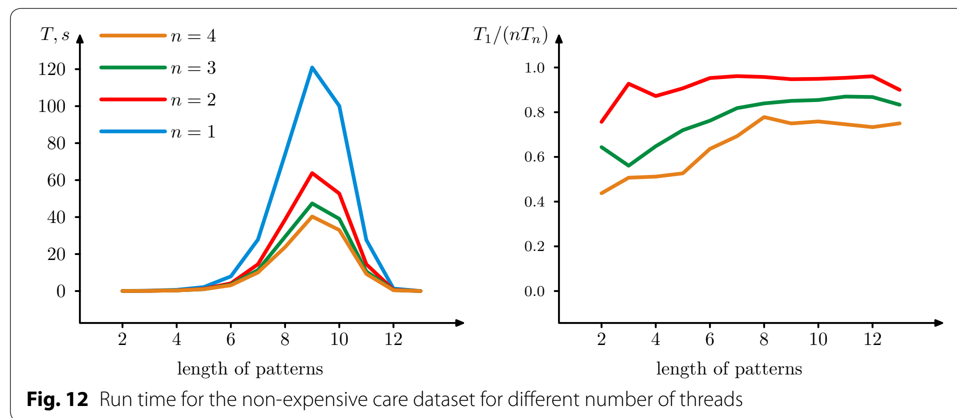


Fig. 12 Run time for the non-expensive care dataset for different number of threads

do not want to be bound to a particular CPU, so that we can easily transfer the codes to new CPU architectures in the future.

Discussion

We have proposed several approaches to improve performance of algorithms for pattern mining with uncertainty and time restrictions. For each pattern we store a binary vector to find if the pattern is a subpattern of a given record. These binary vectors are stored in a compressed form, so information related to every 32 records can be stored as a 32-bit number. This not only allows us to efficiently use available memory, but also to speed calculations as only one CPU instruction needs to be used to find a candidate binary vector for a pattern containing any two given patterns. This approach may also be further extended for datasets with many records, e.g. hundreds of thousands, if SIMD CPU instructions for modern processors are used. In this way we may only use one instruction to process 128 records if the CPU has SSE2 capability, 256 records in case of AVX2 technology and 512 records for AVX-512.

The *Apriori* Principle allowed us to avoid searching patterns when some of their subpatterns are not frequent. An ordered storage of previously found patterns reduces the search time for a new candidate pattern. Multithreading capabilities of modern processors can also be used in an efficient way if the search for candidate patterns and checking the candidate patterns which have been found is split between available threads. For test datasets this approach gave us very promising results (about 70% of the theoretical acceleration, which is often hard to achieve).

Some algorithms previously designed to work with uncertainty in data were very slow compared to ones used for sequential pattern mining. The proposed code optimisation strategies showed that they can be very efficient and potentially applied to real time problems. This opens up a very large range of problems with errors in data which can now be solved numerically. Problems with time restrictions can also benefit from the suggested optimisation ideas as the run times are similar to ones without any restrictions.

In principle the ideas proposed in the paper are not limited to problems with uncertainty in data and time restrictions but can be useful for a wider range of problems from sequential to temporal pattern mining.

Conclusions

In this work we have presented a novel algorithm which can accommodate a range of time series problems, with or without time stamp uncertainties and with or without temporal constraints. We have demonstrated several levels of optimisation of the initial *Apriori* version to show methods which can be used to speed up most pattern mining calculations. We profiled and verified the algorithms with the existing fast SPAM code (for the case of sequential dataset with sequences length of one) and the RobustSPAM algorithm from [37] (with the assumption that uncertainty interval $\beta > 0$ and there are no coincident events). We showed that the algorithm outperforms RobustSPAM, and its final state of the art optimised versions outperform SPAM. The algorithms have been tested on two different datasets, an adult social care dataset and a weather temperature dataset.

Future works

The ideas presented in the paper offer a number of routes for further development. The algorithm, and the optimisation ideas are planned to be extrapolated for the case of sequential pattern mining (see Definition in [6]) on the datasets with temporal uncertainties with constraints on temporal length. In this case the structure of ID lists needs to be redefined, taking into account the information of which sequence in the row an element belongs to. The algorithm can also be extrapolated for the case of uncertain stream datasets. In this case the way of storage/accessing dataset should be revised. Once the frequent patterns are found with the algorithm FARPAMp, further analysis could be applied such as:

- Classification analysis (see for example [63, 64]);
- Clustering (see for example [65]);
- Building predictive models, etc. (see for example [66]).

In application to the Adult Social Care dataset the found frequent patterns will be cleaned from irrelevant ones and used to build a predictive model. The Adult Social Care dataset is divided into two groups: patients later assigned to an "expensive care" and everyone else. It is intended to used supervised learning machine tools for predictive analysis (for example, Random Forest or an Artificial Neural Network).

Abbreviations

AVX: advanced vector extensions; BFS: breadth first search; BIDE: BI-directional extension based frequent closed sequence mining; CloSpan: closed sequential pattern mining; CPU: central processing unit; CT: computed tomography; DFS: depth first search; EU: European Union; FreeSpan: frequent pattern-projected Sequential pattern mining; FARPAM: FAst Robust PAttern Mining; FARPAMp: FARPAM with prior information; ID: identifier; MPI: message passing interface; MRC: medical research council; MRI: magnetic resonance imaging; OpenMP: open multi-processing; PrefixSpan: prefix projected sequential pattern mining; pSPADE: parallel version of SPADE; RAM: random-access memory; SIMD: single instruction, multiple data; SPADE: Sequential PAttern Discovery using Equivalence classes; SPAM: Sequential PAttern Mining; SPMF: sequential pattern mining framework; SSE: streaming SIMD extensions.

Acknowledgements
The authors wish to thank Will Ridge for providing insights into Adult Social Care dataset and Anna Palczewska for cleaning and recoding it.

Authors' contributions
ST prepared the main text, ST and VT developed, coded and profiled the algorithms, VT preprocessed the weather dataset, JP and GA formulated the problem, suggested few initial ideas for algorithm improvement and influenced the shape of research. All authors suggested related works, discussed the structure of the paper and results. All authors read and approved the final manuscript.

Funding
This work has been supported by EPSRC Grant EP/N013980/1 (QuantiCode: Intelligent infrastructure for quantitative, coded longitudinal data).

Availability of data and materials
The weather datasets used for the current study and the codes are available from ST or VT on request.

Competing interests
The authors declare that they have no competing interests.

Author details
[1] School of Computing and Engineering, University of Huddersfield, Queensgate, Huddersfield HD1 3DH, UK. [2] School of Biological Sciences, University of Manchester, Oxford Road, Manchester M13 9PL, UK. [3] School of Mathematics, University of Leeds, Leeds LS2 9JT, UK.

Received: 17 February 2019 Accepted: 29 April 2019
Published online: 13 May 2019

References

1. Huh J-H. Big data analysis for personalized health activities: machine learning processing for automatic keyword extraction approach. Symmetry. 2018;. https://doi.org/10.3390/sym10040093.
2. Batal I, Cooper GF, Fradkin D, Harrison J, Moerchen F, Hauskrecht M. An efficient pattern mining approach for event detection in multivariate temporal data. Knowl Inform Syst. 2016;46(1):115–50. https://doi.org/10.1007/s10115-015-0819-6.
3. Lee S, Huh J-H. An effective security measures for nuclear power plant using big data analysis approach. J Supercomput. 2018;. https://doi.org/10.1007/s11227-018-2440-4.
4. Yu C, Boyd J. Fb $^+$-tree for big data management. Big Data Res. 2016;4(C):25–36. https://doi.org/10.1016/j.bdr.2015.11.003.
5. Agrawal R, Imieliński T, Swami A. Mining association rules between sets of items in large databases. SIGMOD Rec. 1993;22(2):207–16. https://doi.org/10.1145/170036.170072.
6. Agrawal R, Srikant R. Mining sequential patterns. In: Proceedings of the eleventh international conference on data engineering; 1995. p. 3–14. https://doi.org/10.1109/ICDE.1995.380415.
7. Zaki MJ. SPADE: an efficient algorithm for mining frequent sequences. Mach Learn. 2001;42(1/2):31–60. https://doi.org/10.1023/A:1007652502315.
8. Ayres J, Flannick J, Gehrke J, Yiu T. Sequential pattern mining using a bitmap representation. In: Proceedings of the eighth ACM SIGKDD international conference on knowledge discovery and data mining. KDD '02. New York: ACM. 2002. p. 429–35. https://doi.org/10.1145/775047.775109.
9. Han J, Pei J, Mortazavi-Asl B, Chen Q, Dayal U, Hsu M-C. Freespan: frequent pattern-projected sequential pattern mining. In: Proceedings of the sixth ACM SIGKDD international conference on knowledge discovery and data mining. KDD '00. ACM, New York, NY, USA. 2000. p. 355–9. https://doi.org/10.1145/347090.347167.
10. Pei J, Han J, Mortazavi-Asl B, Pinto H, Chen Q, Dayal U, Hsu M-C. Prefixspan: mining sequential patterns efficiently by prefix-projected pattern growth. In: Proceedings 17th international conference on data engineering. 2001. p. 215–24. https://doi.org/10.1109/ICDE.2001.914830.
11. Yan X, Han J, Afshar R. CloSpan: mining: closed sequential patterns in large datasets. p. 166–77. https://doi.org/10.1137/1.9781611972733.15.
12. Wang J, Han J. Bide: efficient mining of frequent closed sequences. In: Proceedings. 20th international conference on data engineering. 2004. p. 79–90. https://doi.org/10.1109/ICDE.2004.1319986.
13. Han J, Pei J, Yin Y, Mao R. Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Data Mining Knowl Dis. 2004;8(1):53–87. https://doi.org/10.1023/B:DAMI.0000005258.31418.83.
14. Zaki MJ. Parallel sequence mining on shared-memory machines. J Parallel Distrib Comput. 2001;61(3):401–26. https://doi.org/10.1006/jpdc.2000.1695.
15. Sutou T, Tamura K, Mori Y, Kitakami H. Design and implementation of parallel modified prefixspan method. In: Veidenbaum A, Joe K, Amano H, Aiso H, editors. High Perform Comput. Berlin: Springer; 2003. p. 412–22. https://doi.org/10.1007/978-3-540-39707-6_36.
16. Vu L, Alaghband G. A load balancing parallel method for frequent pattern mining on multi-core cluster. In: Proceedings of the symposium on high performance computing. HPC '15. Society for computer simulation international, San Diego, CA, USA. 2015. p. 49–58. http://dl.acm.org/citation.cfm?id=2872599.2872606.
17. Qiao S, Li T, Peng J, Qiu J. Parallel sequential pattern mining of massive trajectory data. Int J Comput Intell Syst. 2010;3(3):343–56. https://doi.org/10.1080/18756891.2010.9727705.

18. Tsai C-W, Lai C-F, Chao H-C, Vasilakos AV. Big data analytics: a survey. J Big Data. 2015;2(1):21. https://doi.org/10.1186/s40537-015-0030-3.
19. Kocheturov A, Momcilovic P, Bihorac A, Pardalos PM. Extended vertical lists for temporal pattern mining from multivariate time series. 2018. arXiv:1804.10025.
20. Lin J, Keogh E, Wei L, Lonardi S. Experiencing sax: a novel symbolic representation of time series. Data Mining Knowl Dis. 2007;15(2):107–44. https://doi.org/10.1007/s10618-007-0064-z.
21. Zhao J, Papapetrou P, Asker L, Boström H. Learning from heterogeneous temporal data in electronic health records. J Biomed Inform. 2017;65:105–19. https://doi.org/10.1016/j.jbi.2016.11.006.
22. Rathee S, Kashyap A. Adaptive-miner: an efficient distributed association rule mining algorithm on Spark. J Big Data. 2018;5(1):6. https://doi.org/10.1186/s40537-018-0112-0.
23. Lin C-W, Hong T-P. Temporal data mining with up-to-date pattern trees. Exp Syst Appl. 2011;38(12):15143–50. https://doi.org/10.1016/j.eswa.2011.05.090.
24. Guil F, Bailón A, Álvarez JA, Marín R. Mining generalized temporal patterns based on fuzzy counting. Exp Syst Appl. 2013;40(4):1296–304. https://doi.org/10.1016/j.eswa.2012.08.061.
25. Moskovitch R, Shahar Y. Classification of multivariate time series via temporal abstraction and time intervals mining. Knowl Inform Syst. 2015;45(1):35–74. https://doi.org/10.1007/s10115-014-0784-5.
26. Chen Y-C, Weng JT-Y, Hui L. A novel algorithm for mining closed temporal patterns from interval-based data. Knowl Inform Syst. 2016;46(1):151–83. https://doi.org/10.1007/s10115-014-0815-2.
27. Yao J, Kong S. The application of stream data time-series pattern reliance mining in stock market analysis. In: 2008 IEEE international conference on service operations and logistics, and informatics, vol. 1. 2008. p. 159–63. https://doi.org/10.1109/SOLI.2008.4686383.
28. Zhu C, Zhang X, Sun J, Huang B. Algorithm for mining sequential pattern in time series data. In: 2009 WRI international conference on communications and mobile computing, vol. 3. 2009. p. 258–62. https://doi.org/10.1109/CMC.2009.208.
29. Chen J, Chen P. Sequential pattern mining for uncertain data streams using sequential sketch. J Netw. 2014;9(2):252–8. https://doi.org/10.4304/jnw.9.2.252-258.
30. Reddy VS, Rao TV, Govardhan A. CASW: context aware sliding window for frequent itemset mining over data streams. Int J Comput Intell Res. 2017;13(2):183–96.
31. Lee VE, Jin R, Agrawal G. In: Aggarwal CC, Han J (eds). Frequent pattern mining in data streams. Cham: Springer; 2014. p. 199–224. https://doi.org/10.1007/978-3-319-07821-2_9.
32. Giannotti F, Nanni M, Pinelli F, Pedreschi D. Trajectory pattern mining. In: Proceedings of the 13th ACM SIGKDD international conference on knowledge discovery and data mining. KDD '07. New York: ACM; 2007. p. 330–9. https://doi.org/10.1145/1281192.1281230.
33. Papapetrou P, Kollios G, Sclaroff S, Gunopulos D. Mining frequent arrangements of temporal intervals. Knowl Inform Syst. 2009;21(2):133. https://doi.org/10.1007/s10115-009-0196-0.
34. Wu SY, Chen YL. Mining nonambiguous temporal patterns for interval-based events. IEEE Trans Knowl Data Eng. 2007;19(6):742–58. https://doi.org/10.1109/TKDE.2007.190613.
35. Kalaivany M, Uma V. Mining sequential patterns for interval based events by applying multiple constraints. Int J Comput Sci Appl. 2014;4(4):59–66. https://doi.org/10.5121/ijcsa.2014.4406.
36. Ruan G, Zhang H, Plale B. Parallel and quantitative sequential pattern mining for large-scale interval-based temporal data. In: 2014 IEEE international conference on Big Data (Big Data). 2014. p. 32–9. https://doi.org/10.1109/BigData.2014.7004410.
37. Palczewska A, Palczewski J, Aivaliotis G, Kowalik L. RobustSPAM for inference from noisy longitudinal data and preservation of privacy. In: 2017 16th IEEE international conference on machine learning and applications (ICMLA). 2017. p. 344–51. https://doi.org/10.1109/ICMLA.2017.0-137.
38. Palmes P, Pung HK, Gu T, Xue W, Chen S. Object relevance weight pattern mining for activity recognition and segmentation. Pervasive Mobile Comput. 2010;6(1):43–57. https://doi.org/10.1016/j.pmcj.2009.10.004.
39. Pei J, Han J. Constrained frequent pattern mining: a pattern-growth view. SIGKDD Explor Newslett. 2002;4(1):31–9. https://doi.org/10.1145/568574.568580.
40. Pei J, Han J, Wang W. Mining sequential patterns with constraints in large databases. In: Proceedings of the eleventh international conference on information and knowledge management. CIKM '02. New York: ACM. 2002. p. 18–25. https://doi.org/10.1145/584792.584799.
41. Bonchi F, Lucchese C. On closed constrained frequent pattern mining. In: ICDM '04. Fourth IEEE international conference on data mining. 2004. p. 35–42. https://doi.org/10.1109/ICDM.2004.10093.
42. Laxman S, Sastry PS. A survey of temporal data mining. Sadhana. 2006;31(2):173–98. https://doi.org/10.1007/BF02719780.
43. Han J, Cheng H, Xin D, Yan X. Frequent pattern mining: current status and future directions. Data Mining Knowl Dis. 2007;15(1):55–86. https://doi.org/10.1007/s10618-006-0059-1.
44. Lin M-Y, Lee S-Y. Efficient mining of sequential patterns with time constraints by delimited pattern growth. Knowl Inform Syst. 2005;7(4):499–514. https://doi.org/10.1007/s10115-004-0182-5.
45. Yang J, Yang C, Wei Y. Frequent pattern mining algorithm for uncertain data streams based on sliding window. In: 2016 8th international conference on intelligent human–machine systems and cybernetics (IHMSC), vol. 02. 2016. p. 265–8. https://doi.org/10.1109/IHMSC.2016.293.
46. Wang L, Cheung DWL, Cheng R, Lee SD, Yang XS. Efficient mining of frequent item sets on large uncertain databases. IEEE Trans Knowl Data Eng. 2012;24(12):2170–83. https://doi.org/10.1109/TKDE.2011.165.
47. Ge J, Xia Y, Wang J. Towards efficient sequential pattern mining in temporal uncertain databases. In: Cao T, Lim E-P, Zhou Z-H, Ho T-B, Cheung D, Motoda H, editors. Advancs knowledge discovery and data mining. Cham: Springer; 2015. p. 268–79. https://doi.org/10.1007/978-3-319-18032-8_21.
48. Cuzzocrea A, Leung CK-S, MacKinnon RK. Mining constrained frequent itemsets from distributed uncertain data. Fut Gen Comput Syst. 2014;37:117–26. https://doi.org/10.1016/j.future.2013.10.026.

49. Calders T, Garboni C, Goethals B. Efficient pattern mining of uncertain data with sampling. In: Zaki MJ, Yu JX, Ravindran B, Pudi V, editors. Advances in knowledge discovery and data mining. Berlin: Springer; 2010. p. 480–7.
50. Korf RE. Depth-first iterative-deepening: an optimal admissible tree search. Artif Intell. 1985;27(1):97–109. https://doi.org/10.1016/0004-3702(85)90084-0.
51. Han J, Kamber M, Pei J. 6—mining frequent patterns, associations, and correlations: basic concepts and methods. In: Han J, Kamber M, Pei J, editors. Data mining. The Morgan Kaufmann Series in data management systems. 3rd ed. Boston: Morgan Kaufmann; 2012. p. 243–78. https://doi.org/10.1016/B978-0-12-381479-1.00006-X.
52. Savasere A, Omiecinski E, Navathe SB. An efficient algorithm for mining association rules in large databases. In: Proceedings of the 21th international conference on very large data bases. VLDB '95. San Francisco: Morgan Kaufmann Publishers Inc.; 1995. p. 432–44. http://dl.acm.org/citation.cfm?id=645921.673300.
53. Orlando S, Lucchese C, Palmerini P, Perego R, Silvestri F. kDCI: a multi-strategy algorithm for mining frequent sets. In: FIMI. 2003.
54. Toivonen H. Sampling large databases for association rules. In: Proceedings of the 22th international conference on Very large data bases. VLDB '96. San Francisco: Morgan Kaufmann Publishers Inc.; 1996. p. 134–45. http://dl.acm.org/citation.cfm?id=645922.673325.
55. Bayardo RJ Jr. Efficiently mining long patterns from databases. SIGMOD Rec. 1998;27(2):85–93. https://doi.org/10.1145/276305.276313.
56. Burdick D, Calimlim M, Gehrke J. Mafia: a maximal frequent itemset algorithm for transactional databases. In: Proceedings of the 17th international conference on data engineering. IEEE computer society, Washington, DC; 2001. p. 443–52. http://dl.acm.org/citation.cfm?id=645484.656386.
57. Pasquier N, Bastide Y, Taouil R, Lakhal L. Discovering frequent closed itemsets for association rules. In: Proceedings of the 7th international conference on database theory. ICDT '99. London: Springer; 1999. p. 398–416. http://dl.acm.org/citation.cfm?id=645503.656256.
58. Zaki MJ. Scalable algorithms for association mining. IEEE Trans Knowl Data Eng. 2000;12(3):372–90. https://doi.org/10.1109/69.846291.
59. Zaki M.J, Gouda K. Fast vertical mining using diffsets. In: Proceedings of the ninth ACM SIGKDD international conference on knowledge discovery and data mining. KDD '03; 2003. p. 326–35. https://doi.org/10.1145/956750.956788.
60. Fu T-C. A review on time series data mining. Eng Appl Artif Intell. 2011;24(1):164–81. https://doi.org/10.1016/j.engappai.2010.09.007.
61. Knuth DE. The art of computer programming, vol. 3. 2nd ed. Redwood City: Addison Wesley Longman Publishing Co., Inc.; 1998.
62. Fournier-Viger P, Lin JC-W, Gomariz A, Gueniche T, Soltani A, Deng Z, Lam HT. The SPMF open-source data mining library version 2. In: Berendt B, Bringmann B, Fromont É, Garriga G, Miettinen P, Tatti N, Tresp V, editors. Machine learning and knowledge discovery in databases. Cham: Springer; 2016. p. 36–40.
63. Pijls W, Potharst R. Classification based upon frequent patterns. In: Kowalczyk R, Loke SW, Reed NE, Williams GJ, editors. Advances in artificial intelligence. PRICAI 2000 workshop reader. Berlin: Springer; 2001. p. 72–9.
64. Wang P, Wu X-C, Wang C, Wang W, Shi B-L. CAPE—a classification algorithm using frequent patterns over data streams. J Comput Res Develop. 2004;41:1677–83.
65. Zimek A, Assent I, Vreeken J. In: Aggarwal CC, Han J, editors. Frequent pattern mining algorithms for data clustering. Cham: Springer. 2014. p. 403–23. https://doi.org/10.1007/978-3-319-07821-2_16.
66. Rezig S, Achour Z, Rezg N. Using data mining methods for predicting sequential maintenance activities. Appl Sci. 2018;8:2184. https://doi.org/10.3390/app8112184.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.