Information Sciences 582 (2022) 865–896

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Mining weighted sequential patterns in incremental uncertain databases

Kashob Kumar Roy [a], Md Hasibul Haque Moon [a], Md Mahmudur Rahman [a], Chowdhury Farhan Ahmed [a,*], Carson Kai-Sang Leung [b]

[a] Department of Computer Science and Engineering, University of Dhaka, Bangladesh
[b] Department of Computer Science, University of Manitoba, Canada

ARTICLE INFO

Article history:
Received 17 June 2020
Received in revised form 27 September 2021
Accepted 2 October 2021
Available online 7 October 2021

Keywords:
Data mining
Sequential pattern mining
Weighted sequential patterns
Uncertain database
Incremental database

ABSTRACT

Due to the rapid development of science and technology, the importance of imprecise, noisy, and uncertain data is increasing at an exponential rate. Thus, mining patterns in uncertain databases have drawn the attention of researchers. Moreover, frequent sequences of items from these databases need to be discovered for meaningful knowledge with great impact. In many real cases, weights of items and patterns are introduced to find interesting sequences as a measure of importance. Hence, a constraint of weight needs to be handled while mining sequential patterns. Besides, due to the dynamic nature of databases, mining important information has become more challenging. Instead of mining patterns from scratch after each increment, incremental mining algorithms utilize previously mined information to update the result immediately. Several algorithms exist to mine frequent patterns and weighted sequences from incremental databases. However, these algorithms are confined to mine the precise ones. Therefore, in this work, we develop an algorithm to mine frequent sequences in an uncertain database in this work. Furthermore, we propose two new techniques for mining when the database is incremental. Extensive experiments have been conducted for performance evaluation. The analysis showed the efficiency of our proposed framework.

1. Introduction

The way toward mining concealed information from the massive extent of data is known as data mining. This information can be frequent patterns, irregular patterns, correlated patterns, association rules among events, etc. Researchers have concentrated mostly on mining frequent patterns, which can be a set of items or a sequence of itemsets or any substructure that frequently occurs in a given database and a minimum support threshold. A plethora of mining algorithms have shown their superior performance in different applications of data.

Apriori [2] is the first algorithm based on candidate generation and testing paradigm in the field of data mining which finds frequent itemsets and association rules among them. A well-known anti-monotone (downward closure) property was introduced in [2] to reduce the search space for mining frequent itemsets. Due to the huge memory and time requirements of *Apriori* [2] like algorithms, pattern growth-based approaches [20] have drawn great attention for this task.

* Corresponding author at: Department of Computer Science and Engineering, University of Dhaka, Dhaka 1000, Bangladesh.
E-mail addresses: mahmudur@cse.du.ac.bd, mahmudur@du.ac.bd (M.M. Rahman), farhan@du.ac.bd (C.F. Ahmed), kleung@cs.umanitoba.ca (C.K.-S. Leung).

https://doi.org/10.1016/j.ins.2021.10.010
0020-0255/

Consequently, different extensions of frequent itemset mining problems were introduced i.e., handling weight constraint [21,49] to mine interesting patterns, and utility constraint [17,18,31,32] to mine high-utility patterns.

In many applications, the order or sequence of the items in databases is essential and must be maintained while mining. Thus, the problem of sequential pattern mining was introduced first in [41]. *GSP* [41] is a generalized solution based on a candidate generation and testing approach. It generates a lot of false-positive sequences and requires multiple scans of the database to obtain true-positive ones. Afterward, *PrefixSpan* [38] was proposed to overcome this limitation in the mining of sequential patterns. It is based on a pattern-growth approach and follows a depth-first search strategy. Approaches like *PrefixSpan* employ different efficient pruning techniques to reduce the search space while growing larger patterns. Rizvee et al. [40] proposed a tree-based approach to mine sequential patterns and demonstrated their efficient performance.

However, uncertainty is inherent in real-world data due to the noise and inaccuracy of many data sources. With the increasing use of uncertain data in modern technologies such as sensor data, environment surveillance, medical diagnosis, security, and manufacturing systems, uncertain databases are growing larger. Uncertainty of an item in a database makes both the itemset mining and sequential pattern mining difficult. Several algorithms developed in [1,23,26,27] are able to mine frequent itemsets in uncertain databases. Uncertain itemset mining with weight constraints is addressed in [3,24,29]. Following *PrefixSpan* [38], there are several algorithms proposed to mine sequential patterns in uncertain databases [35,39,50].

Moreover, many real-life applications reflect that all the frequent sequences are not equally important. Hence, different weights are assigned to different items of a database corresponding to the significance of an item. As a result, each sequence gets a weight, and thus interesting sequential patterns can be mined against different weight and support thresholds. Sequential pattern mining with weight constraints has been explored in [49]. To maintain the anti-monotone property, the upper bound of weight is commonly set as the maximal weight of all items in the database [21,49]. *uWSequence* [39] can handle weight constraints in mining uncertain sequence databases. It follows the expected support based mining process. To incorporate anti-monotone property in the mining process, it has proposed an upper-bound measure of expected support. Besides, it uses a weighting filter separately to handle the weight constraint in mining sequences. Thus, it can efficiently mine only those sequences that have high frequencies with high weights. Generally, in weighted pattern mining, sequences that have low frequencies with high weights or high frequencies with low weights are also important in several real-life applications. Again, it is necessary to design sophisticated pruning upper bounds to mine weighted patterns efficiently while maintaining the anti-monotone property as well as limiting false positive pattern generation.

Therefore, in this work, we propose multiple novel pruning upper bounds that are theoretically tightened instead of respective upper bounds already introduced in the literature. Besides, efficient maintenance of candidate patterns is required to develop a faster method for computing expected supports of patterns. Hence, we utilize a hierarchical index structure to maintain candidate patterns in a space-efficient way that leads to a way faster support computation method than state-of-art methods.

Furthermore, most real-life databases are dynamic in nature. In recent years, a large amount of research has been conducted in incremental mining, i.e., [4,17,22,32,36,46] to frequent itemsets with/without different constraints such as weight, utility etc., [6,21,30] to find the updated set of sequential patterns, or weighted sequential patterns or high utility sequential patterns from dynamic databases. Note that all above incremental algorithms are confined into precise databases. Nonetheless, all of these sequential pattern mining algorithms for uncertain databases are not efficient to handle the dynamic nature of data. Because, running a *batch algorithm* like *PrefixSpan* or *uWSequence* from scratch for each increment requires a huge amount of time and memory. Thus the lack of efficient methods on incremental mining of weighted sequential patterns in uncertain databases and its importance has stimulated us to explore this field.

1.1. Motivation

The use of uncertain data in the modern world is increasing day by day. In most cases, the database is not static. New increments are added to the database gradually; hence the set of frequent sequences may change. After each increment, running existing algorithms from scratch, which can mine frequent sequences in static uncertain databases, is very expensive in terms of time and memory. Therefore it has become inevitable to design an efficient technique that can maintain and update frequent sequences when the database grows incrementally. A few scenarios described below reflect the importance of finding frequent weighted sequences from the incremental uncertain database.

1.1.1. Example one

Frequent pattern mining is being widely applied in medical data. It is beneficial to discover hidden knowledge or extract important patterns from massive data on patient symptoms. Health workers/organizations can use these discovered patterns to give patients proper treatment at the right time or observe disease behavior during an outbreak. For instance, the severe acute respiratory syndrome coronavirus 2 (SARS-CoV-2) causes the coronavirus disease 2019 (COVID-19). Researchers from the whole world are working on the COVID-19 outbreak as the world is expecting to face a huge economic recession with losing a lot of people. If we see the nature of this disease's symptoms, then it is clear that the symptoms have the nature of sequential occurrences. Again, there is uncertainty in the patient data inherently. Because the same physiological index corresponds to different symptom association probabilities for patients due to their different physical conditions.

Let us consider a symptom sequence of a COVID-19 patient, e.g., $\{(fever : 0.4), (cough : 0.5), (\text{sorethroat} : 0.7, breathing\ problem : 0.9)\}$ where each real value denotes the association probability for the corresponding symptom. Furthermore, all symptoms may not have the same significance to diagnose a disease. Some symptoms might be severe, but others might not be, e.g., shortness of breath, chest pain, etc., are severe symptoms for COVID-19; in contrast, dry cough, tiredness, etc., are mild symptoms. Consequently, the weights of symptoms should play a crucial role in mining important symptom sequences from patient data. So, considering the nature of the symptoms, it can be said that finding weighted frequent sequences of the symptoms of the corona patients can be helpful to predict whether someone is infected or even the current stage of infection. Besides, it will be possible to take proper treatment for an infected patient by predicting the next stage of infection based on discovered frequent patterns of symptoms. For example, at a particular stage of infection, a patient may have mild pain. From the database of symptoms, it has been seen that a patient having mild pain would have severe breathing problems afterward. So it will be helpful to take proper precautions for the patient.

Moreover, the spreading of the virus is so rapid that the amount of patient data is growing every moment; its nature is changing over time. Even the nature of COVID-19 is also varying from place to place. Thus, due to rapidly growing patient data, non-incremental mining algorithms are not efficient enough to find frequent sequences within a short time; it requires a massive amount of time to run the mining algorithm on the whole database from scratch every time we get new data. Consequently, an efficient algorithm to find weighted frequent sequences from incremental databases is very much needed. Therefore, incrementally mined patterns from massive patient data flow are essential to determine how the symptoms change with time or vary from one region/country to another.

1.1.2. Example two

Mining social network behavioral patterns can be another example of uncertain data. These patterns can be discovered from users' activities in social networks. If we observe a user's activities over a while, we can estimate how similar a user is with a student, a photographer, and so on. In other words, we can assign a probability associated to each category, e.g., $\{(student : 0.9), (photographer : 0.7, cyclist : 0.3), (tourist : 0.7)\}$ where it indicates that the probability of being a student for a particular user is estimated after analyzing his activities over a period of time in social networks, as a photographer or a cyclist after analyzing his activities over the next period, and so on. This period could be a couple of minutes or hours or even days or weeks. Again, users are using their social networks at every moment while new users are joining social networks. As a result, a huge amount of data is being produced at every moment, which means that the database of users' activities is growing rapidly. Therefore, an efficient incremental algorithm is required to mine user behavioral patterns on the fly. Nowadays, these discovered patterns are beneficial in a wide range of applications. For instance, in this era of social network-based digital marketing, the study of consumer activities is most important for marketers to understand consumers' behavioral patterns for their customized advertisements. If we discover the patterns on the fly, it is possible to customize advertisements on a user's newsfeed based on his latest activities over social networks. Besides, different patterns discovered from social network data are also crucial in influencer marketing, trend analysis, social event detection, social spam analysis, etc.

Furthermore, anomaly or fraud analysis over users' activities requires observing the behavioral patterns over a period of time. The behavioral patterns of a fraud user show a deviation from that of regular users over a while. Thus, to detect fraud users from its social network activities, it needs to mine the behavioral patterns incrementally for a certain period of time. Therefore, an efficient incremental mining algorithm can play an essential role in discovering anomalies in the patterns on the fly because it is essential to detect fraud users or other anomalies, e.g., rumors, within the shortest possible time.

1.1.3. Example three

The traffic management system is being automated to identify the behavior of vehicles and drivers. Several methods like Automatic Number Plate Recognition, Speed Recognition, Vehicle Type Recognition, and Trajectory Analysis produce uncertain data. We can mine different patterns like "Going to road A first and then road B is a frequent behavior of 10% of the cars". These patterns are generally sequential. In an intelligent transportation system (ITS), different roads/junctions may carry different significance. Again, all data collected through sensor/GPS are associated with some certainty values. Consequently, it is to be said that weighted sequential patterns can play an important role in traffic automation, such as planning and monitoring traffic routes. However, different patterns can be seen in different parts of a day, i.e., patterns at the office/school opening time are quite the opposite of office/school closing time in a day. Again, weekday patterns are different from weekend patterns. Patterns even vary between different periods in a year. Note that patterns may change over different periods of a day, a week, a month, or even a year. Therefore, these changes in daily patterns, weekly patterns, or yearly patterns can be mined from incrementally growing traffic databases. Patterns mined from different database increments are helpful to draw the pattern trends across different periods of a day or even a year. These pattern trends are required for analyzing the seasonal behavior of traffic in ITS.

A few other applications are Tumor Necrosis Factor Receptor (TNFR) disease analysis, DNA sequencing (micro-level information with uncertainty), mining in crime data, weather data, fashion trend [39], and vehicle recognition data [35]; Wireless Sensor Network (WSN) data monitoring [50]; social network behavior analysis [3], etc., where a benefit of mining weighted frequent sequential patterns is to discover more meaningful hidden knowledge. As the existing algorithms are not efficient to mine weighted sequential patterns in incremental uncertain databases, finding efficient techniques has become an inevitable research issue.

1.2. Contributions

In this work, we propose a new framework to deal with weight constraints in mining sequential patterns in uncertain databases to address the issues mentioned above. In this framework, we introduce a new concept of *weighted expected support* of a sequential pattern. An efficient algorithm *FUWS* is developed based upon this framework to find weighted sequential patterns from any static uncertain databases. Furthermore, we propose two techniques to find the updated set of weighted sequential patterns when the uncertain database is of dynamic nature. To the best of our knowledge, this is the first work to mine weighted sequential patterns from incremental uncertain databases. Our key contributions of this work are as follows:

1. An efficient algorithm, *FUWS*, to mine weighted sequential patterns in uncertain databases.
2. Two new techniques, *uWSInc* and *uWSInc+*, for mining weighted sequential patterns in incremental database of uncertain sequences.
3. A new hierarchical index structure, *USeq-Trie*, for maintaining weighted uncertain sequences.
4. Two upper bound measures, $expSup^{cap}$ and wgt^{cap}, are for expected support and weight of a sequence, respectively.
5. A pruning measure, $wExpSup^{cap}$, to reduce the search space of mining patterns.
6. An extensive experimental study to validate the efficiency and effectiveness of our approach and its supremacy with respect to the existing methods.

Although there has been a good amount of work in the broad domain of pattern mining, it is clear that mining weighted sequential patterns in uncertain databases is not explored well despite its rising importance. This article presents solutions to this research issue and provides extensive experimental results to validate our claims.

The rest of the article is organized in the following sections: background study and discussion of related works in Section 2, our proposed solutions with a proper simulation in Section 3, analysis of experimental results in Section 4 and finally, conclusions in Section 5.

2. Literature review

As with the development of modern technologies, the usages of data are being intensified day by day. A lot of varieties have emerged with a variety of information stored and the knowledge required for different scenarios. Hence, various algorithms have already been developed for mining heterogeneous information. A review of the literature is described below.

2.1. Sequential Pattern Mining

A *sequence database* is a list of data tuples where each tuple is an ordered set of itemsets/events. Unlike itemset mining, it contains order information for events. Thus, sequential patterns mined from a sequence database can significantly impact many applications [16]. In recent years, a plethora of research has been conducted on sequential pattern mining [16,19,43]. The problem of *Sequential Pattern Mining (SPM)* was first discussed in [41]. In which, Srikant and Agrawal proposed a generalized solution named *GSP* [41], which is inspired by the itemset mining algorithm *Apriori* [2]. It has the problem of infrequent pattern generation, and it needs a massive amount of running time. It also needs extensive memory to store all the k-sequences (i.e., sequences of length k to use in the generation of $(k+1)$-sequences.

Afterward, following *FP-Growth* [20] for itemset mining, pattern-growth based approach for sequential pattern mining, *PrefixSpan* was introduced in [38]. It uses the divide-and-conquer technique to mine frequent sequences from a precise database. *PrefixSpan* starts mining from frequent sequences of length 1. Then it projects databases into smaller parts by taking the frequent sequences as a prefix. Afterward, it expands longer patterns and further projects into smaller databases recursively. Later, many improvements have been found in different specific applications by designing efficient pruning techniques in *PrefixSpan* to reduce the search space. A recent work [40] has proposed a compact and efficient tree-structure, *SP-Tree*, to store the whole database. They have utilized the idea of co-existing item table to facilitate the mining of sequential patterns and proposed an algorithm named *Tree-Miner* which holds the *build once mine many* property. Still, *Tree-Miner* algorithm needs a huge amount of memory space to store the whole database which makes it incompetent for incremental mining.

2.2. Weighted Sequential Mining

All frequent items, as well as all sequences, are not equally important. Such examples have been discussed in Section 1.1. Different weights are assigned to different items to reflect the significance of various patterns. The weight of an itemset or sequence can be calculated using the item weights. To mine interesting patterns, the concept of weighted support is introduced in [48,49]. They defined the *weighted support* of a sequence as the resultant value of multiplying its support count with weight value. However, this weighted support violates the anti-monotone property. To apply this property, an upper-bound value of weighted support is used for a sequence. Based on this upper bound value, sequences are extended from frequent 1-sequences using *PrefixSpan* like approach. *WIP* [48] finds weighted frequent itemsets where *WSPAN* [49] is popular for min-

ing weighted frequent sequences. An extra scan of the database is required to find the weighted support of generated patterns and remove the false ones.

2.3. Mining in Uncertain Databases

Handling uncertain databases has become a major concern as their use is increasing in almost every application field. Several pattern-growth based solutions have been proposed, such as *UFP-Growth* [25] and *PUF-Growth* [26]. Other algorithms for mining patterns in large uncertain datasets [45], mining of weighted frequent uncertain itemsets [3,29], mining high-utility uncertain itemsets with both positive and negative utility [18] are proved to be efficient. Yan et al. [47] explored uncertain data in 2D space.

However, in the case of sequential pattern mining and different constraints in it, it still needs researchers' attention. Muzammal et al. [34] formulated uncertainty in uncertain sequential pattern mining as source-level uncertainty, i.e., each tuple contains a probability value and element-level uncertainty, i.e., each event in tuples contains a probability value. Two measures of frequentness such as *expected support* and *probabilistic frequentness* are commonly used in frequent itemset and sequential patterns in uncertain databases. Muzammal et al. [35] explored source-level uncertainty in probabilistic databases and proposed a dynamic programming algorithm to calculate the expected support and also breadth-first and depth-first methods based on candidate generation-and-test paradigm to mine patterns. *U-PrefixSpan* [50] follows the *PrefixSpan* algorithm [38] to mine classic sequential patterns in uncertain databases under the probabilistic frequentness measure.

In contrast, *uWSequence* [39] mines sequential patterns based on expected support. It uses a weighting filter separately to mine interesting patterns. To the best of our knowledge, *uWSequence* [39] is the only work to mine weighted sequential patterns from uncertain databases. It proposed an upper-bound measure of expected support of a sequence called $expSupport^{top}$, which is used in the core mining process to project the database and grow patterns. A research concern is how to determine this upper bound measure to reduce the search space of the pattern-growth approach.

Another line of research focuses on high utility-based sequential pattern mining in uncertain databases. Projection-based *PMiner* algorithm [33] takes into account both average utility and uncertainty factors to efficiently mine high average-utility sequential patterns in the uncertain databases. [31] proposed a projection-based *PHAUP* algorithm with three novel pruning strategies under an innovative high average-utility sequential pattern mining framework that is superior to mine high average-utility sequential patterns than *PMiner* [33]. *UHUOPM* [9] introduced the concept of Potential High Utility Occupancy Patterns (PHUOPs) to incorporate three factors: support, probability, and utility occupancy while maintaining the downward closure property in the mining process. Ahmed et al. [8] proposed an evolutionary model called *MOEA-HEUPM* to find the non-dominated high expected-utility patterns from uncertain databases without prior knowledge by utilizing a multiobjective evolutionary algorithm based on decomposition (MOEA/D). Gautam et al. [11] presented an efficient algorithm *HEUSPM* to discover high expected utility sequential patterns in IoCV environments.

2.4. Incremental Mining Algorithms

For incremental frequent itemset mining, *FUP* [7] is one of the early and well-known contributions which needs to rescan the database when an item is frequent in the incremented portion but was absent in the result set before this increment. *CanTree* [28] proposed a tree structure where nodes are ordered canonically instead of frequency. It captures the whole transaction database and does not require any rescan of the whole database or any reconstruction of the tree for any increment. *CP-tree* [42] periodically restructures the incremental tree structure according to the frequency descending order of items. Thus, it achieves not only a highly compressed tree structure but also a remarkable gain in mining time compared to the corresponding *CanTree*. Two efficient tree structures [4] have been proposed to mine weighted frequent patterns from an incremental database with only one scan. Incremental weighted erasable pattern mining from incremental databases has also been explored in [36] which proposed efficient list structures. Lin et al. [12] proposed *RWFI-Mine* and its further improvement named as *RWFI-EMine* to discover recent weighted frequent itemsets efficiently in a temporal database while considering both weight and the recency constraints of patterns. Gan et al. [10] introduced the concept of Recent High Expected Weighted Itemset (RHEWI) to take the weight, uncertainty and recency constraints of patterns into account. Consequently, it proposed two projection-based algorithms *RHEWI-P* and *RHEWI-PS* to mine RHEWIs from uncertain temporal databases. *ILUNA* [13] proposed a single-pass incremental mining of uncertain frequent patterns without false positive patterns.

Recently, a number of incremental methods have been developed to mine high utility patterns in [17,32,46]. Wang et al. [45] proposed two incremental algorithms to mine *Probabilistic Frequent Itemsets (PFI)* from large uncertain databases: (a) *uFUP*, which is a *candidate generation and test* based algorithm inspired by *FUP*[7] that mines exact *PFI* and (b) $uFUP_{app}$, which shows more efficiency in time and memory than *uFUP*, but mines approximate *PFI*. Techniques based on maintaining the whole database in a compact tree enabled efficient mining of the updated set of patterns after each increment.

We naturally need a massive memory space to store the whole database in a tree structure for sequential pattern mining because the number of nodes required to store all data sequences is exponentially high compared to an itemset database. For example, for an itemset $\{a, b, c\}$, there are a large number of possible sequential instances because of different event formation and their order of appearance, such as $< (a)(b)(c) >, < (b)(a)(c) >, < (a)(b, c) >, < (a, b, c) >, < (a, b), (b), (a, c) >$, and so

on. All of the algorithms for sequential pattern mining discussed in the previous sections have a common limitation: they are designed to be performed once on the static database. If any update in the database occurs, users have to run these algorithms from scratch each time. This is very inefficient in terms of time and memory, especially if the increments are frequent and small. In incremental mining, the challenge is to find the updated set of frequent sequences in the shortest possible time. Another major concern is that the algorithm cannot be expensive in memory usage. *IncSpan* [6] algorithm proposed a solution for incremental mining of sequential algorithms with the concept of *Semi-Frequent Sequences* (*SFS*). Later, *IncSpan+* [37] identified its limitation and proposed a corrected version which claimed to give complete results but includes the problem of full database scanning. However, *PBIncSpan* [5] proved that *IncSpan+* also fails to find complete results. Using a prefix-based tree structure for pattern maintenance, [5] shows that finding complete results is very challenging when the number of nodes becomes huge, which is obvious in incremental sequence databases.

PreFUSP-TREE-INS is proposed in [30] to reduce the number of rescans and it incorporates the *pre-large concept*. Note that, *pre-large sequences* are the same as *semi-frequent sequences*. Their main limitation is that it performs better only when the increment size is small, such as 0.05%, 0.1%, and 0.2% of the original database. Besides, there is a safety bound of *pre-large concept*, i.e., the patterns stored in the *pre-large* set can help up to a specific limit of database update. Results in [30] suggest that this safety bound is very small as it showed efficient performance if the total increment ratio is below 0.2%. These results motivated us to design an efficient incremental solution for multiple large increments, i.e., to support both larger increment size and larger total increment ratio.

Moreover, Wang and Huang [44] proposed incremental algorithms named *IncUSP-Miner, IncUSP-Miner+* to mine high-utility sequential patterns. They proposed a compact tree structure named *Candidate-pattern tree* to maintain the patterns, a tighter upper bound of utility-based measures to prune the tree nodes better, and few strategies to reduce the tree-node updates and database rescans. The major limitation is that they perform better only when the ratio between the incremented size and original database size is small.

Nevertheless, to the best of our knowledge, *WIncSpan* [21] is the only incremental solution for weighted sequential pattern mining. Similar to *IncSpan* and *IncSpan+*, it also uses extra buffers to store *semi-frequent sequences (SFS)* from initial database. It also has the same limitation of *IncSpan+* that any new pattern arriving later or any pattern that was initially not in the semi-frequent set cannot be found even if it becomes frequent after future increments. However, it does not need to rescan the database after any increment. Results showed that a reasonable amount of buffering *SFS* becomes enough to find the almost completed set of the updated result within a very short time.

As shown in Table 1, the current literature suggests re-running *uWSequence* [39] from scratch every time after an increment occurs to uncertain sequential databases. It is very costly in terms of time and space when the database grows larger and larger. Therefore, we have explored this problem and propose two efficient techniques for incremental mining, *uWSInc* and *uWSInc+*, under a novel framework for mining both unweighted and weighted uncertain sequential patterns, where multiple theoretical tightened pruning upper bound measures and an efficient hierarchical index structure to maintain patterns, *USeq-Trie*, have been developed. Consequently, this framework leads to an efficient design of an algorithm *FUWS* for static uncertain databases. In the following discussions, we will be consistent with weighted uncertain pattern mining. Nonetheless, it is to be noted that we can easily adapt our weighted framework for mining unweighted patterns by setting the weights of all items as 1.0.

3. A framework for mining weighted uncertain frequent sequences

Before diving into the detailed description of our proposed framework, we will discuss some preliminaries which will be referred throughout this article.

3.1. Preliminaries

Sequences* and *Uncertain Sequence Database. Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of all items in a database. An itemset or event $e_i = (i_1, i_2, \ldots, i_k)$ is a subset of I. A sequence $S = <e_1, e_2, \ldots, e_m>$ is an ordered set of itemsets [41]. For example, $S_1 = <(i_2), (i_1, i_5), (i_1)>$ consists of 3 consecutive itemsets. In case of uncertain sequences, items in each itemset are assigned with their existential probabilities such as $S_1 = <(i_2: p_{i_2}), (i_1: p_{i_1}, i_5: p_{i_5}), (i_1: p_{i_2})>$ [35,39,50]. An *uncertain sequence database (DB)* is a collection of uncertain sequences. An example database containing six uncertain sequences is shown in Table 2.

***Support* and *Expected Support*.** *Support* of a sequence in a database is the number of data tuples that contains *S* as a subsequence. For example, $< (a)(b) >$ has a support of 5 in Table 2. In uncertain databases, *expected support* count makes more sense than this general support count. Different authors defined *expected support* in several ways for an itemset or a sequence in literature. Here, we adopted the definition of *expected support* for a sequence from *uWSequence* [39] where all items in a sequence are considered independent of each other. The *expected support* is defined as the sum of the maximum probabilities of that sequence in each data tuple. The probability of a sequential pattern is defined simply by multiplying the probability values of that pattern's items in a data tuple. In Table 2, for the sequence, $< (a)(b) >$, where *b* occurs in a different event after the occurrence of *a*, the expected support,

Table 1
A brief summary of survey algorithms.

	General	Weighted	High Utility	Uncertain
Sequential Pattern Mining	[38,40,41]	[39,49]	[9,31,14,15,43]	[31,33,34,35,39,50]
Incremental Itemset Mining	[7,28,42]	[4,22,36]	[32, 46]	[13,45]
Incremental Sequential Mining	[6,30]	[21]	[44]	

Table 2
Uncertain Sequential Database, *DB*.

Id	Uncertain Sequence
1	(a:0.9, c:0.6)(a:0.7)(b:0.3)(d:0.7)
2	(a:0.6, c:0.4)(a:0.5)(a:0.4, b:0.3)
3	(a:0.3)(a:0.2, b:0.2)(a:0.4, b:0.3, g:0.5)
4	(a:0.1, c:0.1)(a:0.3, b:0.1, c:0.4)
5	(d:0.1)(a:0.4)(d:0.1)(a:0.5, c:0.6)
6	(b:0.3)(b:0.4)(a:0.1)(a:0.1, b:0.2)

$$expSup(< (a)(b) >) = \omega 1 + \omega 2 + \omega 3 + 0.1 \times 0.1 + 0 + 0.1 \times 0.2 = 0.57$$
$$where,$$
$$\omega 1 = max(0.9 \times 0.3, 0.7 \times 0.3)$$
$$\omega 2 = max(0.6 \times 0.3, 0.5 \times 0.3)$$
$$\omega 3 = max(0.3 \times 0.2, 0.3 \times 0.3, 0.2 \times 0.3).$$

To explain the maximum probability of the pattern $< (a)(b) >$ in 1st sequence, it is the maximum of 0.9×0.3 (a in 1st event, b in 3rd event) and 0.7×0.3 (a in 2nd event, b in 3rd event).

Sequence *Size* and *Length*. *Size* of a sequence α is the number of total itemsets/events in it and is represented by $|\alpha|$. *Length* of a sequence is the total count of items present in all events of the sequence. For example, size of $< (a,c)(a,b,c) >$ is 2 but its length is 5. Sequence of length m is also called a m-sequence.

Extension of a Sequence. A sequence α can be extended with an item i in two ways, i.e., i-extension and s-extension. If item i is added to the last event of α then the resultant sequence, say β, is called i-extension of α. For example, $< (a)(b,c) >$ is an i-extension of $< (a)(b) >$ with item c. In i-extension, size of a sequence does not increase but its length increases. Similarly, if item i is appended to α as a new event then the new sequence β is called α's s-extension with i. For example, $< (a)(b)(c) >$ is a s-extension of $< (a)(b) >$ with item c.

***Weight* of a sequence.** Similar to *expected support*, there are several definitions of a *sequence weight*. We adopted the definition of the *weight* of a sequence denoted as *sWeight* in [49,21] where *sWeight* is the sum of its each individual item's *weight* divided by the length of the sequence. Weights of the items are belong to the range between 0 to 1 as shown in Table 3. For example, $sWeight(< (a)(ac) >) = (0.8 + 0.8 + 0.9)/3 = 0.833$.

***Frequent* and *Semi-frequent Sequences*.** The set of sequences that suffice a given minimum support threshold (or expected support threshold for uncertain database) are called *frequent sequences*. Similarly, the sequences that meet a given minimum threshold of *weighted expected support*, are called *weighted frequent uncertain sequences* or *weighted uncertain sequential patterns*. A *buffer ratio*, μ, which is of positive value less than 1.0, is chosen to lower the minimum support threshold to find *semi-frequent sequences* that are not frequent but their values are very closed to the minimum support threshold. As discussed earlier, *semi-frequent sequences* are helpful to find the updated set of result sequences when the database is incremental.

In the following subsections, we propose a new framework for mining weighted frequent sequential patterns in uncertain databases where a new concept of weighted expected support is introduced to incorporate the weight constraint in the mining process. Based on this framework, our proposed algorithms will be discussed, followed by an example simulation. Before

Table 3
Weight of different items.

Item	Weight
a	0.8
b	1.0
c	0.9
d	0.9
e	0.7
f	0.9
g	0.8

diving into the detailed description, additional required definitions, proposed measures, and lemmas with proof are presented in the following discussion.

Definition 1. *maxPr* is the maximum possible probability of the considering sequential pattern in the projected database. For a pattern $\alpha = < (i_1) \dots (i_m) >$, *maxPr* used in *uWSequence* [39] can be defined as

$$maxPr(\alpha) = \prod_{k=1}^{|\alpha|} \widehat{P}_{DB|\alpha_{k-1}}(i_k) \text{ where } \alpha_{k-1} = (i_1) \dots (i_{k-1}) \tag{1}$$

Here, $\widehat{P}_{DB|\alpha}(i)$ = maximum possible probability of item i in the database $DB|\alpha$, which is projected with α as current prefix. Rahman et al. [39] shows that the *maxPr* measure holds the anti-monotone property.

Example 1. The *maxPr* value for $< (c)(a) >$ = 0.6 × 0.7 = 0.42 in Table 2 where maximum possible probability of item c in DB (as shown in Table 2) is $\widehat{P}_{DB}(< (c) >) = 0.6$ and again,

$\widehat{P}_{DB|<(c)>}(< (a) >) = 0.7$ because 0.7 is maximum possible probability of item a in the $< (c) >$-projected database, $DB| < (c) >$.

As an another example, we can find that $maxPr(< (a)(c) >)$ = 0.54.

Definition 2. The $maxPr_S(\alpha)$ is defined as the maximum probability of a sequential pattern α in a single data sequence S. It can be formulated as:

$$maxPr_S(\alpha) = \max_{\mathcal{E} \in \mathcal{Q}} \left(\prod_{k \in \mathcal{E}} p_k \right) \tag{2}$$

where $\mathcal{Q}$ denotes a set of all the sets of the sequential positions for each occurrence of the pattern α in S. In addition, $\mathcal{E}$ is a single set of the sequential positions for a particular occurrence of the pattern α in S. Moreover, p_k is the existential probability of the corresponding item at k-th position in S.

Example 2. Suppose, we need to find the $masPr_S(< (a)(b) >)$ in the 1st sequence $S = S^{<1>}$ in Table 2. Here, $\mathcal{Q} = \{\{1,4\},\{3,4\}\}$.

Therefore, $maxPr_S(< (a)(b) >) = max(0.9 \times 0.3, 0.7 \times 0.3) = 0.27$.

Definition 3. The *expected support* for a sequential pattern α can be calculated from a database DB using the equation as follows,

$$expSup(\alpha) = \sum_{i=1}^{|DB|} maxPr_{S^{<i>}}(\alpha) \tag{3}$$

Where, $S^{<i>}$ denotes the i-th Sequence in the database DB and $|DB|$ denotes the size of the database DB.

Example 3. To calculate the $expSup(< (ac) >)$ in Table 2, from the definition of $maxPr_S(\alpha)$, we get

$maxPr_{S^{<1>}}(< (ac) >) = 0.54, maxPr_{S^{<2>}}(< (ac) >) = 0.24, maxPr_{S^{<3>}}(< (ac) >) = 0,$

$maxPr_{S^{<4>}}(< (ac) >) = 0.12, maxPr_{S^{<5>}}(< (ac) >) = 0.30$ and $maxPr_{S^{<6>}}(< (ac) >) = 0$.

Thus, $expSup(< (ac) >) = 1.20$.

Definition 4. $expSup^{cap}(\alpha)$ is an upper bound of expected support of a pattern α which is defined as

$$expSup^{cap}(\alpha) = maxPr(\alpha_{m-1}) \times \sum_{\forall S \in (DB|\alpha_{m-1})} maxPr_S(i_m) \tag{4}$$

Example 4. To compute the $expSup^{cap}(< (ac)(b) >)$ in Table 2, as per the definition,

$$expSup^{cap}(< (ac)(b) >) = maxPr(< (ac) >) \times \sum\nolimits_{\forall S \in DB|(< (ac) >)} maxPr_S(< (b) >)$$

where, $maxPr(< (ac) >) = 0.54$ and $\sum_{\forall S \in DB|(< (ac) >)} maxPr_S(< (b) >) = 0.7$: because $DB|(< (ac) >)$ has three non-empty sequences i.e., $< (a:0.7)(b:0.3)(d:0.7) >, < (a:0.5)(a:0.4, b:0.3) >$, and $< (a:0.3, b:0.1, c:0.4) >$. The values of $maxPr_S(< (b) >)$ for these three sequences are 0.3, 0.3, and 0.1.

Consequently, $expSup^{cap}(< (ac)(b) >) = 0.54 \times 0.7 = 0.378$.

Similarly, the $expSup^{cap}$ value of another sequence $< (ac) >, expSup^{cap}(< (ac) >) = 1.8$.

Lemma 1. *The $expSup^{cap}$ of a sequential pattern is always greater than or equal to the actual expected support of that pattern.*

Proof. To keep the proof less complicated, we consider a sequential pattern $\alpha = < (i_0)(i_1)\ldots(i_m) >$ where each event/itemset consists of a single item, i_k.

According to the definitions, $\forall i_k \in \alpha : maxPr(i_k) \geqslant maxPr_S(i_k)$.

$\Rightarrow maxPr(i_0) \times \sum_{\forall S \in (DB|i_0)} maxPr_S(i_1) \geqslant \sum_{\forall S \in DB} maxPr_S(< i_0)(i_1) >)$

$\Rightarrow maxPr(\alpha_{m-1}) \times \sum_{\forall S \in (DB|\alpha_{m-1})} maxPr_S(i_m) \geqslant \sum_{\forall S \in DB} maxPr_S(\alpha)$

$\Rightarrow expSup^{cap}(\alpha) \geqslant expSup(\alpha)$

$\therefore$ The equality holds only when each item has same existential probability for its all positions in whole database. Otherwise, $expSup^{cap}(\alpha) > expSup(\alpha)$ will always be true. □

Lemma 2. *For any sequence α, the value of $expSup^{cap}(\alpha)$ is always less than or equal to the $expSupport^{top}(\alpha)$ which is used as an upper bound of expected support in uWSequence [39] and can be equivalently defined as $expSupport^{top}(\alpha) = maxPr(\alpha_{m-1}) \times maxPr(i_m) \times sup_{i_m}$ where sup_{i_m} denotes the support count of i_m.*

Proof. According to definitions, $\forall S, \forall i_k \in \alpha : maxPr_S(i_k) \leqslant maxPr(i_k)$

$\Rightarrow \sum maxPr_S(i_m) \leqslant maxPr(i_m) \times sup_{i_m}$

$\Rightarrow maxPr(\alpha_{m-1}) \times \sum maxPr_S(i_m) \leqslant maxPr(\alpha_{m-1}) \times maxPr(i_m) \times sup_{i_m}$

$\Rightarrow expSup^{cap}(\alpha) \leqslant expSupport^{top}(\alpha)$ □

Thus, being a tighter upper bound of expected support, $expSup^{cap}$ can reduce the search space more in pattern-growth based mining process and hence it generates less false positive patterns than $expSupport^{top}(\alpha)$.

Definition 5. $WES(\alpha)$ is the *weighted expected support* of a sequential pattern α defined as

$$WES(\alpha) = expSup(\alpha) \times sWeight(\alpha) \tag{5}$$

It is inspired by the widely used concept of *weighted support* for precise databases as described in Section 2.

Example 5. According to Table 2 and Table 3, weighted expected support of $< (a)(ac) >$,
$WES(< (a)(ac) >) = (0.1 \times 0.3 \times 0.4 + 0.4 \times 0.5 \times 0.6) \times (0.8 + 0.8 + 0.9)/3 = 0.11$.

Definition 6. Weighted Frequent Sequential Pattern: a sequence α is called weighted frequent sequential pattern if $WES(\alpha)$ meets a minimum weighted expected support threshold named as *minWES*. This minimum threshold is defined to be,

$$minWES = min\_sup \times database\ size \times WAM \times wgt\_fct \tag{6}$$

Here, *min_sup* is user given value in range [0,1] related to a sequence's expected support, *WAM* is weighted arithmetic mean of all item-weights present in the database defined as

$$WAM = \frac{\sum_{i \in I} f_i \times w_i}{\sum_{i \in I} f_i} \tag{7}$$

where w_i and f_i are the weight and frequency of item i in current updated database. The value of *WAM* changes after each increment in the database. *wgt_fct* is user given positive value chosen for tuning the mining of weighted sequential patterns. Choice of *min_sup* and *wgt_fct* depends on aspects of application.

Example 6. Let us assume that $min\_sup = 0.2$ and $wgt\_fct = 0.75$.

From Table 2 and Table 3, $WAM = \frac{(14\times0.8)+(8\times1.0)+(5\times0.9)+(3\times0.9)+(1\times0.8)}{14+8+5+3+1} = 0.88$; database size = 6; Therefore, $minWES = 0.792$.

However, the measure *WES* does not hold anti-monotone property as any item with higher weight can be appended to a weighted-infrequent sequence, and the resulting super sequence may become weighted-frequent. So, to employ anti-monotone property in mining weighted frequent patterns, we propose two other upper bound measures, wgt^{cap} and $wExpSup^{cap}$, which are used as upper bound of weight and weighted expected support, respectively.

Definition 7. We define an upper bound of weight for a pattern α of length m, $wgt^{cap}(\alpha)$ as follows,

$$wgt^{cap}(\alpha) = \max(mxW_{DB}(DB|\alpha_{m-1}), mxW_s(\alpha)) \tag{8}$$

where $mxW_{DB}(DB|\alpha_{m-1})$ is the weight of the item with maximum weight value in the projected database and $mxW_s(\alpha)$ is the weight of the item with maximum weight value in the sequential pattern α.

Example 7. Suppose, we need to calculate the $wgt^{cap}(\alpha)$, where, $\alpha =< (ac) >$.

From Table 3, $mxW_s(< (ac) >) = \max(0.8, 0.9) = 0.9$.

Again, items of $(DB| < (a) >)$ are a, b, c, d, and g in Table 2.

So, $mxW_{DB}(DB|(< (a) >) = \max(0.8, 1.0, 0.9, 0.9, 0.8) = 1.0$.

Therefore, $wgt^{cap}(< (ac) >) = \max(1.0, 0.9) = 1.0$.

To handle the downward property of weighted frequent patterns in precise databases, Ishita et al. [21] and Yun [49] attempted to use the maximal weight of all items in the whole database as the upper bound of the weight of a sequence, noticing that this upper bound may generate much more false-positive patterns. To narrow the search space and keep the number of false candidate patterns as small as possible, we use wgt^{cap} as an upper bound in mining weighted patterns. Intuitively, using wgt^{cap} instead of the maximal weight of all items is more beneficial because there may be many patterns for which the value of wgt^{cap} is less than the maximal weight of all items. Hence, the patterns may meet the minimum threshold because of the higher value of maximal weight but may not satisfy the threshold due to the lower value of wgt^{cap}, resulting in a narrower search space of mining patterns and fewer candidate patterns.

Lemma 3. *For any sequential pattern α of length m, the value of $wgt^{cap}(\alpha)$ is always greater than or equal to the sWeight value of its all super patterns.*

Proof. Let us assume that $\alpha \subset \alpha'$ for some sequential pattern α' of length m' where $m' > m$.

According to Definition 7, $mxW_S(\alpha) \geqslant sWeight(\alpha)$. The equality holds when the weights of all item in α are equal. Similarly, $mxW_S(\alpha') \geqslant sWeight(\alpha')$.

Now, if the weights of all items in database are not equal, then $mxW_{DB}(DB|\alpha_{m-1}) \geqslant mxW_{DB}(DB|\alpha'_{m'-1})$ must hold since $DB|\alpha_{m-1}$ contains all frequent items of $DB|\alpha'_{m'-1}$.

Moreover, it is straightforward that $mxW_S(\alpha')$ is always greater than or equal to $mxW_S(\alpha)$. Nevertheless, when $mxW_S(\alpha') > mxW_S(\alpha)$, the item with maximum weight in α' must come from the projected database, $DB|\alpha_{m-1}$ and thus $mxW_S(\alpha') \leqslant mxW_{DB}(DB|\alpha_{m-1})$.

$\therefore \max(mxW_S(\alpha'), mxW_{DB}(DB|\alpha'_{m'-1})) \leqslant \max(mxW_S(\alpha), mxW_{DB}(DB|\alpha_{m-1}))$

$\Rightarrow wgt^{cap}(\alpha') \leqslant wgt^{cap}(\alpha)$.

$\Rightarrow sWeight(\alpha') \leqslant wgt^{cap}(\alpha)$

Again, if the weights of all items are equal, then $wgt^{cap}(\alpha') = sWeight(\alpha') = sWeight(\alpha) = wgt^{cap}(\alpha)$.

Therefore, we can conclude that the value of $wgt^{cap}(\alpha)$ is always greater or equal to the value of $sWeight(\alpha)$ or $sWeight(\alpha')$ which is true for all cases.

Definition 8. As mentioned before, the upper bound of weighted expected support is $wExpSup^{cap}(\alpha)$, defined as:

$$wExpSup^{cap}(\alpha) = expSup^{cap}(\alpha) \times wgt^{cap}(\alpha) \tag{9}$$

Example 8. Considering the pattern $\alpha =< (ac) >$, the value of $wExpSup^{cap}(< (ac) >) = 1.8 \times 1.0 = 1.8$ where $expSup^{cap}(\alpha) = 1.8$ (as computed in Example 4) and $wgt^{cap}(< (ac) >) = 1.0$ (see in Example 7).

Lemma 4. *The value of $wExpSup^{cap}(\alpha)$ is always greater than or equal to weighted expected support of a sequential pattern $\alpha, WES(\alpha)$. Hence, using the $wExpSup^{cap}$ value of any pattern as the upper bound of weighted expected support in mining patterns, it may generate some false positive frequent patterns.*

Proof. Lemma 1 and 3 has showed that for a sequential pattern α, $expSup^{cap}(\alpha) \geqslant expSup(\alpha)$ and $wgt^{cap}(\alpha) \geqslant sWeight(\alpha)$

$\Rightarrow expSup^{cap}(\alpha) \times wgt^{cap}(\alpha) \geqslant expSup(\alpha) \times sWeight(\alpha)$

$\Rightarrow wExpSup^{cap}(\alpha) \geqslant WES(\alpha)$

$\therefore$ The value of $wExpSup^{cap}(\alpha)$ is always greater than or equal to the value of $WES(\alpha)$ for any sequence α.

As a result, some patterns might be introduced as frequent patterns due to its higher value of $wExpSup^{cap}$ being not actually weighted frequent. □

Lemma 5. *If the value of $wExpSup^{cap}$ for a sequential pattern, α, is below the minimum weighted expected support threshold minWES, the value of WES for that pattern and its all super patterns must not satisfy the threshold. In other words, the pattern α and its all super patterns must not be frequent if the value of $wExpSup^{cap}(\alpha)$ does not satisfy the threshold. Thus, the anti-monotone property holds while mining patterns.*

Proof. Assume that $\alpha \subseteq \alpha'$ for some patterns α'. Recall that, $\text{expSup}(\alpha) = \sum_{\forall S \in DB} maxPr_S(\alpha)$ [39]. By definition, $expSup(\alpha) \geqslant expSup(\alpha')$.

Again, $expSup^{cap}(\alpha) \geqslant expSup(\alpha) \Rightarrow expSup^{cap}(\alpha) \geqslant expSup(\alpha')$.

Moreover, $wgt^{cap}(\alpha) \geqslant sWeight(\alpha')$.

Now, $wExpSup^{cap}(\alpha) = expSup^{cap}(\alpha) \times wgt^{cap}(\alpha) \geqslant expSup(\alpha') \times sWeight(\alpha') = WES(\alpha')$.

So, if $wExpSup^{cap}(\alpha) < minWES$ holds, then $WES(\alpha') < minWES$ must hold for any $\alpha' \supseteq \alpha$.

Therefore, upper bound $wExpSup^{cap}$ could be able to find out the complete set of frequent patterns. □

Pruning Condition. In pattern-growth based mining algorithms, we define a pruning condition to reduce the search space. According to Lemma 5, we can safely define our pruning condition which is to be used in mining algorithm as follows:

$$wExpSup^{cap}(\alpha) \geqslant minWES \tag{10}$$

The $wExpSup^{cap}(\alpha)$ determines whether the current pattern α will be used to generate its super patterns or not. To prune weighted infrequent patterns earlier during the mining process but maintain the anti-monotone property, the $wExpSup^{cap}$ is calculated instead of *WES* to mine potential candidate patterns. If, for any k-sequence α, $wExpSup^{cap}(\alpha) < minWES$, then any possible extension of α to a $(k+1)$-sequence can be safely pruned, i.e, mining of longer patterns with prefix α can be ignored without missing any actual frequent patterns. However, to get actual frequent patterns, we must prune out false positive patterns from the set of candidate patterns because $wExpSup^{cap}$ is an approximate value.

3.2. USeq-Trie: maintenance of patterns

In our proposed algorithms, we have used a hierarchical data structure, *USeq-Trie*, to store patterns compactly and to update their weighted expected support efficiently. Each node in *USeq-Trie* will be created as either an s-extension or i-extension from its parent node. Each edge is labeled by an item. In an s-extension, the edge label is added as a different event. In an i-extension, it is added in the same event as its parent. The sequence of the edge labels in a path to a node from the root node denotes a pattern.

For example, $< (a) >, < (ab) >, < (b) >, < (c) >, < (c)(d) >$, and $< (d) >$ are frequent sequential patterns which are stored into *USeq-Trie* shown in Fig. 1. In this figure, the s-extensions are denoted by solid lines and i-extensions by dashed lines. By traversing the *USeq-Trie* in depth-first order, we will get all patterns stored in it. Each node represents the pattern up to that node from the root node and stores its weighted expected support which is ignored in Fig. 1 for simplicity.

Insertion and Deletion. To insert patterns into the *USeq-Trie*, we start to traverse it from the root node and take the first item in a pattern as the current item. If there is a child node with an edge labeled by the current item, we go to that node. Otherwise, we create a child node with an edge labeled by the current item. Then we move to the child node and set the next item in the pattern as the current item. Recursively, we follow the same process up to the last item in the pattern. For example, now we insert $< (ab)(c) >$ into the *USeq-Trie* shown in Fig. 1. The root node has a child node with an edge labeled by *a*. So, we go to the child node labeled by (a) and check whether it has an edge labeled by *b*. The extension has to be an i-extension as *a* and *b* are in the same event. As we can see, such a node already exists. So, we go to the node which is labeled by *(ab)*. Afterward, we have to check if there is an s-extension by an edge labeled *c*. But there is no such child node. Therefore, we create a node by s-extension and set *c* as the edge label and *(ab)(c)* as the node label. Similarly, we insert $< (b)(c) >$ and

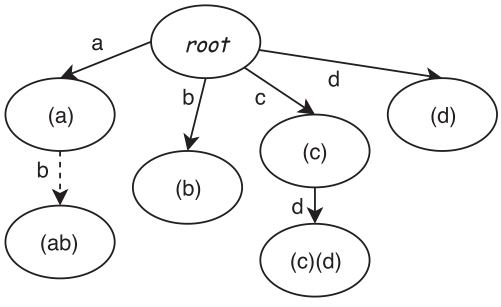


Fig. 1. Storing frequent sequences in a *USeq-Trie*.

$< (cd) >$ into the *USeq-Trie*. The updated *USeq-Trie* is shown in Fig. 2. To delete a pattern from *USeq-Trie*, we traverse the corresponding path which represents the pattern in bottom-up order. The node in the path having no child nodes in the *USeq-Trie* will be removed while traversing the corresponding path from the leaf node to the root node. The resulting *USeq-Trie* after deleting $< (ab)(c) >$ is shown in Fig. 3.

Support Calculation. We propose an efficient method denoted as *SupCalc* by using *USeq-Trie* to calculate weighted expected support of patterns. In this method, it reads sequences from the dataset one by one and updates the support of all candidate patterns stored into the *USeq-Trie* against this one. For a sequence $S =< e_1 e_2 .. e_n >$ (where e_i is an event or itemset), the steps are defined as follows:

- At each node, define an array the *size of S*, which is *n* in this case. At the root node, all values of the array will always be 1.0.
- Traverse the *Useq-Trie* in depth-first order. After following an edge, let the current pattern from root to a particular node be α which ends with the item i_k. The maximum *weighted expected support* of the pattern α is stored at proper indices of the following node's array. These proper indices are the ending positions of α as a sub-sequence in *S*. Set values to zero at other indices.
- While traversing the *USeq-Trie*, iterate all events in *S*. (i) For *s-extension* with an item i_k, we calculate the support of the current pattern α (ends with i_k in a new event) by multiplying the probability of item i_k in current event, e_m with the maximum probability in the parent node's array up to the event e_{m-1}. The resulting support is stored at position m in the following node's array. (ii) For i-extension, the support will be calculated by multiplying the probability of the item i_k in e_m with the value at position m in the parent node's array and stored at position m in the following child node's array. After that, the maximum value in the resulting array multiplied with its weight will be added to the weighted expected support of the current patterns at the corresponding node.
- Use the resultant array to calculate the support of all super patterns while traversing the next nodes.

Algorithm 1: Procedure of SupCalc

Input: *DB:* initial database, *candidateTrie*: stores candidate patterns
Output Calculated weighted expected supports for all patterns

Procedure SupCalc(*DB*, *candidateTrie*)
  for all $S =< e1, e2, e3, \ldots, en >\in DB$ **do** ▷ e_k is an itemset/event
    $ar \leftarrow$ the array of size equals to the number of events in $|S|$, which is initialized as 1
    $wgt\_sum, itm\_cnt \leftarrow 0$ and 0 ▷ *wgt_sum* – sum of items' weights and *itm_cnt* – their counts in a pattern
    TrieTraverse(*S*, *null*, *candidateTrie.root*, *ar*, *wgt_sum*, *itm_cnt*)
Procedure TrieTraverse(*S*, *cur_itmset*, *cur_node*, *ar*, *wgt_sum*, *itm_cnt*)
  for all *node* ∈ *cur_node.descendents* **do**
    cur_edge ← edge label between current child *node* and *cur_node*
    cur_ar ← the array of size *ar* initialized as 0
    for all $e_k \in S$
      if S-Extension Is TRUE **then**
        cur_itmset ← *cur_edge*
        if*cur_itmset* $\in e_k$ **then**
          mxSup ← $\max_{i=1}^{k-1} ar_i$
          $cur\_ar_k \leftarrow mxSup \times p_{cur\_edge}$ ▷ p_{cur_edge} denotes the existential probability of *cur_edge* in e_k
        if *cur_itmset* $\notin e_k$
          $cur\_ar_k \leftarrow 0$
      if I-Extension Is TRUE **then**
        cur_itmset ← (*cur_itmset* ∪ *cur_edge*) ▷ *cur_itmset* is extended with *cur_edge*
        if *cur_itmset* $\in e_k$ **then**
          $cur\_ar_k \leftarrow ar_k \times p_{cur\_edge}$ ▷ p_{cur_edge} denotes the existential probability of *cur_edge* in e_k
        if *cur_itmset* $\notin e_k$ **then**
          $cur\_ar_k \leftarrow 0$
    $mxSup \leftarrow \max_{i=1}^{|S|} cur\_ar_i$
    $cur\_wgt\_sum \leftarrow wgt\_sum + wgt_{cur\_edge}$ ▷ wgt_{cur_edge} denotes the weight of *cur_edge*
    $cur\_itm\_cnt \leftarrow itm\_cnt + 1$
    $node.WES \leftarrow node.WES + mxSup \times \frac{cur\_wgt\_sum}{cur\_itm\_cnt}$
    TrieTraverse(*S*, *cur_itmset*, *node*, *cur_ar*, *cur_wgt_sum*, *cur_itm_cnt*)

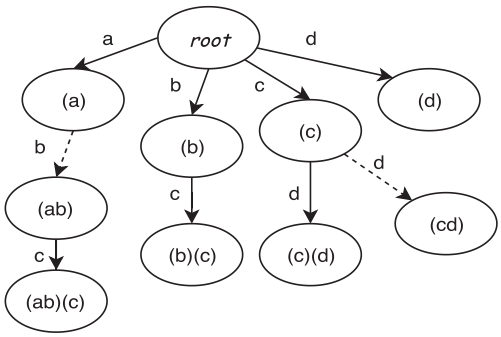


Fig. 2. After insertion of patterns.

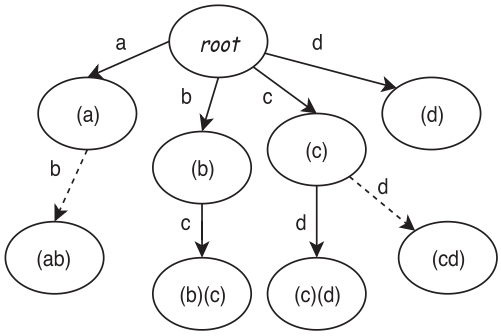


Fig. 3. After deleting $< (ab)(c) >$ pattern.

For example, we consider a data sequence, $S =< (a:0.8)(b:0.6)(a:0.9, b:0.7)(c:0.3)(d:0.9) >$. To calculate weighted expected support, the size of array is equal to 5 where each value is set initially as 1. For the edge label *a*, the values at indices 1 and 3 of the array will be 0.8 and 0.9, respectively, and at other indices will be 0. The maximum value in the array which is 0.9 multiplied with its weight 0.8 will be added to *WES* of the current pattern at the child node which is labeled by *(a)*. *(ab)* is an i-extension of *(a)* with item *b*. Item *b* is only in the e_3 itemset in *S*. The value of *(ab)* at position 3 in the array of node labeled by *(ab)* will be the probability of item *b* in e_3 itemset in *S* multiplied with the value at position 3 in the array of node labeled by *(a)* which is $0.7 \times 0.9 = 0.63$. The values at other positions in the array of node *(ab)* will be 0.0. The maximum value in the resulting array (0.63) multiplied with its *sWeight* $((0.8 + 1.0)/2 = 0.9)$ will be added to the *WES* of node *(ab)*. Afterward, by traversing the *USeq-Trie* in depth-first order, let us consider the next branch from the root node of the *USeq-Trie*. The array for the edge label *b* contains values of 0.6 and 0.7 at indices 2 and 3, respectively. The *WES* of the node labeled by (b) equals to the value of max(0.6,0.7) multiplied with the weight of *b*. Next, $(b)(c)$ is a s-extension of (b) with item *c*. Item *c* is only in the e_4 event in *S*. The value of $(b)(c)$ at position 4 in the array of node labeled by $(b)(c)$ will be the probability of item *c* in e_4 event in *S* multiplied with the maximum value up to the index 3 in the array of node labeled by (b), which is $0.3 \times 0.7 = 0.21$. The values at other positions in the array of node $(b)(c)$ will be 0.0. The maximum value in the resulting array (0.21) multiplied with its *sWeight* $((1.0 + 0.9)/2 = 0.95)$ will be added to the *WES* of node $(b)(c)$. Similarly, the support of all other patterns at corresponding nodes will be calculated. The results are shown in Fig. 4.

The pseudo-code has been given in Algorithm 1. It takes $O(N \times |S|)$ to update *N* number of nodes against the sequence *S*. Therefore, the total time complexity of actual support calculation is $O(|DB| \times N \times k)$ where *k* is the maximum sequence length in the dataset. It outperforms the procedure used in *uWSequence* [39] which needs $O\left(|DB| \times N \times k^2\right)$. Moreover, we can remove false-positive patterns and find frequent ones with *O(N)* complexity. Thus, the use of *USeq-Trie* leads our solution to become more efficient.

3.3. FUWS: faster mining of uncertain weighted frequent sequences

In order to reduce the number of false-positive patterns by introducing a sophisticated upper bound measure, $wExpSup^{cap}$, and to make mining patterns more efficient, we develop an algorithm named as *FUWS* inspired by *PrefixSpan* [38], to mine weighted sequential patterns in an uncertain database. The sketch of *FUWS* algorithm is as follows,

- Process the database such that the existential probability of an item in a sequence is replaced with the maximum probability of its all next occurrences in this sequence. This idea is similar to the preprocess function of *uWSequence* [39]. In

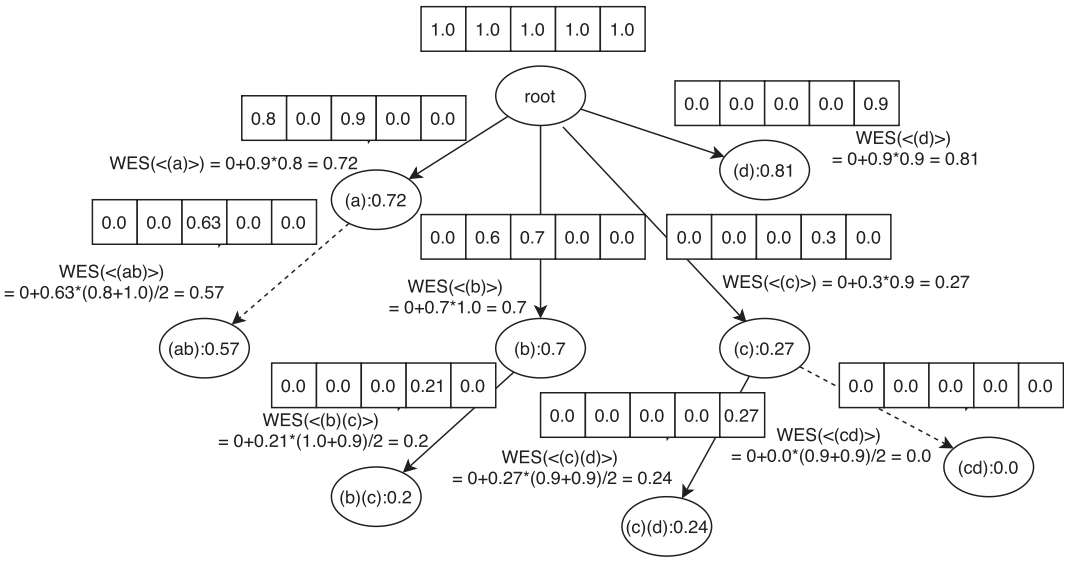


Fig. 4. Pattern Maintenance and WES calculation using *USeq-Trie*.

addition, sort the items in an event/itemset in lexicographical order. This preprocessed database will be used to run the *PrefixSpan* like mining approach to find the candidates for frequent sequences.

- Calculate *WAM* of all items present in the current database and set the threshold of weighted expected support, *minWES*.
- Find 1-length frequent items and, for each item, project the preprocessed database into smaller parts and expand longer patterns recursively. Store the candidates into a *USeq-Trie*.
- While growing longer patterns, extend current prefix α to α' with an item β as s-extension or i-extension according to the pruning condition defined in Eq. 10.
- Use of $wExpSup^{cap}$ value instead of actual support generates few false-positive candidates. Scan the whole actual database, update weighted expected supports and prune false-positive candidates based on their weighted expected support.

Algorithm 2: Procedure of *FUWS*

Input: *DB:* initial database, *min_sup:* support threshold, *wgtFct:* weight factor
Output: *FS:* set of weighted frequent patterns

1 **procedure** FUWS(*DB*, *min_sup*, *wgtFct*)
2 $pDB, WAM \leftarrow preProcess(\text{DB})$
3 $minWES \leftarrow min\_sup \times |pDB| \times WAM \times wgt\_fct$
4 $extItms, maxPrs, iWgts \leftarrow Determine(pDB)$ to find the set of potential s-extendable items.
5 **for all** $\beta \in extItms$ **do**
6 $expSup^{cap}(\beta) \leftarrow \times \sum_{\forall S \in (pDB)} maxPr_S(\beta)$
7 $wgt^{cap}(\beta) \leftarrow \max(iWgts)$ ▷ *iWgts* – List of all items' weights in *pDB*
8 **if** $wExpSup^{cap}(\beta) = expSup^{cap}(\beta) \times wgt^{cap}(\beta) \geqslant minWES$ **then**
9 $candidateTrie \leftarrow FUWSP\Big(pDB|\beta, \beta, maxPrs_\beta, iWgts_\beta, iWgts_\beta, 1\Big)$
10 **Call** SupCalc(*DB*, *candidateTrie*)
11 FS ← Remove false positives and find frequent patterns from *candidateTrie*
12 **procedure** FUWSP(*DB*, α, $maxPr_\alpha$, mxW_α, $sWgt_\alpha$, $|\alpha|$)
13 $extItms$, $mxPrs$, $iWgts \leftarrow Determine(DB)$ to find set of potential i or s-extendable items for prefix α.
14 **for all** $\beta \in extItms$ **do**
15 $expSup^{cap}(\alpha \cup \beta) \leftarrow maxPr_\alpha \times \sum_{\forall S \in (DB|\alpha)} maxPr_S(\beta)$
16 $wgt^{cap}(\alpha \cup \beta) \leftarrow \max(mxW_\alpha, iWgts)$ ▷ *iWgts* – List of all items' weights in *DB*
17 **if** $wExpSup^{cap}(\alpha \cup \beta) = expSup^{cap}(\alpha \cup \beta) \times wgt^{cap}(\alpha \cup \beta) \geqslant minWES$ **then**
18 $maxPr_{\alpha\cup\beta} \leftarrow maxPr_\alpha \times mxPrs_\beta$
19 $sWgt_{\alpha\cup\beta} \leftarrow sWgt_\alpha + sWgts_\beta$
20 $FUWSP\Big(DB|\beta, (\alpha \cup \beta), maxPr_{\alpha\cup\beta}, \max\Big(mxW_\alpha, sWgts_\beta\Big), sWgt_{\alpha\cup\beta}, |\alpha \cup \beta|\Big)$

Pseudo-code for *FUWS* is shown in Algorithm 2. The function *preProcess* in Line 2 prepares the input database for pattern-growth approach *FUWSP* in *FUWS*. It computes *WAM* to incorporate our weight constraint. In Algorithm 2, the function, *Determine*, finds the set of all the potential extendable items, the maximum probabilities of items in the processed database, and their weights from the weight vector. In our mining process *FUWSP*, we have used $wExpSup^{cap}$, the upper bound for weighted expected support of a sequence, to find out the set of i-extendable and s-extendable items and their probabilities and weights in Line 13. In Lines 14–20, when the value of $wExpSup^{cap}(\alpha \cup \beta)$ satisfies the threshold *minWES*, *FUWSP* is called recursively for the pattern $(\alpha \cup \beta)$ to generate its super sequences. Otherwise, the algorithm prunes the current pattern $(\alpha \cup \beta)$ and its super patterns. As a result, *FUWSP* generates all potential candidate sequences recursively. In addition to that, we use *candidateTrie* which is a *USeq-Trie* to store candidates and update their weighted expected support efficiently. However, at Lines 10–11, *FUWS* calls the *SupCalc* function in Algorithm 1 to calculate weighted expected support for all candidate sequences stored into the *candidateTrie* and remove false ones. An extensive simulation of *FUWS* and its experimental results are discussed in Sections 3.5 and 4 respectively.

3.4. Two approaches for incremental database

As we have mentioned, mining the complete set of frequent weighted sequential patterns is very expensive with respect to time and space when the database grows dynamically, so to find out the almost complete set of weighted frequent sequences (FS), we propose two techniques, *uWSInc* and *uWSInc+*. In both approaches, we lower the minimum threshold *minWES* to $minWES' = minWES \times \mu$ where $0 < \mu < 1$ is an user-chosen buffer ratio. Sequences with weighted expected support less than *minWES* but at least equal to $minWES'$ are stored as weighted semi-frequent sequences, which are named as *SFS*.

3.4.1. uWSInc: faster incremental mining of uncertain weighted frequent sequences

Instead of running an algorithm from scratch after each increment, *uWSInc* algorithm works only on the appended part of the database. At first, it runs *FUWS* once to find *FS* and *SFS* from the initial dataset and uses *USeq-Trie* to store frequent and semi-frequent sequences. After each increment ΔDB, the algorithm follows the steps as listed below:

1. Update database size and *WAM* value. Calculate the new threshold of weighted expected support.
2. For each sequence α in *FS* and *SFS*, update its weighted expected support, WES_α, by using our proposed faster support calculation method, *SupCalc*, described in Algorithm 1.
3. Update the *FS* and *SFS* by comparing updated WES_α values with the new *minWES* and $minWES'$ respectively. A sequence may go to one of the updated FS' or SFS', or vice versa or it may become infrequent. Once a pattern becomes infrequent, it will be removed and its information will get lost.
4. Use FS' and SFS' as *FS* and *SFS* for the next increment.

Algorithm 3: Procedure of *uWSInc*

Input: *DB:* initial database, ΔDB : new increments, *min_sup:* support threshold, μ: buffer ratio, *wgt_fct:* weight factor
Output: *FS:* set of weighted frequent patterns

```
1  procedure InitialMining(DB, ΔDB_i, min_sup, μ, wgt_fct)
2    seqTrie ← FUWS(DB, min_sup × μ, wgt_fct)
3    for all ΔDB_i do
4      DBSize ← DBSize + ΔDB_iSize
5      seqTrie ← uWSInc(ΔDB_i, min_sup, μ, wgt_fct, seqTrie)
6  procedure uWSInc(ΔDB_i, min_sup, μ, wgt_fct, seqTrie)
7    Call SupCalc(ΔDB_i, seqTrie)
8    minWES ← min_sup × DBsize × WAM × wgt_fct
9    for all β ∈ (FS ∨ SFS) stored into seqTrie
10     if wExpSup(β) < (minWES × μ) then
11       Remove pattern β from seqTrie
12   FS ← seqTrie.find_frequent_patterns(minWES)
```

Pseudocode for *uWSInc* is given in Algorithm 3. The *uWSInc* algorithm runs *FUWS* once on the initial database, *DB*. In Line 2, it stores both *FS* and *SFS* into a single *seqTrie* which brings out more space compactness. For each increment ΔDB_i, it calls *SupCalc* function in Algorithm 1 to update the weighted expected support (*WES*) values of all *FS* and *SFS* accordingly. In Lines 9–11, it traverses the *seqTrie* and removes patterns whose *WES* is less than $minWES'$. Finally, it traverses the *seqTrie* to find frequent patterns whose *WES* is greater or equal to the threshold *minWES* that is *FS*. The simulation of *uWSInc* by using an example has been explained in Section 3.5 and the details of result analysis are shown in Section 4.

3.4.2. uWSInc+: incremental mining of uncertain weighted frequent sequences for better completeness

Let us consider some cases: (a) an increment to the database may introduce a new sequence which was initially absent in both *FS* and *SFS* but appeared frequently in later increments; (b) a sequence had become infrequent after an increment but could have become semi-frequent or even frequent again after next few increments. There are many real-life datasets where new frequent patterns might appear in future increments due to their seasonal behavior, different characteristics, or concept drift. The *uWSInc* algorithm does not handle these cases. To address these cases, we maintain another set of sequences denoted as promising frequent sequences (*PFS*) after each increment ΔDB introduced into *DB*. Promising frequent sequences are neither globally frequent nor semi-frequent, but their weighted expected supports satisfy a user-defined support threshold named as *LWES* that is used to find locally frequent patterns in ΔDB at a particular point. Here, whether a pattern is globally frequent (or semi-frequent) or not is determined by its frequency in the entire database, and whether a pattern is locally frequent or not is determined by its frequency in an increment. Intuitively, it can be assumed that locally frequent patterns may become globally frequent or semi-frequent after the next few increments. The patterns whose *WES* values do not meet the local threshold, *LWES*, are very unlikely to become globally frequent or semi-frequents. Thus maintaining *PFS* may significantly increase the performance of an algorithm in finding the almost complete set of frequent patterns after each increment. To incorporate the concept of promising frequent sequences (*PFS*) in mining patterns, we propose another approach, called *uWSInc+*, shown in Algorithm 4.

In Fig. 5, the determination of sequences in our proposed *uWSInc+* architecture has been presented. Each frequent sequence will be either frequent, semi-frequent, promising frequent, or infrequent after each increment. Similarly, one of the four cases will occur to each semi-frequent or promising frequent sequence. *uWSInc+* stores FS, SFS and PFS into *USeq-Trie* and maintains them for next increments. Nevertheless, when any sequence becomes an infrequent sequence, it will not be stored further. As a result, *uWSInc+* loses that information. Again, any infrequent sequence can be *PFS* or *SFS* or *FS* after several increments. Intuitively, an infrequent sequence will become *PFS* before being *SFS* or *FS* as the size of an increment is usually smaller than that of the whole database. Since *uWSInc+* stores *PFS*, consequently, it would be able to capture them. Therefore, the maintenance of *PFS* makes *uWSInc+* more robust to any concept drifts in incremental databases.

Algorithm 4: Procedure of *uWSInc+*

Input: *DB:* initial database, ΔDB : new increments, *min_sup:* minimum support threshold, μ: buffer ratio, *wgt_fct:* weight factor

Output: *FS:* set of frequent patterns

```
1  procedure InitialMining(DB, ΔDB_i, min_sup, μ, wgt_fct)
2    seqTrie ← FUWS(DB, min_sup × μ, wgt_fct)
3    pfsTrie ← Trie to store PFS, which is initialized as empty
4    for all ΔDB_i do
5      DBSize ← DBSize + ΔDB_iSize
6      seqTrie, pfsTrie ← uWSInc+(ΔDB_i, min_sup, μ, wgt_fct, seqTrie, pfsTrie)
7  procedure uWSInc+ΔDB_i, min_sup, μ, wgt_fct, seqTrie, pfsTrie
8    LWES = 2 × min_Sup × μ × ΔDB_iSize × WAM′ × wgt_fct      ▷ choice of LWES may vary
9    lfsTrie ← FUWS(ΔDB_i, 2 × min_Sup × μ, wgt_fct)
10   Call SupCalc(ΔDB_i, seqTrie)
11   Call SupCalc(ΔDB_i, pfsTrie)
12   minWES ← min_sup × DBsize × WAM × wgt_fct
13   for all α ∈ (FS ∨ SFS) stored into seqTrie do
14     if wExpSup(α) < (minWES × μ) then
15       Delete pattern α from seqTrie
16       if wExpSup(α) ⩾ LWES then
17         Insert pattern α into pfsTrie
18   for all β ∈ PFS stored into pfsTrie do
19     if wExpSup(β) ⩾ (minWES × μ) then
20       Delete pattern β from pfsTrie
21       Insert pattern β into seqTrie
22     els if wExpSup(β) < LWES then
23       Delete pattern β from pfsTrie
24   for all γ ∈ LFS stored into lfsTrie do
25     if wExpSup(γ) ⩾ (minWES × μ) then
26       Insert γ into seqTrie
27     els if wExpSup(γ) ⩾ LWES) then
28       Insert γ into pfsTrie
29 FS ← seqTrie.find_frequent_patterns(minWES)
```

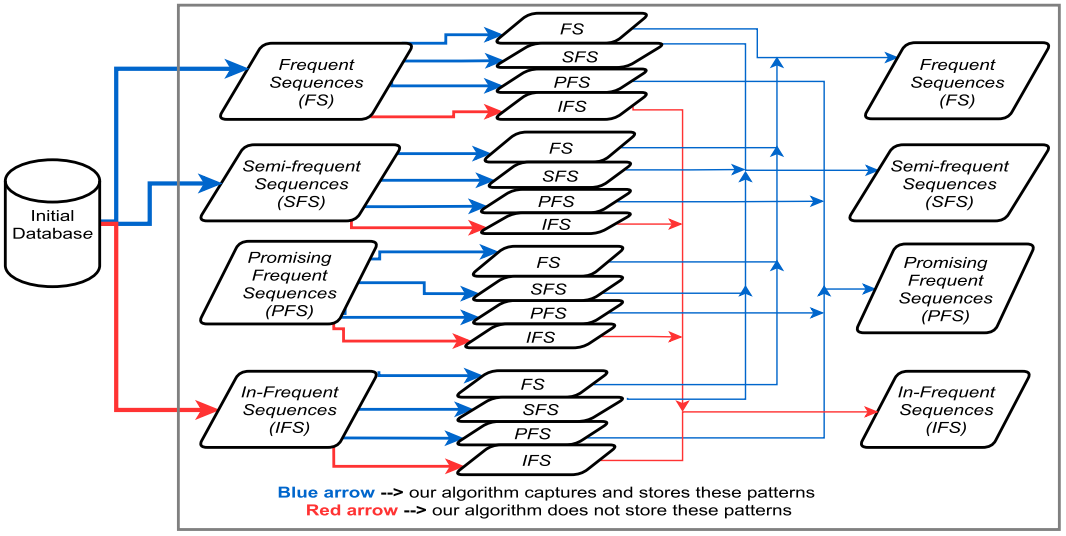


Fig. 5. Determination of sequences in our proposed *uWSInc+* architecture.

Similar to *uWSInc*, frequent and semi-frequent sequences generated by running *FUWS* on the initial database are stored as *FS* and *SFS*. For space efficiency, we store *FS* and *SFS* together into a single *USeq-Trie* instead of using two different *USeq-Trie* structures. In addition, a different *USeq-Trie*, which is initially empty, is used to store promising frequent sequences (*PFS*).

After each increment ΔDB, the steps of the algorithm are as follows:

1. Update database size, *WAM*, *minWES*, and *minWES*′.
2. Run *FUWS* only in ΔDB to find locally frequent sequences against a local threshold, *LWES* and store them into a *USeq-Trie*, named as *LFS*. Users can choose *LWES* based on the aspects of application.
3. For all α in *FS*, *SFS* and *PFS*, update WES_α by using *SupCalc* method in Algorithm 1.
   - if $WES_\alpha < LWES$, delete α's information.
   - else if $WES_\alpha < minWES'$, move α to *PFS*′.
   - else if $WES_\alpha < minWES$, move α to *SFS*′.
   - else move α to *FS*′.
4. Move each pattern α from *LFS* to *PFS*′ or *SFS*′ or *FS*′ based on WES_α.
5. Use *FS*′, *SFS*′, and *PFS*′ as *FS*, *SFS*, and *PFS* respectively for the next increment.

In Algorithm 4, *uWSInc+* stores *FS* and *SFS* into *seqTrie* which are generated from running *FUWS* once on the initial databae *DB* in Line 2. It initializes *pfsTrie* as an empty *USeq-Trie* in Line 3. Then for each increment ΔDB_i, it runs *FUWS* only in ΔDB_i to find locally weighted frequent sequences considering *LWES* and stores them into another *USeq-Trie* named as *lfsTrie* in Line 9. After that, it calls *SupCalc* function to update the weighted expected support for all patterns ($FS \vee SFS$) stored into *seqTrie* and *PFS* stored into *pfsTrie*. In Line 13–28, it updates the *seqTrie* and *pfsTrie* according to the updated *WES* of all patterns. Finally, it finds the frequent sequences *FS* by traversing the *seqTrie*.

The proposed algorithms *uWSInc* and *uWSInc+* never generate any false positive patterns which can be verified from Lemma 6.

Lemma 6. uWSInc and uWSInc + algorithms never generate any false-positive sequences. In other words, any generated sequential pattern α, that is generated by one of uWSInc algorithm or uWSInc + algorithm, must be a true-positive pattern.

Proof. Assume that a mined weighted frequent sequential pattern, α, is a false positive pattern. Then, there could be two sources of this pattern.

1. Case 1: The pattern α is mined by *uWSInc* algorithm
2. Case 2: The pattern α is mined by *uWSInc+* algorithm

As, every mined sequential pattern, α, is mined using either *uWSInc* algorithm or *uWSInc+* algorithm in the proposed incremental architectures.

According to the proposed system architecture, both *uWSInc* and *uWSInc+* approaches apply *FUWS* algorithm to mine weighted frequent sequences(*FS*) and semi-frequent sequences(*SFS*) from initial database.

Now, let us assume that a mined weighted frequent sequence, α, is a false positive pattern. Then,

1. Case 1: The sequence α is mined by *uWSInc* algorithm:
   From Algorithm 3, we can see that the *uWSInc* process applies the *FUWS* algorithm to find all the weighted frequent sequence *FS* and weighted semi-frequent sequence *SFS* from the initial database.
   If α is a sequence mined by *uWSInc*, then α must be a sequence from the set of *FS* or *SFS*. (1)
   So, a sequence α mined by *FUWS* algorithm can be a false positive sequence. (2)
   But, according to Lemma 5, all the sequences mined by *FUWS* must be true positive. (3)
   Thus, analyzing the Statements (1), (2), and (3), we can say that it is clearly a contradiction.
   $\therefore$ A sequence α, which is mined by *uWSInc* algorithm, must be a true positive sequence. And none of the mined sequences by *uWSInc* algorithm can be a false-positive sequence. (4)
2. Case 2: The sequence α is mined by *uWSInc+* algorithm:
   From Algorithm 4, we can see that all the sequences generated by the *uWSInc+* algorithm mined from one of the two processes below:
   - Process 1: using the *FUWS* algorithm to find all the weighted frequent sequence *FS* and weighted semi-frequent sequence *SFS* from the initial database
   - Process 2: measuring the frequent measure for the incremental databases, it finds two sets of sequences as local frequent sequences *LFS* and promising frequent sequence *PFS*.

   For Process 1,
   If α is a sequence mined by *uWSInc+*, then α must be a sequence from the set of *FS* or *SFS*. (5)
   So, a sequence α mined by *FUWS* algorithm can be a false positive sequence. (6)
   But, according to Lemma 5, all the sequences mined by *FUWS* must be true positive. (7)
   Thus, analyzing the Statements (5), (6) and (7), we can say that it is clearly a contradiction, too.
   For Process 2,
   It can be said that the local frequent measure lacks the global occurrences.
   Thus, all the local measures must be lower than the global measures. So, there is no chance of getting any false-positive sequences.
   $\therefore$ A sequence α which is mined by *uWSInc+* algorithm, must be a true positive sequence. And none of the mined sequences by the *uWSInc+* algorithm can be a false-positive sequence. (8)

$\therefore$ From the Statements (4) and (8), we can conclude that "*uWSInc* and *uWSInc+* algorithms never generate any false-positive sequences. In other words, any generated sequence α which is generated by the *uWSInc* or *uWSInc+* algorithms, must be a true positive sequence." □

Details of an example simulation and result analysis are shown in Section 3.5 and Section 4, respectively.

3.5. Example simulation

Let us consider Table 2 as the initial database *DB* and the increments shown in Table 4. For this simulation, set the support threshold min_sup = 20%, buffer ratio μ=0.7, and wgt_fct = 1.0. As a result, the minimum weighted expected support threshold for frequent sequences, *minWES* = 1.06 and for semi-frequent sequences, $minWES' = 0.74$.

The detailed simulations of *FUWS* on *DB* in Table 2 are shown in Fig. 6 where we compared the value of $wExpSup^{cap}$ for a sequence with $minWES'$ to find out the candidates of frequent and semi-frequent sequences together. The semi-frequent sequences will be used in our incremental techniques later.

FUWS algorithm processes *DB* in the way that has been described in Algorithm 2. The preprocessed database has been shown as *pDB* in Fig. 6. The *FUWS* algorithm uses *pDB* to find the potential candidates. An item is called s-extendable (or i-extendable) when the value of $wExpSup^{cap}$ for the sequence extension (or itemset extension) of a prefix by that item

Table 4
Increments, ΔDB_i.

Increment	Id	Sequence
ΔDB_1	7	(c:0.6, a:0.7)(a: 0.8)(f:0.9, a:0.6)
	8	(c:0.6, a:0.4)(c:0.8)(a:0.6)(f:0.5)(g:0.4, c:0.7)
	9	(f:0.8)(a:0.3)(c:0.9)(d:0.9)(f:0.5, a:0.7, d:0.4)
	10	(c:0.7)(a:0.1)(a:0.8, c:0.6, d:0.8)
ΔDB_2	11	(f:0.1)(f:0.3, c:0.7)(a:0.9)(d:0.9)(f:0.2, g:0.1)
	12	(a:0.2, c:0.1)(b:0.8)(f:0.4, e:0.4)(g:0.1)(e:0.5, g:0.2)
	13	(c:0.6)(a:0.9)(d:0.6)(e:0.6)(a:0.5, e:0.4, c:0.1)

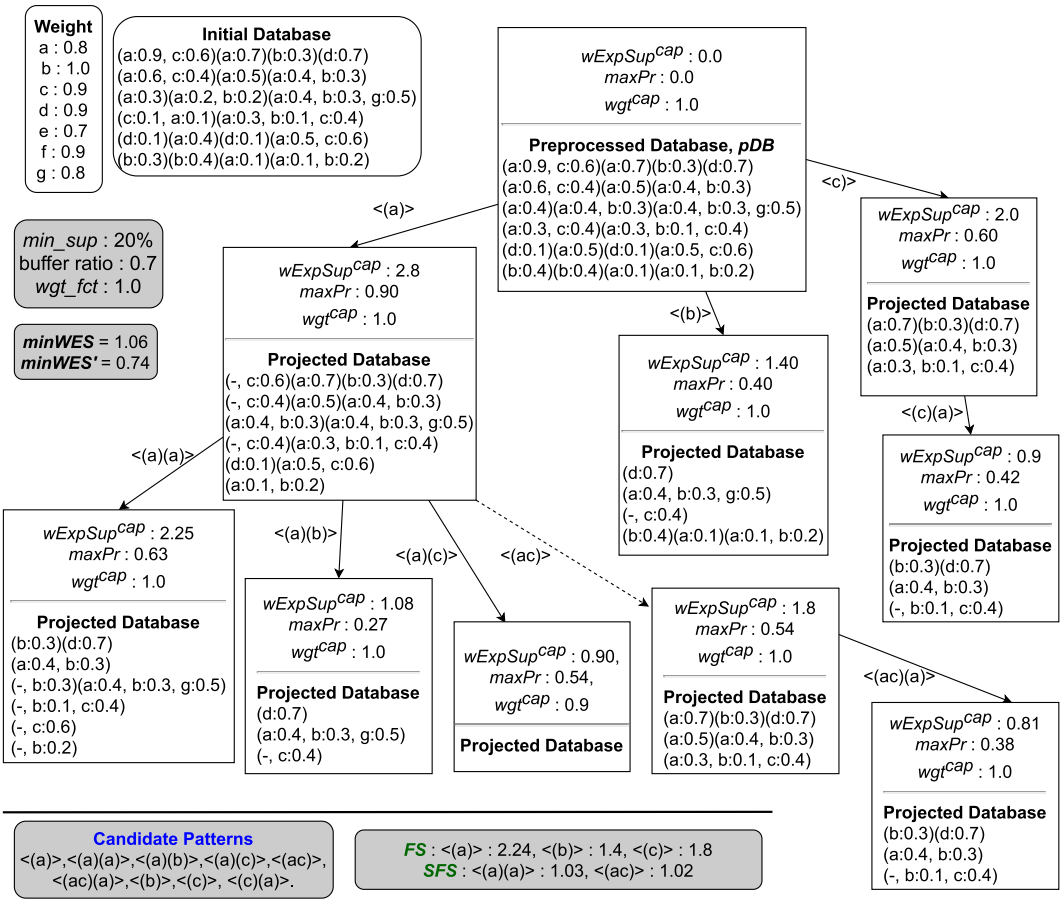


Fig. 6. FUWS simulation for Initial Database, DB in Table 2.

satisfies the threshold $minWES'$. First of all, it finds extendable items considering the prefix pattern which is empty initially. Thus the resulting super sequences are $< (a) >: 2.8, < (b) >: 1.40, < (c) >: 2.0$ where the real numbers associated with each item denote the value of $wExpSup^{cap}$ for the respective super patterns. It projects the database recursively while considering the newly generated super pattern as prefix pattern and finds out extendable items following the same process. In Fig. 6, an edge label indicates an extendable item and the edge type indicates the extension type; solid lines are for s-extensions and dashed lines are for i-extensions.

In this example, *FUWS* algorithm considers $< (a) >$ as first prefix pattern and it projects *pDB* accordingly. After that, the algorithm finds i-extendable item $< (\_c) >$ and s-extendable items for the prefix pattern $< (a) >$ which are $< (a) >, < (b) >$ and $< (c) >$. Consequently, newly generated super patterns are $< (a)(a) >: 2.25, < (a)(b) >: 1.08, < (a)(c) >: 0.90$ and $< (ac) >: 1.8$. Then it considers each super patterns as prefix pattern individually and tries to find out longer patterns by following the same process recursively. In this example, the algorithm does not find any extendable items for prefix patterns $< (a)(a) >, < (a)(b) >, < (a)(c) >$, except for $< (ac) >: 1.8$ which has only one super sequence $< (ac)(a) >: 0.81$, so then it backtracks to project the database considering $< (b) >$ as next 1-length prefix pattern. Again, there are no extendable items for $< (b) >$. It takes $< (c) >$ as next prefix pattern and finds only one extendable item, which is $< (a) >$. The algorithm repeats the same process for $< (c)(a) >$. No extendable items are found for prefix pattern $< (c)(a) >$. As there are no unexplored patterns, the recursive process terminates. It stores all potential candidate patterns into *USeq-Trie*.

To remove false positive patterns, the algorithm scans the database *DB* to calculate the actual weighted expected support *WES* for all candidates. Finally, it finds frequent and semi-frequent sequences by comparing the values of *WES* with *minWES* and $minWES'$. The resultant *FS*: $< (a) >: 2.24, < (b) >: 1.4, < (c) >: 1.8$ and *SFS*: $< (a)(a) >: 1.03, < (ac) >: 1.02$ where the real values correspond to *WES*.

After the first increment, ΔDB_1, the updated value of *minWES* is 1.74. The *uWSInc* algorithm scans the ΔDB_1 and updates the weighted expected support values for patterns in *FS* and *SFS* which are found in the initial *DB*. As a result, *FS* and *SFS* are updated as shown in Table 5. The second approach, *uWSInc+* runs *FUWS* and finds the locally frequent set, *LFS*, for ΔDB_1 using 0.96 as *LWES*. Users can set different local threshold *LWES* based on the size and nature of increments, distribution of items,

Table 5
Simulation for Increments, ΔDB_i.

Inc.		uWSInc		uWSInc+
ΔDB_1	FS:	$<(a)>: 4.56, <(a)(a)>: 1.90, <(ac)>: 1.99, <(c)>: 4.50$	LFS:	$<(a)>: 2.32, <(c)>: 2.7, <(d)>: 1.53, <(f)>: 1.98, <(a)(f)>: 0.99, <(ac)>: 0.97, <(c)(a)>: 1.83, <(c)(f)>: 1.23, <(f)(c)>: 0.96$
	SFS:	$<(b)>: 1.4$	FS:	$<(a)>: 4.56, <(a)(a)>: 1.9, <(ac)>: 1.99, <(c)>: 4.50, <(c)(a)>: 1.83, <(f)>: 1.98$
			SFS:	$<(c)(d)>: 1.23, <(b)>: 1.4, <(c)(f)>: 1.25, <(d)>: 1.53$
			PFS:	$<(a)(f)>: 0.99, <(f)(c)>: 0.96$
ΔDB_2	FS:	$<(a)>: 6.16, <(a)(a)>: 2.26, <(c)>: 5.76$	LFS:	$<(a)>: 1.6, <(a)(d)>: 1.15, <(b)>: 0.8, <(c)>: 1.26, <(d)>: 1.35, <(c)(d)>: 0.89, <(c)(a)>: 1.99, <(c)(a)(d)>: 0.77, <(e)>: 0.77$
	SFS:	$<(ac)>: 2.05, <(b)>: 2.20$	FS:	$<(a)>: 6.16, <(a)(a)>: 2.26, <(c)>: 5.76, <(c)(a)>: 2.82, <(d)>: 2.88, <(f)>: 2.61$
			SFS:	$<(ac)>: 2.05, <(b)>: 2.2, <(c)(d)>: 2.12$
			PFS:	$<(a)(d)>: 1.15, <(c)(f)>: 1.41, <(a)(f)>: 1.22, <(e)>: 0.77, <(f)(c)>: 1.03, <(c)(a)(d)>: 0.77$

etc. In this simulation, let us assume that $LWES = 2 \times min_sup \times |\Delta DB_i| \times WAM \times \mu \times wgt_fct$for ΔDB_i. By scanning ΔDB_1, it updates the WES for *FS*, and *SFS* which are found in initial database *DB*. After that *FS*, *SFS* and *PFS* have been updated according to the updated *minWES* and *LWES*. The results are shown in Table 5. From Table 5, we can see that new pattern <(c)(a)> and <(f)> appear in *FS* and $<(b)>$, <(d)>, <(c)(d)> and <(c)(f)> appear in *SFS* of *uWSInc+* but not in *uWSInc*. These patterns <(d)>, <(c)(d)> and <(c)(f)> might be frequent later after few increments. The *uWSInc+* might be able to find them which *uWSInc* could never do.

Similarly for the second increment ΔDB_2, the *uWSInc* and *uWSInc+* algorithms use *FS*, *SFS*, and *PFS* which are updated after first increment and follow the same process to generate updated *FS*, *SFS*, and *PFS*. The results are shown in Table 5. Finally, we can see that three patterns <(c)(a)>, <(d)> and <(f)> have become frequent after this increment which are found by *uWSInc+* but not *uWSInc*. This makes the difference between our two approaches clear as *uWSInc+* can find them but *uWSInc* cannot.

3.6. Analysis of time and space complexity

Before going to the experimental performance evaluation in Section 4, it is necessary to discuss the runtime and memory complexity of our proposed algorithms. We use Big-O-notation to denote the upper bound of complexity while considering that each computer operation takes approximately constant time. Here, throughout this section, M indicates the total number of sequences in the given dataset. Similarly, N – the number of nodes in *USeq-Trie*, L – the maximum length of a data sequence, D – the maximum depth of *USeq-Trie* (aka maximum length of candidate patterns), and S – the maximum size of an extendable itemset.

- SupCalc (in Algorithm 1) – This function scans the given dataset sequence by sequence and calls *TrieTraverse* function for each data sequence to calculate the support of all candidate patterns where *TrieTraverse* function takes $O(N \times L)$. Therefore, overall runtime complexity will be $O(M \times N \times L)$. Moreover, in the worst possible cases, memory complexity will be $O(D \times L)$ due to the depth-first traversal on *USeq-Trie*.
- FUWS (in Algorithm 2) – To process the whole data-set by using *preProcess* function, it may require $O(M \times L)$units of time and $O(M \times L)$ units of memory space. Then *Determine* function may take $O(M \times L)$ time along with $O(M \times S)$ unit memory. Moreover, the execution tree of the recursive process *FUWSP* could expand exhaustively, though any branch could be pruned out. Thus in the worst possible case, the runtime will be $O\left(S^D \times M \times L\right)$to find potential candidate patterns from an uncertain database.
- uWSInc (in Algorithm 3) – This algorithm will take $O(N \times M \times L)$ to find updated set of *FS* and *SFS* after updating their support using *SupCalc* function for each increment into the dataset.
- uWSInc+ (in Algorithm 4) – Since it runs *FUWS* on each increment ΔDB_i to find *LFS*, the first part of its complexity will be same as the runtime complexity of *FUWS* algorithm. Its complexity mostly depends on the complexity of *FUWS*. Besides that, the complexity for the rest of this algorithm depends on the complexity of *SupCalc* algorithm to update the weighted support of *FS*, *SFS*, and *PFS*, which are generated from previous increments along with to update the *FS*, *SFS*, and *PFS* after each increment. Approximately, it will be $O(N \times M \times L)$.

The above complexity analysis shows that our proposed algorithms can find weighted frequent sequences from both static and incremental datasets efficiently with respect to time and memory. Extensive experimental performance analysis on different real-life datasets is provided in the following section to validate this claim.

4. Performance evaluation

To evaluate our algorithms and show the effect of weights and uncertainty in data mining, we needed real-life standard datasets with noise and probability. Unfortunately none of the datasets given in data mining repositories such as SPMF[1] and FIMI[2] are uncertain. Hence, we have used the general sequence datasets after assigning weights and probabilities by using different distributions to demonstrate and analyze the performance of our proposed solutions.

Datasets. Among the datasets that we have used, *Retail*, *Foodmart*, *OnlineRetail*, and *Chainstore* are market-basket data; *Kosarak*, *FIFA*, and *MSNBC* are click stream data; *Sign* contains sign language utterance; *Accident* contains traffic accident data and *Leviathan* is a dataset of word sequences converted from a famous novel, Leviathan. However, the *Retail*, *OnlineRetail*, *Foodmart*, and *Chainstore* datasets are given in itemset format. So, we have converted them into *SPMF sequential format* where each *transaction* is a single *sequence* and its each individual *item* is considered to be a single *event*. A short description of all the tested datasets is given in Table 6.

Assignment of Probability and Weight. We have used a normal distribution to assign probability in the existing popular real-life datasets to reflect the nature of uncertainty. The normal distribution is more prominent in statistics and is widely used in the field of data mining as it fits many natural phenomena. We have assigned the existential probability to each item using a gaussian distribution with mean $\mu = 0.5$ and standard deviation $\sigma = 0.25$ for our general experimental purpose. Moreover, the weight for an item has been assigned using a gaussian distribution with $\mu = 0.5$ and $\sigma = 0.125$ as a general-purpose. Fig. 7 shows the distributions for probability and weight values in the general setting of our experiments.

Besides this general processing, we have tested the performance of our proposed algorithm with respect to different distributions of probability and weight values. We have also analyzed the results by changing different parameters to verify the correctness and efficiency of our algorithms. All of these analyses are discussed in the following sections. Section 4.1 demonstrates the performance of our static algorithm, *FUWS*, for mining *weighted/unweighted sequential patterns* from uncertain databases. Section 4.2 shows the efficiency of our proposed techniques, *uWSInc* and *uWSInc+*, for the incremental mining. We have implemented our algorithms using *Python* programming language and a machine that has Core™i5-9600U 2.90 GHz CPU with 8 GB RAM.

4.1. Performance of uncertain sequential pattern miner, FUWS

Here we provide the experimental results that show the performance of *FUWS*. We have compared the performance with the existing algorithm, *uWSequence* [39], which is discussed in Section 2.3. *uWSequence* proposed a framework where the definition of *weighted sequential pattern* is different from our proposed definition that is inspired by the widely accepted concept of *weighted support* in the literature of weighted pattern mining from precise datasets. While *uWSequence* is the current best algorithm of mining *weighted sequential patterns* from uncertain databases, the authors also showed that it outperforms the existing methods for mining *sequential patterns* in uncertain databases without weight constraint. Hence, it is sufficient to compare the performance with *uWSequence* to show the efficiency of our proposed *FUWS*. We have set the weights of all items to 1.0, which brings both algorithms under a unifying framework as the definition of the *weighted sequential pattern* differs.

Evaluation Criteria. We have considered two main criteria to evaluate the performance of *uWSequence* and *FUWS* for the same support threshold and same assignment of probability values in a dataset.

(a) **Total number of candidates generated.** Recall that both *FUWS* and *uWSequence*, like *PrefixSpan*, use different upper bounds for actual expected support value. For this reason, both of them generate some *false positive candidates* which get removed when their actual expected support values are measured through an extra scan of the database. The performance of a mining algorithm should be called superior if it generates fewer candidates than other state-of-art algorithms and finds an equal number of actual patterns.
(b) **Total running time required.** Runtime is an established criterion to evaluate the efficiency of pattern mining algorithms. The less the total required time, the more efficient the algorithm is. We conduct experiments on several real-life datasets to demonstrate the performance of *FUWS* compared to the current best state-of-art algorithm *uWSequence*.

Experiments on this section are conducted by varying the following parameters:

(a) **Minimum support threshold.** The minimum support threshold, *min_sup%*, is a value between 0.0 to 1.0, which is used to calculate the minimum (weighted) expected support threshold for determining the (weighted) sequential patterns.

[1] http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php.
[2] http://fimi.uantwerpen.be/data/.

Table 6
Dataset Description.

Dataset	Total Sequences	AverageLenth	Distinct Items	Remarks
Retail	88,163	11.306	16,471	Customar transcations data of 5 consecutive months
Kosarak	990,000	8.1	41,270	Click-stream Data from a news portal
Accidents	340,184	34.808	469	Traffic-accident data for the period 1991–2000
Chainstore	1,112,949	7.2	46,086	Customer transaction data where incremental mining can be greatly effective
Foodmart	4,141	4.424	1,559	Market-basket data with huge variety of items
OnlineRetail	541,909	4.37	2,603	A sparse market-basket dataset
Leviathan	5,834	26.34	9,025	Conversion of the novel *Leviathan* into word sequences
FIFA	26,198	34.74	2,990	click stream data from *FIFA World Cup 98* website
Sign	730	51.997	267	Sign language utterance
MSNBC	31,790	13.33	17	Click-stream data from news website

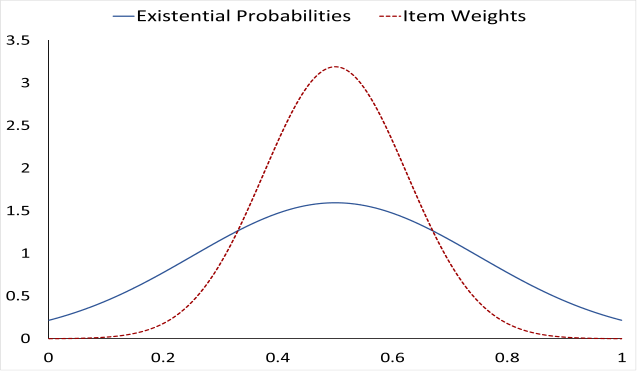


Fig. 7. Distribution of Probability Values and Item Weights.

Experiments are conducted for different *min_sup%* for each dataset to show the effectiveness of *FUWS* at any threshold based on the application requirements and discussed elaborately in Sections 4.1.1 and 4.1.2.

(b) **Probability distribution.** As the item existential probability values are assigned by ourselves, we have run both the algorithms several times with different probability distributions and discussed the experimental results in Section 4.1.3.

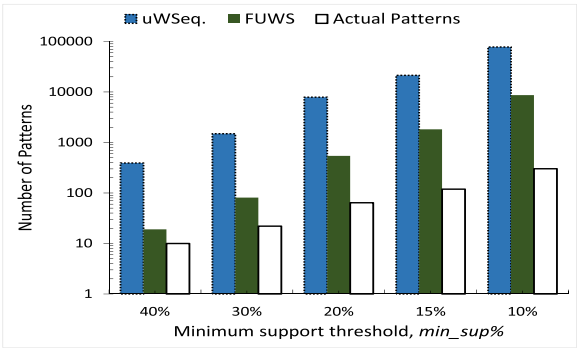
(c) **Choice of weight factor.** The parameter *wgt_fct* has been introduced to tune the mining of weighted patterns. The concept is to get patterns with more weighted expected support by setting this factor to a higher value. We have run our algorithm on several datasets with the same support threshold, same probability, and weight assignment but different weight factors to demonstrate this feature. The results are shown in Section 4.1.4.

4.1.1. Comparison of candidate generation with uWSequence

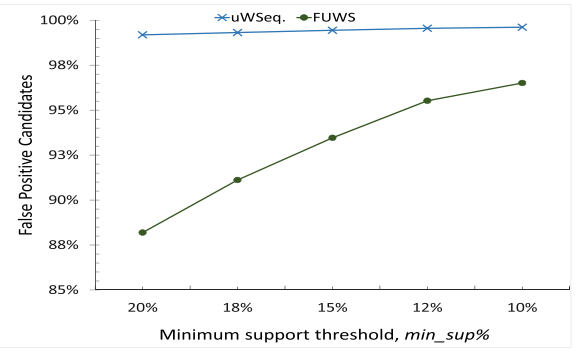
FUWS uses $expSup^{cap}$ as the upper bound measure which is theoretically tighter than the $expSupport^{top}$ used in *uWSequence*. As a result, it generates fewer false positive candidates (Lemma 2). Fig. 8a shows the comparison of candidate generation between *FUWS* and *uWSequence* in *Sign* dataset for different support thresholds. We have plotted the count of sequences in the y-axis on a logarithmic scale for a better graphical representation. As we can see, *FUWS* generates 542 candidates, where *uWSequence* generates 7874 candidates when the support threshold is 20%. However, only 64 of them are found to be weighted frequent after calculating their *actual weighted expected support* (using the efficient *SupCalc method*).

Thus, *FUWS* generates 88.19% false positive candidates whereas this number is 99.18% for *uWSequence* which is shown in Fig. 8b. The difference is about 11 percentage points in this case. We have observed that this difference gets lower when the support threshold decreases. Both *FUWS* and *uWSequence* generate more false positive candidates for a smaller support threshold.

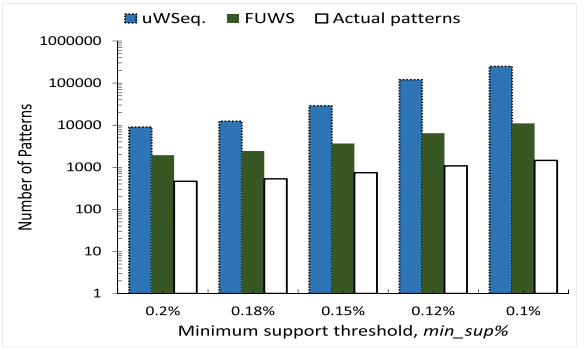
Similarly, comparison of candidate generation in other datasets such as *Kosarak*, *Retail*, *MSNBC*, *FIFA* and *Accident* are shown in Figs. 8c and 9a–d. The difference between *FUWS* and *uWSequence* is huge in the *Accident* dataset. We could not run the *uWSequence* algorithm for lower thresholds on this dataset within our 8 GB memory capacity. We were able only to find the results with a support threshold not less than 40%. From the results in all datasets, we can conclude that a strict upper bound of weighted expected support calculation leads to a significant reduction in candidate generation in the mining process. However, the ratio between the number of candidate patterns and frequent patterns increases with the decrease in the support threshold. Nonetheless, the ratio for *FUWS* is much lower than *uWSequence* for all datasets.



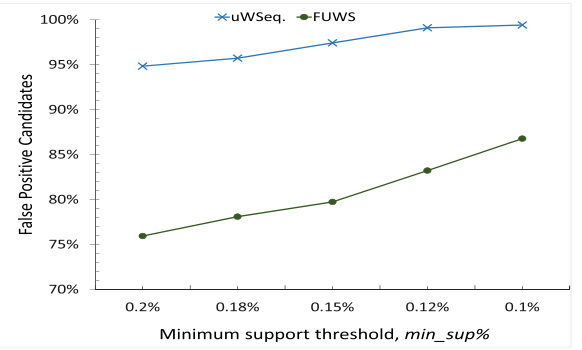
(a) Number of candidate patterns in *Sign* dataset



(b) % of false positive candidates in *Sign* dataset

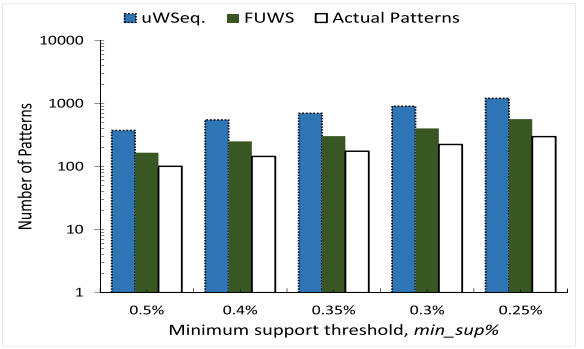


(c) Number of candidate patterns in *Kosarak* dataset

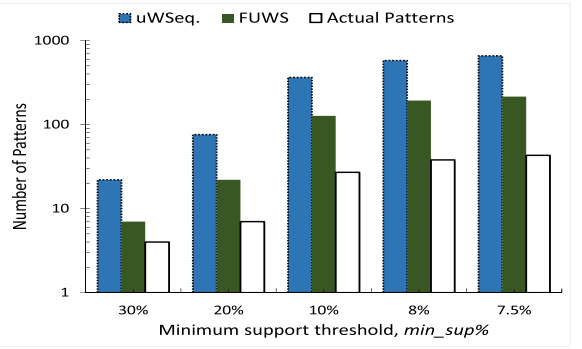


(d) % of false positive candidates in *Kosarak* dataset

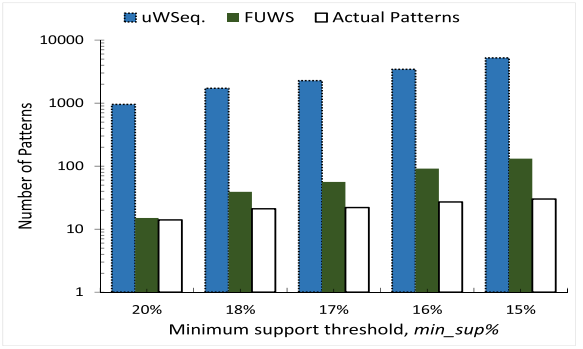
Fig. 8. Comparison of candidate generation between *FUWS* and *uWSequence* in *Sign* and *Kosarak* datasets.



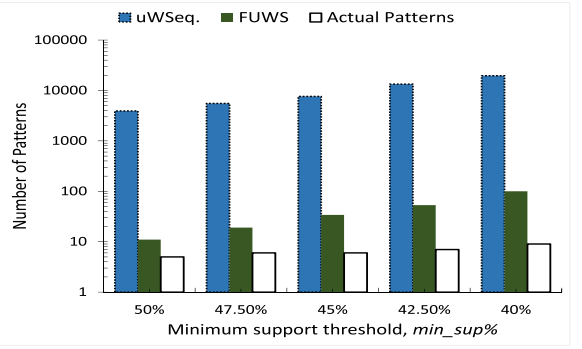
(a) *Retail* dataset



(b) *MSNBC* dataset



(c) *FIFA* dataset



(d) *Accident* dataset

Fig. 9. Comparison of candidate generation between *FUWS* and *uWSequence* in *Retail*, *MSNBC*, *FIFA*, and *Accident*.

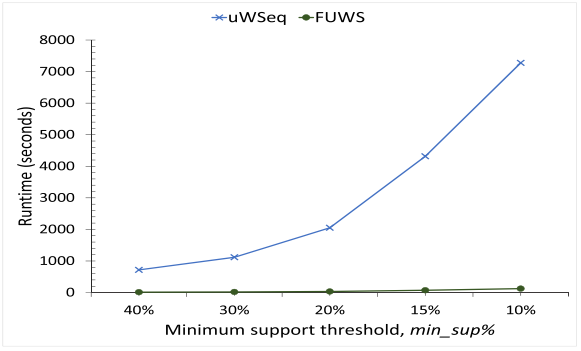
4.1.2. Comparison of runtime with uWSequence

Fig. 10a shows comparison of runtime in *Sign* dataset for different support thresholds. As we can see, the difference between the runtime of the two algorithms increases with the decrease in the support threshold. The curve representing *FUWS* rises slowly, but the curve of *uWSequence* rises up very fast with a slight decrease in the threshold. Fig. 10b shows the runtime comparison between *FUWS* and *uWSequence* in *Kosarak* dataset. It is quite similar to the result in *Sign*. The difference in the runtime between *FUWS* and *uWSequence* algorithms increases exponentially with the decreasing values of the support threshold.

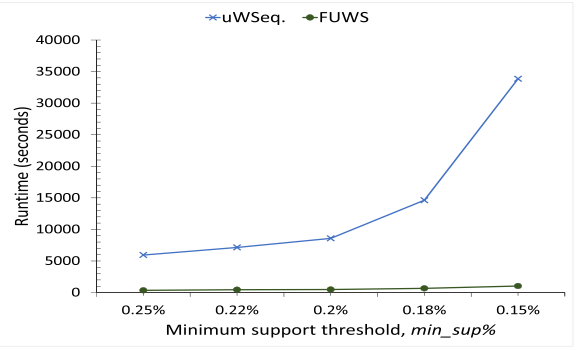
The reasons behind increasing difference in runtime are as follows,

(a) *uWSequence* generates more false-positive candidates than *FUWS* for the same support threshold.
(b) even if the number of candidates was the same, *uWSequence* would consume more time as its complexity of calculating actual weighted expected support is worse than that of *FUWS*.

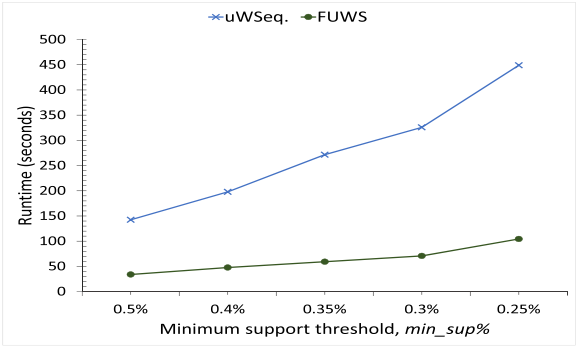
The use of faster support calculation method *SupCalc* based on *USeq-Trie* gives a benefit to *FUWS* in the latter case. Results in *Retail*, *FIFA*, *MSNBC*, and *Leivathan* datasets are also shown in Fig. 10. In every dataset, *FUWS* outperforms the existing *uWSequence* at any minimum support threshold.



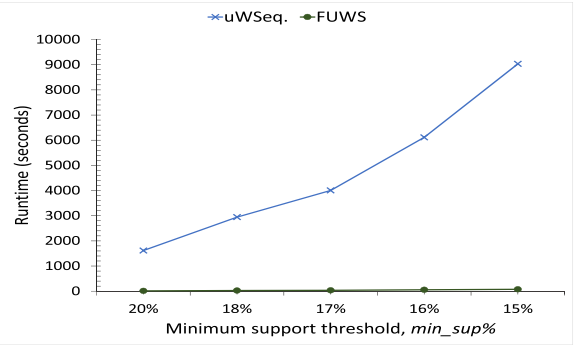
(a) *Sign* dataset



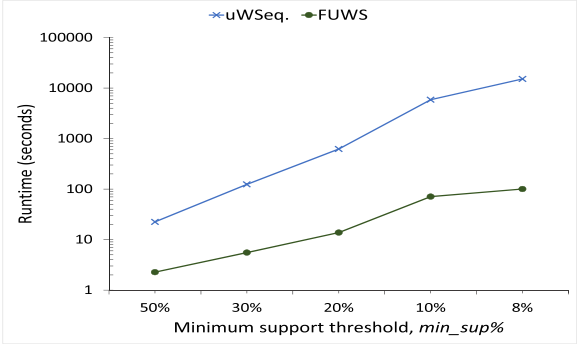
(b) *Kosarak* dataset



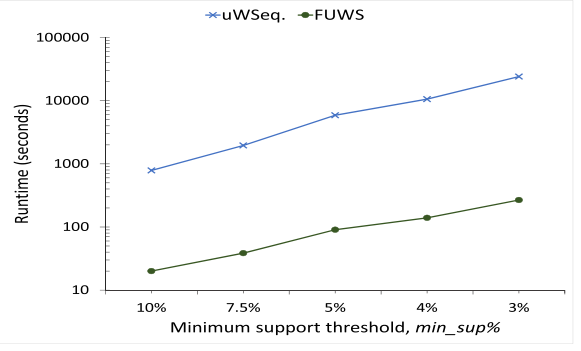
(c) *Retail* dataset



(d) *FIFA* dataset

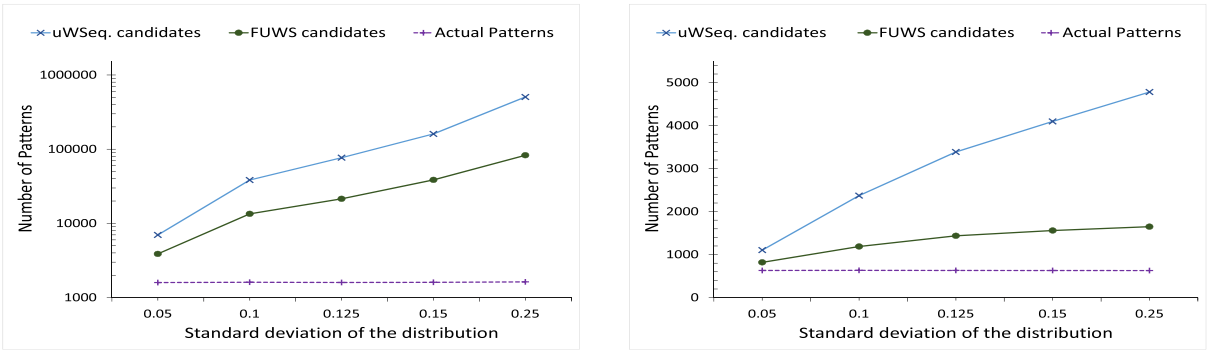


(e) *MSNBC* dataset



(f) *Leviathan* dataset

Fig. 10. Comparison of runtime between *FUWS* and *uWSequence* for various support threshold in different datastets.



(a) *Sign* dataset (b) *OnlineRetail* dataset

Fig. 11. Change in candidate generation for various distribution of probabilities in *Sign* and *OnlineRetail* datasets.

4.1.3. Analysis for different probability distributions

Fig. 11a shows the analysis for different values of standard deviation in *Sign* dataset with 5% min_sup. When the standard deviation value is large, the difference between an item's minimum and maximum existential probability in data sequences is most likely to become larger. This affects the calculation of $expSup^{cap}$ in *FUWS* and $expSupport^{top}$ in *uWSequence* for a prefix sequence. The larger standard deviation value makes the upper bound of expected support calculation less tight in both algorithms. Consequently, both algorithms generate more false-positive candidates than those with a smaller standard deviation. Fig. 11b shows a similar result in *OnlineRetail* dataset with a 0.1% support threshold. Note that, whatever the distribution is, $expSup^{cap}$ is always less than or equal to $expSupport^{top}$. Hence, *FUWS*, using $expSup^{cap}$, always generates fewer false-positive candidates than *uWSequence*. Based on the results, it can be said that *FUWS* outperforms *uWSequence* in any dataset for any kind of distribution of the uncertainty values.

4.1.4. Analysis with respect to different choice of weight factor

The results in *FIFA*, *Leviathan*, and *Retail* datasets are shown in Fig. 12 with support thresholds 15%, 5%, and 0.2% respectively. As we can see, when the weight factor is 0.75, *FUWS* finds 718 patterns to be weighted frequent. When this factor is increased to 1.25 to find patterns with more weight values, *FUWS* gives 303 patterns as weighted frequent sequences. Thus, the choice of weight factor lets the user find interesting sequences according to their necessity.

4.2. Performance of the incremental approaches, uWSInc and uWSInc+

To the best of our knowledge, there is no algorithm to mine weighted sequential patterns from incremental uncertain databases. The baseline approach is to run the *FUWS* algorithm from scratch after every increment. This section compares the two proposed algorithms with the baseline and highlights the differences between them in different datasets.

Evaluation Criteria. The criteria to evaluate the performance of incremental approaches are as follows,

(a) **Required time to find the updated result.** An incremental approach is efficient if it requires significantly less time than this baseline approach which runs the *FUWS* algorithm in the whole updated database after every increment. The baseline approach is naturally very expensive.
(b) **Completeness of the result.** The set of weighted frequent sequences found by the baseline approach is the complete set of actual patterns. The completeness of an incremental algorithm is defined to be the percentage of patterns found compared to this complete set.

Note that an incremental approach may require scanning the whole database after each increment in the worst case to ensure a complete result by an incremental solution. On the other hand, a significant improvement in the runtime can be achieved with only a small sacrifice in completeness, as we will see in the following subsections.

To use the datasets as incremental ones, we used 50% of the dataset to be the initial part. We then introduced five increments (each one randomly chosen to be 20 to 60% of the initial size). However, in the case of the *Retail* dataset, which is mentioned to be a dataset of five months, we used the first one-fifth of its transactions to be the initial portion. We then introduced four increments which roughly represent the next four months. We have also conducted experiments by varying initial size and increment size and tested the scalability. Parameters for these experiments are:

(a) **Minimum support threshold.** Experiments are conducted for different *min_sup%* for each dataset to validate the efficacy of our incremental approaches in finding almost complete results at any threshold based on the application requirements, which is discussed elaborately in Sections 4.2.1 and 4.2.2.

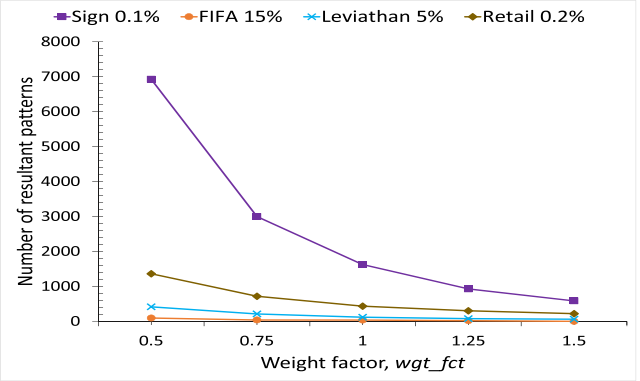


Fig. 12. Change in number of patterns for different weight factors in *Sign*, *FIFA*, *Leviathan*, and *Retail* datasets.

(b) **Buffer ratio.** Buffer ratio, μ, is used to lower the minimum weighted expected support. When $\mu = 1.0$, it means no buffer to store *semi-frequent sequences (in uWSInc, uWSInc+)*. Lower values of μ mean larger buffers. With this lowered threshold, more candidates are generated and tested. As a result, this requires more time. However, it can find more patterns using the *semi-frequent sequences*. Detailed results are shown in Section 4.2.3.
(c) **Increment size.** Many incremental algorithms have the limitation that they do not perform well beyond a certain increment size. This is also called update ratio, i.e., $\frac{(size\ of\ \Delta DB)}{(size\ of\ initial\ DB)}$. We have run our algorithms several times by changing the update ratio when other parameters are fixed and showed the efficiency in Section 4.2.4.
(d) **Dataset size.** Besides the increment size, this is another form of testing the scalability of the incremental approach. We have gradually increased the dataset and plotted the total time needed to find the updated result after each increment starting from an initial dataset. The update ratio is not fixed in this case, rather drawn from a range of 0.2 to 0.6 randomly. Results in Section 4.2.5 show how the algorithms performs for such scaled datasets.
(e) **Intial dataset size.** Existing incremental mining algorithms focus on almost complete results by only buffering *semi-frequent sequences (SFS)* depend on the initial dataset size. Thus, we develop our *uWSInc+* algorithm to overcome this limitation. Hence, we have compared between *uWSInc* (buffers *SFS* only) and *uWSInc+* (along with *SFS*, it buffers extra promising frequent sequences that are mined locally in the increments) in Section 4.2.6 by setting different initial dataset size when other parameters are fixed.

4.2.1. Runtime analysis with respect to support threshold

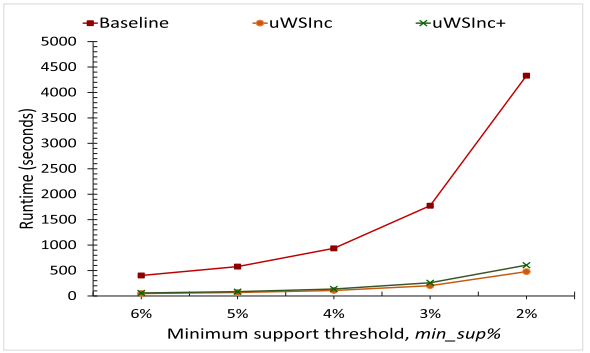
To analyze the runtime with respect to different support thresholds, we have run the baseline approach, *uWSInc*, and *uWSInc+* in a database several times and exhibited the average runtime for each support threshold. Total time required by an approach in the dataset for a single support threshold is the sum of the amount of time required after each increment plus the amount of time required for mining in the initial part. Fig. 13a shows the result in the *Leviathan* dataset for different support thresholds. As we can see, at 6% support threshold, it requires 402.36 s in baseline approach whereas *uWSInc* and *uWSInc+* takes only 47.94 and 58.67 s, respectively.

uWSInc takes 8.39 times less time than the baseline approach; for *uWSInc+*, this number is 6.86. The difference between *uWSInc* runtime and baseline is around 354.42 s and the difference between *uWSInc* and *uWSInc+* is around 10.73 s. These differences with the baseline change very rapidly with a slight change in the support threshold. In the *Retail* dataset, when the minimum support threshold is 0.4%, the difference between *uWSInc* and the baseline is 159.73 s, the difference between *uWSInc+* and the baseline is 137.27 s, and the difference between *uWSInc* and *uWSInc+* is 22.46 s as depicted in Fig. 13c. When the threshold decreases to 0.2%, the above differences rise to 428.51, 360.88, and 67.63 s respectively. Fig. 13e shows the runtime analysis with respect to minimum support threshold in *MSNBC* dataset which is similar to the result of the *Leviathan* dataset.

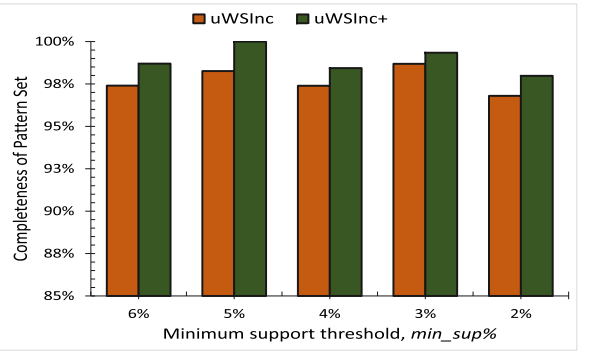
4.2.2. Analysis of completeness with respect to different support thresholds

Fig. 13b shows the completeness comparison between *uWSInc* and *uWSInc+* in *Leviathan* dataset for support threshold values ranging between 2% and 6%. As we can see, at any support threshold point, *uWSInc+* gives better completeness. Fig. 13d shows the difference in completeness between *uWSInc* and *uWSInc+* in *Retail* dataset for different support thresholds. The difference is more understandable than that was seen in *Leviathan* dataset. *Foodmart* dataset has results similar to *Retail* as shown in Fig. 14b. Datasets like *Leviathan* and *Bible* [3] contain almost all the patterns (frequent word sequences) in their initial 50% portion. Any new word that appears in future increments generally does not have enough support to become frequent. Thus, the completeness of our two approaches is very close in these cases for different min_sup values. On the other

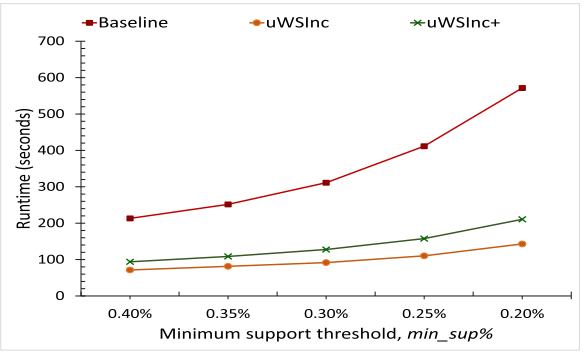
[3] Details of *Leviathan*, *Bible*, *Retail*, and *Foodmart* dataset can be found at http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php



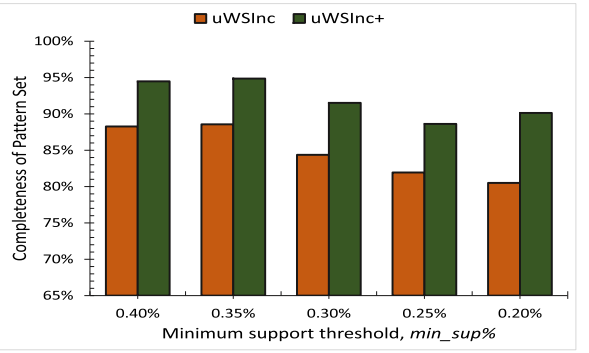
(a) Runtime in *Leviathan*



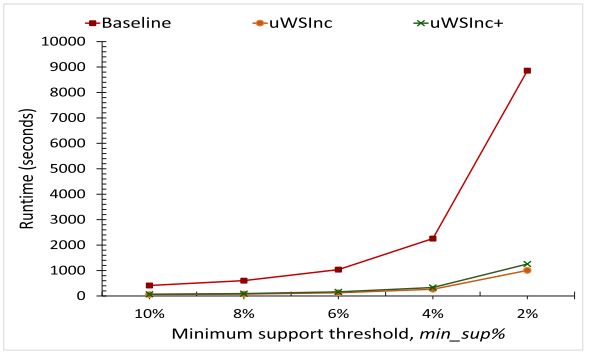
(b) Completeness in *Leviathan*



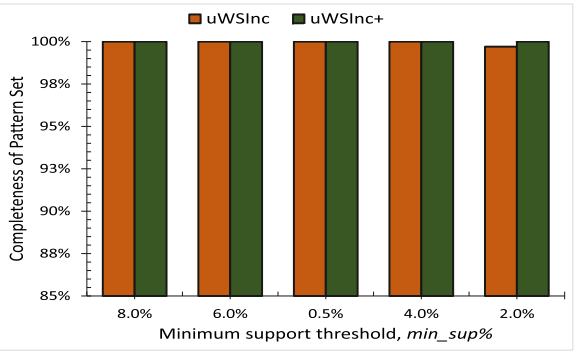
(c) Runtime in *Retail*



(d) Completeness in *Retail*



(e) Runtime in *MSNBC* dataset



(f) Completeness in *MSNBC* dataset

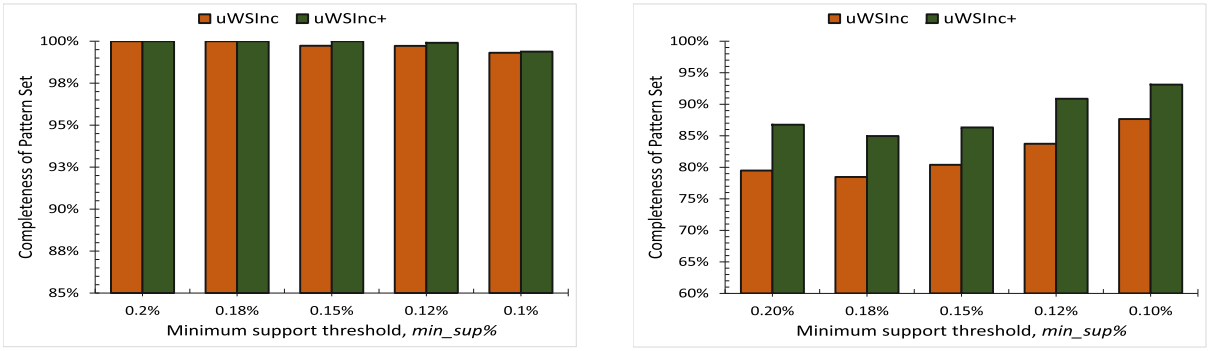
Fig. 13. Performance analysis of *uWSInc* and *uWSInc+* against various support threshold.

hand, market basket datasets like *Retail* and *Foodmart* have scenarios that any item that was initially infrequent or absent can come up in future increments with enough support to be frequent. Most of these new patterns can be found by *uWSInc+* where *uWSInc* can find none of them. Thus, a significant difference in completeness is found in these datasets. Completeness of the result for both algorithm achieve completeness very close to 100% in datasets like *MSNBC* and *Kosarak* as shown in Figs. 13f and 14a.

4.2.3. Analysis with respect to buffer ratio

An incremental mining algorithm that consumes reasonably more time might be preferred to another only if it gives better completeness. Nevertheless, it is a matter of deciding what is an acceptable level of sacrifice. This decision may depend on many factors that vary from user to user. As we have seen runtime and completeness difference between *uWSInc* and *uWSInc +* for varying support thresholds in *Leviathan*, *Retail*, and *Foodmart* dataset, let us discuss the effect of buffer ratio in them while using incremental approaches.

The change in runtime between *uWSInc* and *uWSInc+* in *Leviathan* is shown Fig. 15a. This figure validates our claim as stated above. As we can see, when we use no buffer, i.e., *buffer ratio* = 1.0, both *uWSInc* and *uWSInc+* consume a very short



(a) *Kosarak* dataset (b) *Foodmart* dataset

Fig. 14. Completeness of result in *Kosarak* and *Foodmart* datasets against various support threshold.

amount of time compared To the baseline. However, there is a slight difference between their runtimes. *uWSInc+* takes slightly more time because it has to run *FUWS* locally in each increment that occurs and maintains promising frequent sequences. *uWSInc* does not need this kind of step at all. It is easily observed that the runtime of both *uWSInc* and *uWSInc+* increases with the decrease in buffer ratio. At the same time, the decreased buffer ratio helps to achieve more completeness by maintaining extra sequences. which can be observed in Fig. 15b. For datasets like *Leviathan and Kosarak*, an average choice of buffer ratio = 0.85 gives a satisfactory result.

From Fig. 15c and d, we can point out an interesting case. *uWSInc* achieves around 91% completeness using buffer ratio = 0.5 and it takes around 168 s. Whereas, a result of the same completeness can be achieved by *uWSInc+* in around 127 s by using buffer ratio = 0.85. *Foodmart* is also a market basket dataset like *Retail*. However, in this dataset, both the initial part and the increments are small in size. Runtime and completeness analysis in *Foodmart* for different choice of buffer ratio is shown in Fig. 15e and f respectively. In every case, both runtime and completeness for *uWSInc+* is larger than that of *uWSInc*. Based on the results here, we can conclude that a lower value of buffer ratio gives better completeness, but at the same time, it also increases the amount of time required to generate the result.

4.2.4. Increment size

To evalulate the effect of increment size, we have run our algorithms several times with different *update ratios* ranging from 0.1 to 0.5 in the *Chainstore* dataset and from 0.025 to 0.3 in the *Kosarak* dataset. Here, the size of the initial dataset is 20,000 sequences. Hence, *update ratio* $= 0.2$ means an increment of $20,000 \times 0.2 = 4000$ new sequences. Results in Figs. 16 and 17 show that both of our incremental algorithms are very efficient not only for smaller increments but also for larger increments. However, the choice between *uWSInc* and *uWSInc+* can be made considering the trade-off between *runtime* and *completeness* as discussed in previous sections. We highlight that the efficiency of our incremental mining algorithms is not limited by the update ratio or the total number of increments. Hence, they are highly scalable.

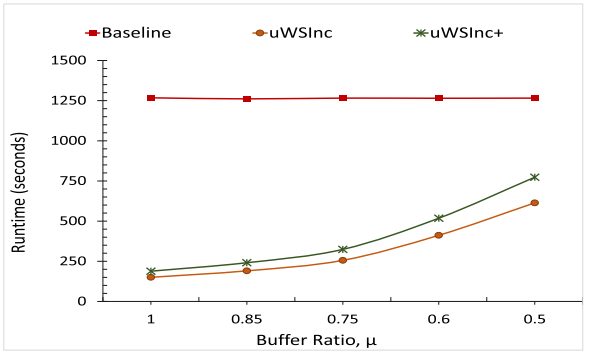
4.2.5. Dataset size

To test scalability against the dataset size, we have run our proposed algorithms and the baseline approach in several large datasets such as *Chainstore* and *Kosarak*. We have considered the first 10 thousand transactions as the initial dataset and then introduced several increments of varying sizes to use the full-length dataset.

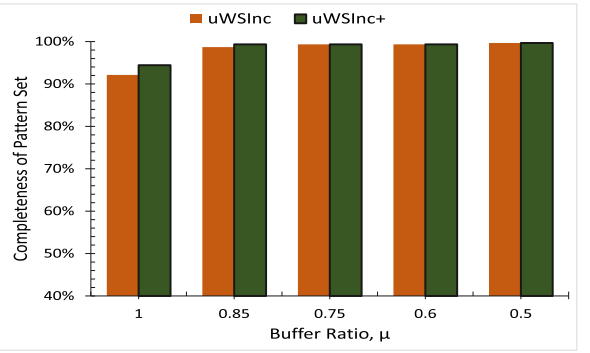
Fig. 18a shows the performance analysis for the *Chainstore* dataset with *min_sup* 0.05%. Fig. 18b shows the scalability performance for the *Kosarak* dataset with *min_sup* 0.1%. Both *uWSInc* and *uWSInc+* take an equal amount of time in the initial phase, and it is slightly higher than the baseline approach. The reason is that our proposed algorithms find and store additional patterns in the initial phase compared to the baseline approach for the same support threshold. Later on, for each increment, the baseline approach runs *FUWS* from scratch in the updated database that leads to consuming huge time and memory. In contrast, our proposed algorithms consume less time. Thus, both our algorithms outperform the baseline approach in large datasets and are efficient in handling multiple updates.

4.2.6. Completeness analysis with different initial size of database

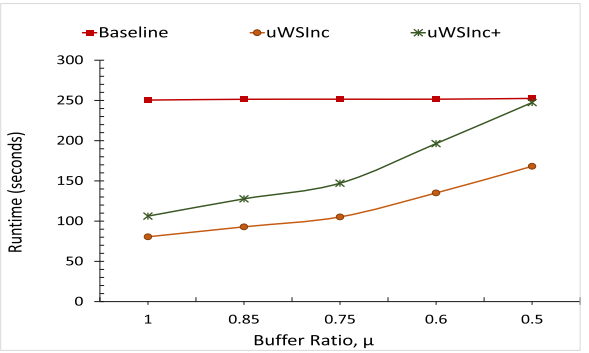
The completeness of our proposed algorithms varies with the initial size of the datasets. The larger the initial size, the more patterns they can find and maintain to update in future increments. The smaller the initial size, the more initially infrequent sequences are found as new frequent patterns after successive increments. We have considered different sizes of the initial dataset, such as the first 10,000, first 20,000, first 30,000, and so on. After that, we have introduced a sufficient number of increments to cover the entire dataset. The size of increments was chosen randomly each time from the range of *update ratio 0.4–0.8* to reflect the real-life use cases. Results for the *Chaninstore* and *OnlineRetail* datasets are shown in Fig. 19. It can be seen that *uWSInc* has a positive trend, which indicates greater completeness, with the larger initial dataset. Because



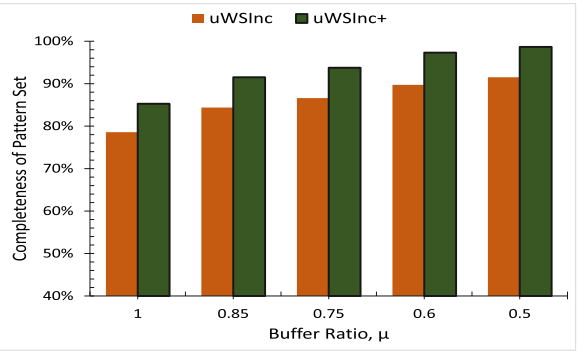
(a) Runtime in incremental **Leviathan**, min_sup 3%



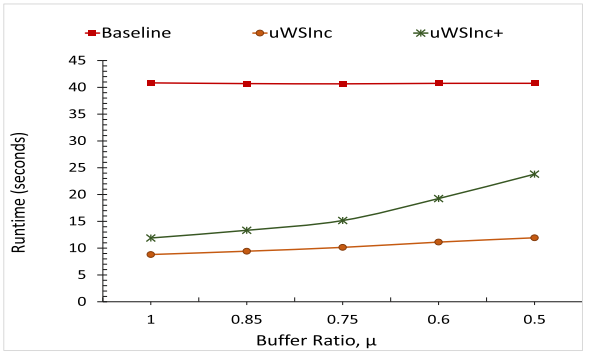
(b) Completeness in incremental **Leviathan**, min_sup 3%



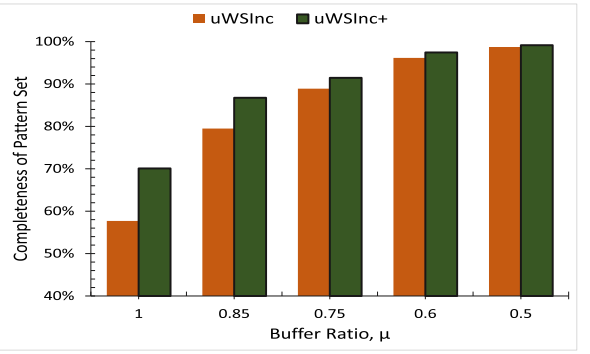
(c) Runtime in incremental **Retail**, min_sup 0.3%



(d) Completeness in incremental *Retail*, min_sup 0.3%



(e) Runtime in incremental *Foodmart*, min_sup 0.2%



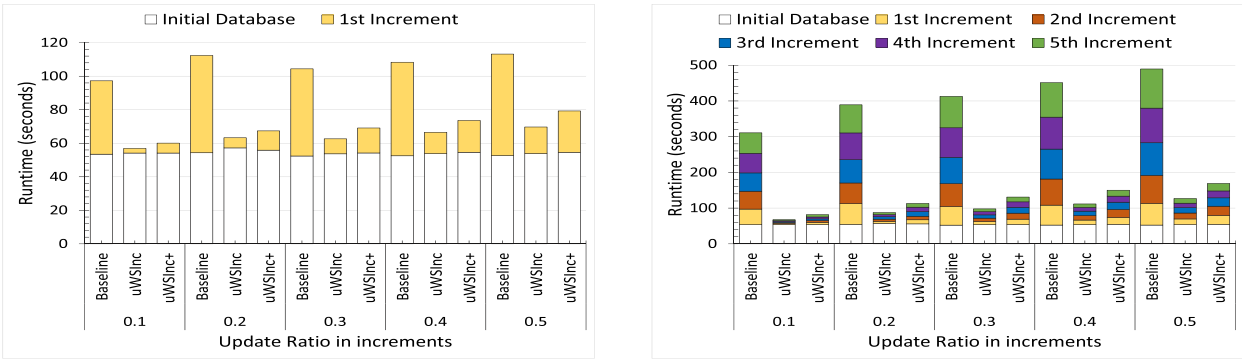
(f) Completeness in *Foodmart*, min_sup 0.2%

Fig. 15. Performance analysis of *uWSInc* and *uWSInc+* for different buffer ratio.

uWSInc updates the result after each increment based on only the semi-frequent patterns stored by mining in the initial phase, a larger initial dataset helps to find and store more potential patterns.

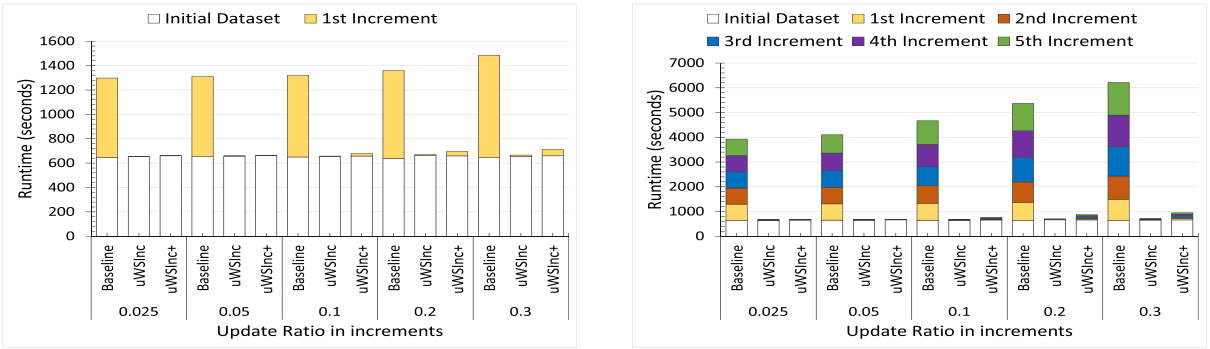
On the other hand, *uWSInc+* is less dependent on the initial size as it also mines locally in the incremented portions, which helps to find new patterns. It also keeps track of patterns that have become infrequent recently and uses them as *promising frequent sequences* in the next increment. Thus, the trend for completeness of *uWSInc+* is somewhat neutral with respect to the initial dataset size, which means that the completeness of *uWSInc+* is less affected by the size of initial datasets. The completeness of incremental approaches also depends on the distribution of items among the increments. However, for any initial dataset size, completeness values are always higher for *uWSInc+* than for *uWSInc*.

Our extensive experimental analysis demonstrates that our proposed *FUWS* outperforms the existing best solution *uWSequence* [39] to mine frequent sequences in uncertain databases. The results also validate the efficiency of *uWSInc* and *uWSInc+*; that they can find the almost complete set of frequent patterns within a very short amount of time after each increment is introduced. Finally, one thing to highlight is that though *uWSInc+* is better than *uWSInc* in terms of completeness, *uWSInc* is faster than *uWSInc+*.



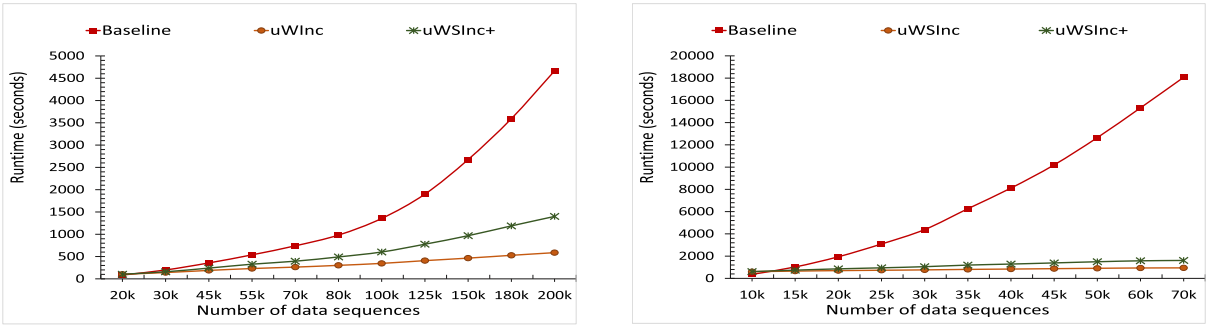
(a) Upto first increment (b) Upto five increments

Fig. 16. Performance of incremental solution for different increment size in *Chainstore* dataset with *min_sup* 0.01%.



(a) Upto first increment (b) Upto five increments

Fig. 17. Performance of incremental solution for different increment size in *Kosarak* dataset with *min_sup* 0.03%.

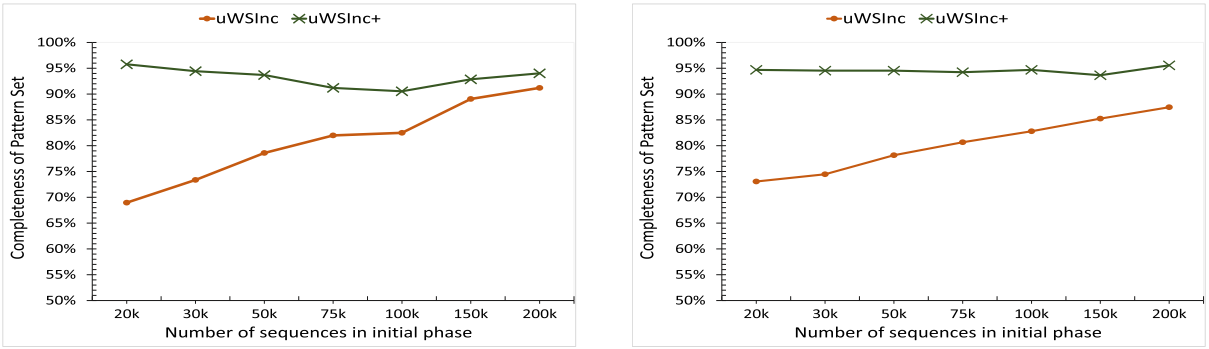


(a) *Chainstore* dataset with *min_sup* 0.05% (b) *Kosarak* dataset with *min_sup* 0.1%

Fig. 18. Performance of incremental solution for increasing database size.

5. Conclusions

In this work, we have developed an algorithm, *FUWS*, to mine weighted sequential patterns in uncertain databases and proposed two new incremental mining approaches, *uWSInc* and *uWSInc+*, to mine weighted sequential patterns efficiently from incremental uncertain databases. The *FUWS* algorithm applies $wExpSup^{cap}$ as an upper bound of weighted expected support to find all potential frequent sequences and then prunes false-positive sequences. We have used a hierarchical index structure named *USeq-Trie* to maintain patterns and *SupCalc* to calculate their support in a faster way. By using *FUWS* as a tool and buffering semi-frequent sequences, the *uWSInc* algorithm works efficiently in mining frequent sequences from



(a) *Chainstore* dataset with *min_sup* 0.05%

(b) *OnlineRetail* dataset with *min_sup* 0.05%

Fig. 19. Performance of incremental solution for different dataset size of initial phase.

incremental uncertain databases which have a uniform distribution of items. In the case of those datasets, the *uWSInc* algorithm is very efficient because most of the frequent sequences are either found in the initial dataset or will come from semi-frequent sequences, and the appearance of new items after increments are sporadic. On the other hand, due to seasonal behavior, concept drifts, or different characteristics of datasets, new patterns can be largely introduced in some real-life datasets. In those cases, the *uWSInc+* algorithm maintains promising sequences after each increment, additionally along with semi-frequent sequences to find new patterns effectively. We have tested them in many real-life and popular datasets by varying different parameters to prove their efficiency. These results show that our proposed techniques could be an excellent tool for many real-life applications that use uncertain sequential data, such as medical reports, sensor data, image processing data, social network data, privacy-preserving data, and so on. In the future, this work can be extended to mine weighted sequential patterns in uncertain data streams. Furthermore, incremental mining of maximal and closed sequential patterns can be interesting for further research.

CRediT authorship contribution statement

Kashob Kumar Roy: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft. **Md Hasibul Haque Moon:** Conceptualization, Methodology, Validation, Formal analysis, Investigation, Data curation, Writing – original draft. **Md Mahmudur Rahman:** Conceptualization, Methodology, Formal analysis, Investigation, Resources, Writing – original draft. **Chowdhury Farhan Ahmed:** Supervision, Conceptualization, Resources, Writing – review & editing. **Carson Kai-Sang Leung:** Conceptualization, Writing – review & editing, Funding acquisition.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this article.

Acknowledgement

We would like to express our deep gratitude to the anonymous reviewers of this article. Their insightful comments have played a significant role in improving the quality of this work. This work is partially supported by NSERC (Canada) and the University of Manitoba.

References

[1] C.C. Aggarwal, Y. Li, J. Wang, J. Wang, Frequent pattern mining with uncertain data, in: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, 2009, pp. 29–38.
[2] R. Agrawal, R. Srikant, et al., Fast algorithms for mining association rules, in: Proc. 20th int. conf. very large data bases, VLDB, vol. 1215, 1994, pp. 487–499..
[3] A.U. Ahmed, C.F. Ahmed, M. Samiullah, N. Adnan, C.K.-S. Leung, Mining interesting patterns from uncertain databases, Inf. Sci. 354 (2016) 60–85.
[4] C.F. Ahmed, S.K. Tanbeer, B.-S. Jeong, Y.-K. Lee, H.-J. Choi, Single-pass incremental and interactive mining for weighted frequent patterns, Expert Syst. Appl. 39 (9) (2012) 7976–7994.
[5] Y. Chen, J. Guo, Y. Wang, Y. Xiong, Y. Zhu, Incremental mining of sequential patterns using prefix tree, in: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2007, pp. 433–440.
[6] H. Cheng, X. Yan, J. Han, IncSpan: incremental mining of sequential patterns in large database, in: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2004, pp. 527–532.
[7] D.W. Cheung, J. Han, V.T. Ng, C. Wong, Maintenance of discovered association rules in large databases: An incremental updating technique, in: Proceedings of the twelfth international conference on data engineering, IEEE, 1996, pp. 106–114.

[8] U. Ahmed, J.C.-W. Lin, G. Srivastava, R. Yasin, Y. Djenouri, An evolutionary model to mine high expected utility patterns from uncertain databases, IEEE Trans. Emerg. Top. Comput. Intell. 5 (1) (2020) 19–28.
[9] Chien-Ming Chen, L. Chen, W. Gan, L. Qiu, W. Ding, Discovering high utility-occupancy patterns from uncertain data, Inf. Sci. 546 (2021) 1208–1229.
[10] W. Gan, J.C.-W. Lin, P. Fournier-Viger, H.-C. Chao, J.M.-T. Wu, J. Zhan, Extracting recent weighted-based patterns from uncertain temporal databases, Eng. Appl. Artif. Intell. 61 (2017) 161–172.
[11] Gautam Srivastava, J.C. Lin, A. Jolfaei, Y. Li, Y. Djenouri, Uncertain-driven analytics of sequence data in IoCV environments, IEEE Trans. Intell. Transp. Syst. 22 (8) (2021) 5403–5414.
[12] J.C.-W. Lin, W. Gan, P. Fournier-Viger, H.-C. Chao, T.-P. Hong, Efficiently mining frequent itemsets with weight and recency constraints, Appl. Intell. 47 (3) (2017) 769–792.
[13] Razieh Davashi, ILUNA: single-pass incremental method for uncertain frequent pattern mining without false positives, Inf. Sci. 564 (2021) 1–26..
[14] Tin C. Truong, H.V. Duong, B. Le, P. Fournier-Viger, EHAUSM: an efficient algorithm for high average utility sequence mining, Inf. Sci. 515 (2020) 302–323.
[15] Wensheng Gan, J.C. Lin, J. Zhang, H. Chao, H. Fujita, P.S. Yu, ProUM: projection-based utility mining on sequence data, Inf. Sci. 513 (2020) 222–240.
[16] P. Fournier-Viger, J.C.-W. Lin, R.U. Kiran, Y.S. Koh, R. Thomas, A survey of sequential pattern mining, Data Sci. Pattern Recogn. 1 (1) (2017) 54–77.
[17] W. Gan, J.C.-W. Lin, P. Fournier-Viger, H.-C. Chao, T.-P. Hong, H. Fujita, A survey of incremental high-utility itemset mining, Wiley Interdiscip. Rev.: Data Min. Knowl. Discov. 8 (2) (2018) e1242.
[18] W. Gan, J.C.-W. Lin, P. Fournier-Viger, H.-C. Chao, V.S. Tseng, Mining high-utility itemsets with both positive and negative unit profits from uncertain databases, in: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2017, pp. 434–446.
[19] W. Gan, J.C.-W. Lin, P. Fournier-Viger, H.-C. Chao, P.S. Yu, A survey of parallel sequential pattern mining, ACM Trans. Knowl. Discov. Data 13 (3) (2019) 1–34.
[20] J. Han, J. Pei, Y. Yin, R. Mao, Mining frequent patterns without candidate generation: a frequent-pattern tree approach, Data Min. Knowl. Discov. 8 (1) (2004) 53–87.
[21] S.Z. Ishita, F. Noor, C.F. Ahmed, An efficient approach for mining weighted sequential patterns in dynamic databases, in: Industrial Conference on Data Mining, Springer, 2018, pp. 215–229.
[22] G. Lee, U. Yun, Single-pass based efficient erasable pattern mining using list data structure on dynamic incremental databases, Future Gener. Comput. Syst. 80 (2018) 12–28.
[23] G. Lee, U. Yun, H. Ryang, An uncertainty-based approach: frequent itemset mining from uncertain data with different item importance, Knowl.-Based Syst. 90 (2015) 239–256.
[24] C.K.-S. Leung, R.K. MacKinnon, F. Jiang, Reducing the search space for big data mining for interesting patterns from uncertain data, in: 2014 IEEE International Congress on Big Data, IEEE, 2014, pp. 315–322.
[25] C.K.-S. Leung, M.A.F. Mateo, D.A. Brajczuk, A tree-based approach for frequent pattern mining from uncertain data, in: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2008, pp. 653–661.
[26] C.K.-S. Leung, S.K. Tanbeer, Fast tree-based mining of frequent itemsets from uncertain data, in: International Conference on Database Systems for Advanced Applications, Springer, 2012, pp. 272–287.
[27] C.K.-S. Leung, S.K. Tanbeer, PUF-tree: a compact tree structure for frequent pattern mining of uncertain data, in: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2013, pp. 13–25.
[28] C.-S. Leung, Q.I. Khan, T. Hoque, CanTree: a tree structure for efficient incremental mining of frequent patterns, in: Fifth IEEE International Conference on Data Mining (ICDM'05), IEEE, 2005, pp. 274–281.
[29] J.C.-W. Lin, W. Gan, P. Fournier-Viger, T.-P. Hong, V.S. Tseng, Weighted frequent itemset mining over uncertain databases, Appl. Intell. 44 (1) (2016) 232–250.
[30] J.C.-W. Lin, T.-P. Hong, W. Gan, H.-Y. Chen, S.-T. Li, Incrementally updating the discovered sequential patterns based on pre-large concept, Intell. Data Anal. 19 (5) (2015) 1071–1089.
[31] J.C.-W. Lin, T. Li, M. Pirouz, J. Zhang, P. Fournier-Viger, High average-utility sequential pattern mining based on uncertain databases, Knowl. Inf. Syst. 62 (3) (2020) 1199–1228.
[32] J.C.-W. Lin, M. Pirouz, Y. Djenouri, C.-F. Cheng, U. Ahmed, Incrementally updating the high average-utility patterns with pre-large concept, Appl. Intell. 50 (11) (2020) 3788–3807.
[33] J.C.-W. Lin, J.M.-T. Wu, P. Fournier-Viger, C.-H. Chen, T. Li, A project-based PMiner algorithm in uncertain databases, in: 2019 International Conference on Technologies and Applications of Artificial Intelligence (TAAI), IEEE, 2019, pp. 1–5.
[34] M. Muzammal, R. Raman, On probabilistic models for uncertain sequential pattern mining, in: International Conference on Advanced Data Mining and Applications, Springer, 2010, pp. 60–72.
[35] M. Muzammal, R. Raman, Mining sequential patterns from probabilistic databases, in: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2011, pp. 210–221..
[36] H. Nam, U. Yun, E. Yoon, J.C.-W. Lin, Efficient approach for incremental weighted erasable pattern mining with list structure, Expert Syst. Appl. 143 (2020) 113087.
[37] S.N. Nguyen, X. Sun, M.E. Orlowska, Improvements of IncSpan: incremental mining of sequential patterns in large database, in: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2005, pp. 442–451.
[38] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, M.-C. Hsu, Mining sequential patterns by pattern-growth: the PrefixSpan approach, IEEE Trans. Knowl. Data Eng. 16 (11) (2004) 1424–1440.
[39] M.M. Rahman, C.F. Ahmed, C.K.-S. Leung, Mining weighted frequent sequences in uncertain databases, Inf. Sci. 479 (2019) 76–100.
[40] R.A. Rizvee, M.F. Arefin, C.F. Ahmed, Tree-Miner: Mining sequential patterns from SP-Tree, in: PAKDD, Springer, 2020, pp. 44–56..
[41] R. Srikant, R. Agrawal, Mining sequential patterns: Generalizations and performance improvements, in: International Conference on Extending Database Technology, Springer, 1996, pp. 1–17.
[42] S.K. Tanbeer, C.F. Ahmed, B.-S. Jeong, Y.-K. Lee, Efficient single-pass frequent pattern mining using a prefix-tree, Inf. Sci. 179 (5) (2009) 559–583.
[43] T. Truong-Chi, P. Fournier-Viger, A survey of high utility sequential pattern mining, in: High-Utility Pattern Mining, Springer, 2019, pp. 97–129..
[44] J.-Z. Wang, J.-L. Huang, On incremental high utility sequential pattern mining, ACM Transactions on Intelligent Systems and Technology (TIST) 9 (5) (2018) 1–26.
[45] L. Wang, D.W.-L. Cheung, R. Cheng, S.D. Lee, X.S. Yang, Efficient mining of frequent item sets on large uncertain databases, IEEE Trans. Knowl. Data Eng. 24 (12) (2011) 2170–2183.
[46] J.M.-T. Wu, Q. Teng, J.C.-W. Lin, C.-F. Cheng, Incrementally updating the discovered high average-utility patterns with the pre-large concept, IEEE Access 8 (2020) 66788–66798.
[47] D. Yan, Z. Zhao, W. Ng, S. Liu, Probabilistic convex hull queries over uncertain data, IEEE Trans. Knowl. Data Eng. 27 (3) (2014) 852–865.
[48] U. Yun, Efficient mining of weighted interesting patterns with a strong weight and/or support affinity, Inf. Sci. 177 (17) (2007) 3477–3499.
[49] U. Yun, A new framework for detecting weighted sequential patterns in large sequence databases, Knowl.-Based Syst. 21 (2) (2008) 110–122.
[50] Z. Zhao, D. Yan, W. Ng, Mining probabilistically frequent sequential patterns in large uncertain databases, IEEE Trans. Knowl. Data Eng. 26 (5) (2013) 1171–1184.