Expert Systems With Applications 214 (2023) 119156

Contents lists available at ScienceDirect

Expert Systems With Applications

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

ITUFP: A fast method for interactive mining of Top-*K* frequent patterns from uncertain data

Razieh Davashi

Faculty of Computer Engineering, Najafabad Branch, Islamic Azad University, Najafabad, Iran
Big Data Research Center, Najafabad Branch, Islamic Azad University, Najafabad, Iran

ARTICLE INFO

Keywords:
Data mining
Frequent pattern mining
Uncertain frequent pattern
Uncertain data
Interactive mining

ABSTRACT

Top-*K* Uncertain Frequent Pattern (UFP) mining is an interesting topic in data mining. The existing TUFP algorithm supports static mining of Top-*K* UFPs; however, in the real world, users need to repeatedly change the *K* threshold to extract the information according to the requirements of their application. In interactive environments, the TUFP algorithm needs to re-scan the database and create the UP-Lists and CUP-Lists from scratch which is very time-consuming. In this paper, a fast method called ITUFP is proposed for interactive mining of Top-*K* UFPs. The proposed method uses a new data structure called IMCUP-List to store information of patterns efficiently. It creates the UP-Lists with a single database scan, extracts the patterns by generating IMCUP-Lists, and stores all the lists. When *K* changes, the proposed algorithm only updates the IMCUP-Lists without having to create the lists from scratch. Accordingly, ITUFP conforms to the "build once, mine many" principle, where the UP-Lists and IMCUP-Lists are created only once and used in mining with different *K* values. This is the first study on interactive mining of Top-*K* UFPs. Extensive experimental results with sparse and dense uncertain data prove that the proposed method is very efficient for interactive mining of Top-*K* UFPs.

1. Introduction

Frequent pattern mining is a key issue in the data mining field. Apriori (Agrawal & Srikant, 1994) and FP-growth (Han, Pei, & Yin, 2000) can be considered as fundamental methods in this field. Apriori extracts the frequent patterns using candidate generation and repeated scans of the database. To solve these problems, the FP-growth was proposed. These two algorithms and many of the algorithms introduced to extract a variety of frequent patterns support precise data. In recent years, with the technological advances in data collection methods, uncertain data has increased (Davashi, 2021a).

In general, methods for mining UFPs fall into five main categories: Apriori-based, H-mine-based, Eclat-based, FP-growth-based, and list-based methods (Davashi, 2021a). Each of these categories can be divided into two subcategories: upper bound-based and exact-based methods (Davashi, 2021a). The upper bound-based methods may produce false positives that are filtered out by an additional scan. The U-Apriori algorithm (Chui, Kao, & Hung, 2007) is the first method proposed for mining UFPs. It is an upper bound-based algorithm that suffers from the common drawbacks of Apriori-based algorithms; repeated database scans and candidate generation. Therefore, an exact-based algorithm called UF-growth (Leung, Carmichael, & Hao, 2007; Leung, Mateo, & Brajczuk, 2008) was proposed, which has a large size. Therefore, it requires a lot of time to extract patterns and a lot of memory to hold its tree nodes. To address the drawbacks of these algorithms, many exact-based algorithms such as CUFP-Mine (Lin & Hong, 2012), AT-mine (Le Wang, Feng, & Wu, 2013), and LUNA (Lee & Yun, 2017) were proposed, which extract UFPs by two database scans. In addition, numerous upper bound-based algorithms such as the UFP-growth (Aggarwal, Li, Wang, & Wang, 2009), CUF-growth (Leung & Tanbeer, 2012), CUF-growth* (Leung & Tanbeer, 2012), Disc-growth (MacKinnon, Strauss, & Leung, 2014), BLIMP-growth (Leung & MacKinnon, 2014), TPC-growth (Leung, MacKinnon, & Tanbeer, 2014), MUF-growth (Leung & MacKinnon, 2015), and UP-Mine (Davashi, 2021b) were proposed to extract UFPs through three database scans. Although these algorithms are designed for extracting uncertain patterns, they are not suitable for Top-*K* UFP mining and must find all patterns.

The problem of Top-*K* UFP mining is to find the *K* UFPs with the

Peer review under responsibility of Submissions with the production note 'Please add the Reproducibility Badge for this item' the Badge and the following footnote to be added:The code (and data) in this article has been certified as Reproducible by the CodeOcean: https://codeocean.com. More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physicalsciencesandengineering/computerscience/journals.

E-mail address: davashi@sco.iaun.ac.ir.

https://doi.org/10.1016/j.eswa.2022.119156
Received 21 February 2022; Received in revised form 1 September 2022; Accepted 25 October 2022
Available online 30 October 2022

highest expected support. The importance of Top-*K* mining methods is that intelligent systems can extract only Top-*K* patterns without generating all patterns. Therefore, the TUFP algorithm (Le, Vo, Huynh, Nguyen, & Baik, 2020) was proposed for mining Top-*K* UFPs. The TUFP algorithm extracts Top-*K* UFPs by using two strategies, raising the threshold and pruning strategy by raising the threshold. The TUFP algorithm supports static mining of Top-*K* UFPs. However, in the real world, users often change the *K* threshold to find new and suitable patterns according to their application needs. Indeed, the interactive mining of Top-*K* UFPs is more functional than the static mining.

The TUFP algorithm is not suitable for interactive environments where the user constantly changes the *K* threshold. This is because every time the user changes the *K* threshold, the TUFP algorithm has to re-scan the database and perform the entire mining process, including the creation of UP-Lists and CUP-Lists, from scratch. Therefore, the TUFP algorithm in interactive environments leads to an increase in disk I/O cost and a lack of reasonable flexibility. On the other hand, many algorithms have already been proposed for interactive mining of variations of frequent patterns from precise data, but these algorithms have drawbacks that reduce their efficiency. For example, these algorithms store database information efficiently in their data structure so that they do not have to rescan the database when the *minSup* value changes. However, they have to repeat the pattern mining process from these data structures, which is very time-consuming. In other words, these algorithms cannot store the data structures created during the mining process so that this data can be used in the next steps. This leads to performance degradation of algorithms in interactive environments. Motivated by the above problems, a fast method for interactive mining of Top-*K* UFPs is proposed in this paper. The main contributions are as follows:

- A single-pass method ITUFP (**I**nteractive **T**op-*K* **U**ncertain **F**requent **P**attern mining algorithm) is proposed for interactive mining of Top-*K* UFPs.
- A new data structure called IMCUP-List is proposed to efficiently store information about patterns so that the ITUFP algorithm can extract patterns in interactive environments only by updating the IMCUP-Lists. Techniques are also proposed to avoid additional computation when processing IMCUP-Lists.
- The proposed method conforms to the "build once, mine many" principle, which means that repeated mining with different *K* values is possible by using previous mining results. Indeed, this is the first method proposed for interactive mining that does not require to repeat the mining process from scratch.
- Extensive evaluations showed that the proposed ITUFP algorithm generates significantly fewer new lists than the TUFP algorithm in an interactive environment. Therefore, the ITUFP algorithm dramatically improves the runtime compared to the TUFP algorithm for all datasets, whether sparse/dense or large/small.

The remainder of this paper is organized as follows: Section 2 reviews related works and highlights the main differences of the current work; Section 3 describes the problem and the proposed method; Section 4 analyzes the time complexity of the proposed method; Section 5 presents and discusses the performance evaluation; and Section 6 contains the conclusions.

2. Related works

This section not only discusses related works, but also highlights the differences between the proposed method and previous works.

2.1. Interactive mining of frequent patterns

Finding an optimal value for the *minSup* threshold has always been a challenge for users. A *minSup* value that is too low will result in many patterns being extracted. A *minSup* value that is too high will result in only a few patterns being extracted and some important patterns being missed. Interactive mining provides the user with the ability to interactively refine and adjust the *minSup* value. In interactive mining, the database is fixed, but the *minSup* value can be continuously changed by the user. Importantly, the algorithms developed for interactive mining follow the "build once, mine many" principle. This principle allows users to perform repeated mining with different *minSup* thresholds using previous mining results or the same data structure (Ahmed, Tanbeer, Jeong, Lee, & Choi, 2012).

CanTree (Leung, Khan, Li, & Hoque, 2007), CP-tree (Tanbeer, Ahmed, Jeong, & Lee, 2009), and EFP-tree (Davashi & Nadimi-Shahraki, 2019) have interactive mining capability. They keep all the frequent and infrequent patterns in their tree structure. When the user changes the *minSup* value, these algorithms can extract the patterns without having to rescan the database and rebuild the tree from scratch.

The Two algorithms $IWFP_{WA}$ and $IWFP_{FD}$ (Ahmed, Tanbeer, Jeong, Lee, et al., 2012) have also been proposed for incremental and interactive mining of weighted patterns. These two algorithms also do not prune the infrequent items from their tree structure. Therefore, they do not need to rebuild their trees when the *minSup* value changes.

HUPMS (Ahmed, Tanbeer, Jeong, & Choi, 2012) is an algorithm for the interactive mining of high utility patterns over data streams. It captures information from a data stream in a batch-by-batch fashion inside the HUS-tree nodes. When the *minSup* value changes, the HUPMS algorithm can mine all high utility patterns in the current window without rebuilding the HUS-tree.

The LINE algorithm (Lee & Yun, 2018) is an algorithm for erasable pattern mining in interactive environments. It creates EI-lists for all infrequent and frequent items with a single database scan. It then combines EI-lists or EP-lists to create EP-lists for longer patterns. Whenever the user changes the *minSup* value, the LINE algorithm extracts the erasable patterns without having to rescan the database and create EI-lists from scratch.

As mentioned earlier, all of the above algorithms can extract patterns in interactive environments without having to rescan the database and rebuild their data structures. However, these algorithms have a major drawback. They have to repeat the mining process from scratch every time *minSup* changes. For example, the LINE algorithm stores the information about transactions in EI-lists using a database scan. When the *minSup* value changes, the algorithm does not need to rebuild the EI-lists from scratch. However, it must rebuild all EP-lists from scratch to find patterns with new *minSup* value, which is very time-consuming.

Unlike the previous algorithms, the ITUFP algorithm proposed in this paper efficiently stores the information of the mining process. Thus, if the *K* value changes, it can use this information in the next steps without repeating the mining process.

2.2. Mining Top-K frequent patterns

The importance of Top-*K* mining methods is that intelligent systems can extract only Top-*K* patterns without generating all patterns. To date, many methods have been proposed to extract a variety of Top-*K* patterns, which are summarized below. The first method for extracting Top-*K* patterns was presented in (Shen, Hong, & Prithard, 1998), which extends the Apriori approach to find the *k*-most interesting frequent patterns. This algorithm requires frequent database scans, which is very time-consuming. Two other Apriori-based algorithms called Itemset-Loop and Itemset-iLoop (Fu, Kwong, & Tang, 2000) were proposed to improve it. However, all of these algorithms have the same disadvantage as the Apriori approach.

The FP-tree-based TFP algorithm (Han, Wang, Lu, & Tzvetkov, 2002) extracts Top-*K* closed patterns whose lengths are not shorter than the given minimum length. After that, the TF^2P-growth algorithm (Hirate, Iwahashi, & Yamana, 2004) was proposed for Top-*K* pattern mining without support threshold. Other FP-tree-based algorithms such as

BOMO (Cheung & Fu, 2004) and ExMiner (Quang, Oyanagi, & Yamazaki, 2006) extract the Top-*K* patterns using an internal support threshold.

Algorithms that use an internal threshold usually follow the same general process. They first set an internal threshold to zero. As the mining process continues, the internal threshold is automatically increased using various strategies to reduce the search space.

The CRMN algorithm (Pyun & Yun, 2014) extracts Top-*K* frequent patterns by reducing the number of combinations of patterns in a single prefix tree path. The ETARM algorithm (Nguyen, Vo, Nguyen, Fournier-Viger, & Selamat, 2018) extracts the Top-*K* patterns using two pruning properties. The FTARM algorithm (Liu, Niu, & Fournier-Viger, 2021) improves on the ETARM algorithm by using a new technique called RGPP that reduces the search space. This technique uses the candidate pruning property to speed up the mining process.

In addition, numerous algorithms have been proposed to extract Top-*K* sequential patterns (Dong, Qiu, Lü, Cao, & Xu, 2019; Zhang, Du, Gan, & Philip, 2021), Top-*K* high utility patterns (Han, Liu, Li, & Gao, 2021; Krishnamoorthy, 2019), Top-*K* closed patterns (Pham, Do, Nguyen, Vo, & Hong, 2020; Wang et al., 2020), and Top-*K* patterns over data streams (Song, Liu, Ge, & Ge, 2019).

2.3. Mining uncertain frequent patterns

The U-Apriori algorithm (Chui et al., 2007) is an Apriori-based method that extracts uncertain patterns through candidate generation and repeated database scans. UCP-Apriori (Chui & Kao, 2008), MBP (Wang, Cheung, Cheng, Lee, & Yang, 2011), and IMBP (Sun, Lim, & Wang, 2012) are other Apriori-based algorithms that have the problems of the candidate generate-and-test approaches. UH-Mine (Aggarwal et al., 2009) and P-HMine (Bhadoria, Kumar, & Dixit, 2011) are H-mine-based algorithms that extract the patterns using a hyperlinked structure. U-Eclat (Calders, Garboni, & Goethals, 2010), UEclat (Abd-Elmegid, El-Sharkawi, El-Fangary, & Helmy, 2010), UV-Eclat (Leung & Sun, 2011), and U-VIPER (Leung, Tanbeer, Budhia, & Zacharias, 2012) are Eclat-based algorithms that extract the patterns using a vertical data structure.

UF-growth (Leung, Carmichael et al., 2007; Leung et al., 2008), CUFP-Mine (Lin & Hong, 2012), and AT-mine (Le Wang et al., 2013) are FP-growth-based methods. These exact-based algorithms require two database scans to construct their trees and mine uncertain patterns. The UF-growth algorithm as the first tree-based algorithm, can have a tree with large size. This is because it only shares the nodes from the UF-tree paths that have the same existential probability value and item name. Therefore, the UF-growth algorithm may require a long runtime to process the nodes and too much memory to store them. CUFP-Mine calculates the expected support for all supersets of its tree nodes and stores them in the tree structure. Therefore, extracting patterns with this algorithm can require a lot of memory and time. The AT-mine algorithm was proposed to solve these problems. This algorithm uses an array called ProArr to store the probability values of the tree paths. This array requires a lot of memory for large databases.

The UFP-growth (Aggarwal et al., 2009), CUF-growth (Leung & Tanbeer, 2012), CUF-growth* (Leung & Tanbeer, 2012), Disc-growth (MacKinnon et al., 2014), BLIMP-growth (Leung & MacKinnon, 2014), TPC-growth (Leung et al., 2014), MUF-growth (Leung & MacKinnon, 2015), and UP-Mine (Davashi, 2021b) are FP-growth-based methods. These upper bound-based algorithms require two database scans to build their trees and one scan to filter out false positives. Therefore, they need three database scans to mine uncertain patterns.

To build a more compact tree than the UF-tree, the UFP-growth algorithm was proposed. However, the UFP-tree depends on the clustering parameter, so UFP-growth can produce a UFP-tree that is as large as a UF-tree. Therefore, the CUF-growth and CUF-growth* algorithms (Leung & Tanbeer, 2012) were proposed to generate more compact trees than the UF-tree and UFP-tree. However, these algorithms produce many false positives. To reduce false positives, the algorithms PUF-growth (Leung & Tanbeer, 2013), Disc-growth (MacKinnon et al., 2014), BLIMP-growth (Leung & MacKinnon, 2014), TPC-growth (Leung et al., 2014), MUF-growth (Leung & MacKinnon, 2015), and UP-Mine (Davashi, 2021b) were proposed. However, they still generate a lot of false positives.

To solve the problems of the CUFP-Mine and AT-mine algorithms, the LUNA algorithm (Lee & Yun, 2017) was proposed. The LUNA algorithm is a list-based algorithm that generates UP-Lists to store information of transactions. It then extracts longer patterns by generating CUP-Lists. While all of the above methods are more or less effective at extracting uncertain patterns, none of them can efficiently extract the Top-*K* uncertain patterns. Therefore, a list-based algorithm, known as TUFP (Le et al., 2020), was proposed to mine Top-*K* UFPs. This algorithm improves the LUNA algorithm (Lee & Yun, 2017) and therefore uses the UP-List and CUP-List structures to extract Top-*K* UFPs. The TUFP algorithm deals with uncertain data; therefore, it extracts the *K* patterns with the highest expected support. It uses an internal threshold to extract Top-*K* UFPs. Consequently, the TUFP algorithm first sets an internal threshold of zero and uses two strategies to reduce the runtime: Raising the threshold and Pruning strategy by raising the threshold. Although the TUFP algorithm successfully extracts the Top-*K* UFPs, it does not work well in interactive environments. This is because every time the user changes *K*, this algorithm has to repeat the process of scanning the database and creating UP-Lists and CUP-Lists, which is very time-consuming.

To handle the interactive mining of Top-*K* UFPs, this paper proposes a fast method called ITUFP, which improves the TUFP algorithm.

3. Proposed method

In this section, the concepts and problem statement are explained. Then, the details of the proposed ITUFP method are described.

3.1. Problem statement and definitions

A given uncertain database with n transactions is represented as $UDB = \{T_1, T_2, \ldots, T_n\}$ and a set of distinct items in *UDB* is defined as $I = \{x_1, x_2, \ldots, x_d\}$. Each transaction T_j in *UDB* is a subset of *I*, and each item in each transaction has a unique existential probability value between (0, 1]. Table 1 shows a transactional uncertain database. $X = \{x_1, x_2, \ldots, x_k\}$ indicates a pattern or *k*-itemset composed of distinct items belonging to *I*.

Definition 1. *Each item x_i in each transaction T_j has a existential probability $P(x_i, T_j)$* $(0 < P(x_i, T_j) \leq 1)$.

For example, in Table 1:
$P(A, TID{:}100) = 0.7$, $P(C, TID{:}400) = 0.4$, and $P(B, TID{:}700) = 0.9$.

Definition 2. *The existential probability of pattern X in transaction T_j is denoted by $P(X, T_j)$ and defined by*:

$$P(X, T_j) = \Pi_{x \in X} P(x, T_j) \quad (1)$$

For example, in Table 1:

Table 1
Example of an uncertain database.

TID	Items			
	A	B	C	D
100	0.7	–	0.1	–
200	0.4	0.4	0.5	0.9
300	–	0.6	0.3	0.2
400	–	–	0.4	0.4
500	1	0.7	–	–
600	–	–	0.2	0.8
700	0.9	0.9	0.9	–

P ({A, C}, TID:100) = 0.7 × 0.1 = 0.07, *P ({A, B, C}, TID:700)* = 0.9 × 0.9 × 0.9 = 0.729.

Definition 3. *The expected support of pattern X in the UDB is denoted expSup(X) and is defined by:*

$$expSup(X) = \sum_{j=1}^{|UDB|} \left(\Pi_{x \in X} P(x, T_j) \right) = \sum_{j=1}^{|UDB|} \left(P(X, T_j) \right) \quad (2)$$

For example, in Table 1:

expSup(A, B) = *P({A, B}, TID:200)* + *P({A, B}, TID:500)* + *P({A, B}, TID:700)* = (0.4 × 0.4) + (1 × 0.7) + (0.9 × 0.9) = 1.67.

The *expSup(X)* satisfies the downward closure property (Agrawal & Srikant, 1994), which allows the search space to be reduced without pattern loss, since *expSup(X)* ≤ *expSup(Y)* for all *Y* ⊂ *X*. Therefore, Property 1 can be defined as follows:

Property 1. *For any Y* ⊂ *X, the expected support satisfies the downward closure property (Davashi, 2021a):*

a) If *expSup(X)* ≥ *minSup*, it is also true that *expSup(Y)* ≥ *minSup*, and
b) If *expSup(Y)* < *minSup*, it is also true that *expSup(X)* < *minSup*.

Definition 4. (*mining Top-K UFPs*) *Given a UDB and a K threshold, the Top-K UFP mining is to discover K patterns with the highest expected support.*

Problem statement. This paper focuses on interactive mining of Top-*K* UFPs where the database is fixed and the *K* threshold can be continuously changed by the user. Given an uncertain database *UDB* and several *K*, the research problem is to find a new set of Top-*K* UFPs based on the new *K* value, following the "build once, mine many" principle.

3.2. The overall architecture of the proposed method

The overall architecture of the proposed ITUFP method is illustrated in Fig. 1. The ITUFP algorithm constructs the UP-Lists with only one *UDB* scan. When the user requests for mining Top-*K* UFPs, the proposed ITUFP constructs IMCUP-Lists, recursively. When all patterns are extracted, the result set, R, is reported to the user and the UP-Lists and IMCUP-Lists constructed are stored for the next steps. If the user changes the *K* threshold and requests a new mining based on the new *K* value, the proposed ITUFP extracts the Top-*K* UFPs by updating the IMCUP-Lists. Then, the algorithm stores the updated IMCUP-Lists for the next steps. Whenever the *K* threshold changes, this process is repeated to extract patterns.

3.3. Proposed method

As mentioned earlier, the TUFP algorithm is not suitable for interactive environments. In interactive environments, an efficient method must have the following features: 1) All the work required to extract the Top-*K* UFPs must be performed by a single scan of the database (read-only once). 2) Whenever the *K* value changes, it should not be necessary to reconstruct the lists and repeat the mining process from scratch (build once, mine many).The proposed ITUFP can easily satisfy the above conditions by using a list structure.

A simple solution to handle an interactive environment would be to store only the UP-Lists for the next steps. Indeed, after the first mining is completed, the UP-Lists are stored. If the user changes the *K*, there is no need to re-scan the database and rebuild the UP-Lists and only the process of building the CUP-Lists is done from scratch. However, this method is not very effective. For this reason, a more effective method is proposed.

In this way, the proposed ITUFP applies an effective method for updating CUP-Lists that does not require repeating the mining process from scratch. It effectively stores the information of CUP-Lists so that this information can be easily updated for the next steps if needed. Therefore, the algorithm can use the knowledge acquired in the previous steps. The current CUP-List structure is not very efficient to update. Therefore, more features would need to be added to this structure. For this purpose, a new data structure called IMCUP-List is proposed.

Definition 5. (***I**nteractive **M**ining of **C**onditional **U**ncertain **P**robability-**List*** (*IMCUP-List*)). *The IMCUP-List is similar to a CUP-List except that it contains two additional parameters called Index1 and Index2. Since the*

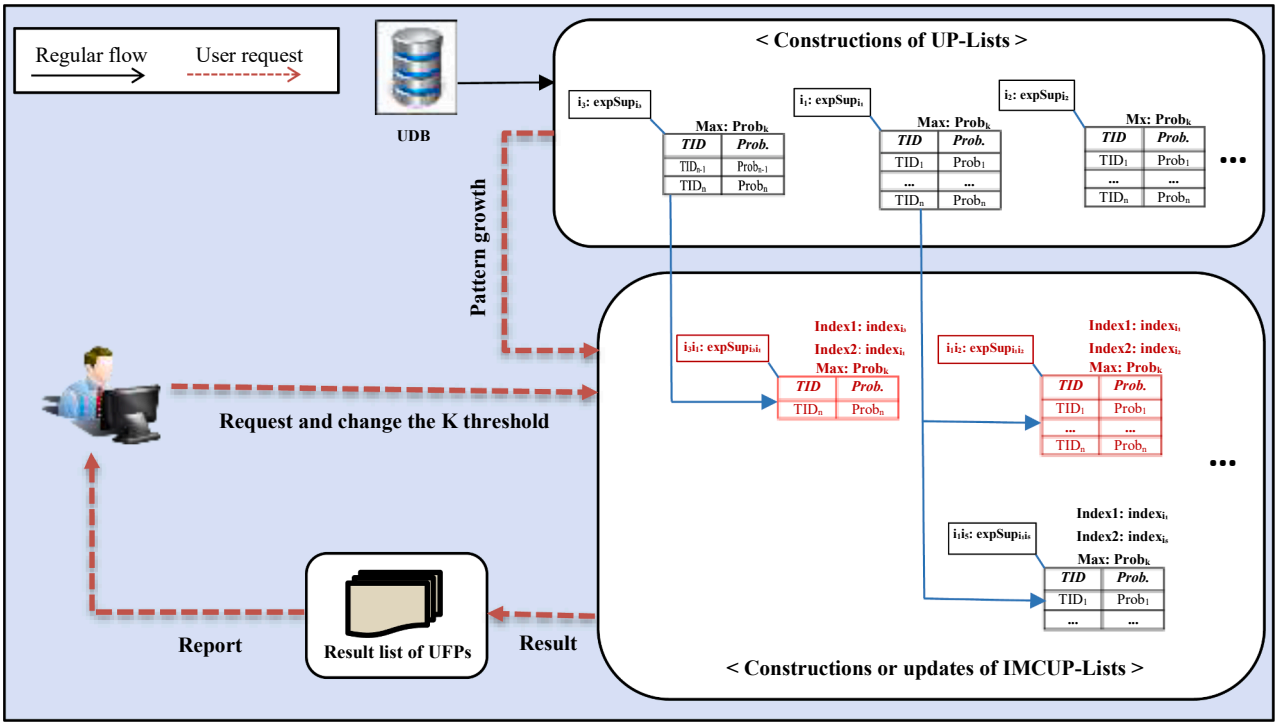


Fig. 1. Overall architecture of proposed ITUFP algorithm.

*algorithm constructs each IMCUP-List by combining two UP-Lists or IMCUP-Lists, Index*1 *illustrates the index of the last processed tuple of the first list and Index*2 *illustrates the index of the last processed tuple of the second list.*

In the LUNA and TUFP algorithms, there is no link between the UP-Lists/CUP-Lists and the CUP-Lists generated from them. These algorithms process the CUP-Lists immediately and do not use them for the next steps. In the proposed algorithm, the UP-Lists and IMCUP-Lists are used in the next steps. Therefore, the proposed algorithm considers a link between each UP-List/IMCUP-List and the IMCUP-Lists generated from it. This link facilitates the access to this information in the next steps. Fig. 2 shows the general architecture of UP-List and IMCUP-List.

3.3.1. Generating of UP-Lists

The ITUFP algorithm stores transactions data in the UP-Lists by scanning the database. It creates UP-Lists for all items and sorts them in descending order of *expSup*. After generating UP-Lists, the proposed algorithm extracts the Top-*K* UFPs in interactive environments by generating or updating IMCUP-Lists as will be described in the following section.

3.3.2. Interactive mining of Top-K UFPs by generating or updating IMCUP-Lists recursively

The proposed algorithm uses the manner of recursive divide-and-conquer to mine the patterns. It uses a Top-*K* array, T^k, to maintain the current Top-*K* UFPs and applies two strategies: 1) raising the threshold and 2) pruning strategy by raising the threshold, to increase the efficiency. In the strategy of raising the threshold, the proposed method relies on an internal *minSup* whose value is zero at the beginning but increases during the mining process. In the pruning strategy by raising the threshold, only lists whose *expSup* value is greater than the updated *minSup* value are combined.

It should be noted that the TUFP algorithm uses all the techniques introduced to improve the LUNA algorithm. Therefore, all these techniques are used by the proposed algorithm to improve the efficiency of the mining process. The techniques PUFP (Potential Uncertain Frequent Pattern) and *ppf* (Pre-Pruning Factor) are the most important of these techniques. When combining two lists, the proposed algorithm checks the PUFP technique. If this condition is satisfied, two lists are combined. The proposed algorithm also uses *ppf* technique for early detection of infrequent lists. In (Lee & Yun, 2017), it was found that it is more efficient to check the *ppf* condition for every-two tuples than to do so for every tuple. For more details about these techniques, please refer to (Lee & Yun, 2017).

According to the given explanations, the mining process of Top-*K* UFPs by the proposed algorithm in interactive environments is as follows.

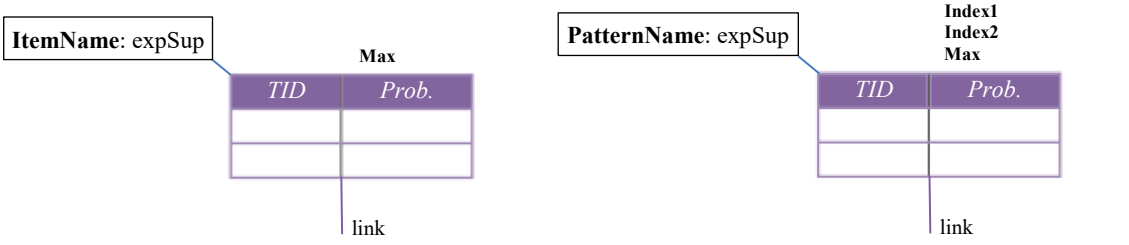
For the first mining process, the proposed algorithm considers a Top-*K* array, T^k, with a length *K* and an internal *minSup* that is initially set to zero. The algorithm stores the first *K* items of UP-Lists in T^k, sorts T^k in descending order of *expSup*. When T^k has *K* elements, *minSup* is updated to the *expSup* value of the last element in T^k. Then, the proposed algorithm starts processing the lists from the beginning of them and combines each frequent list with the next frequent lists to create new IMCUP-Lists. When combining the two lists, the algorithm first checks the PUFP condition and then combines the lists that satisfy this condition. When generating IMCUP-lists, the proposed algorithm uses the *ppf* technique for early detection of infrequent lists. An important difference between the proposed algorithm and the TUFP algorithm is that the proposed algorithm ceases processing the lists corresponding to these infrequent patterns, but does not prune them. Unlike the TUFP algorithm, the proposed algorithm stores all frequent and infrequent lists, since infrequent lists may be frequent in the next steps. Some tuples of these lists have been processed in this step; therefore, the proposed algorithm uses two indexes named Index1 and Index2 for the IMCUP-Lists. As mentioned in Definition 5, the proposed algorithm constructs each IMCUP-List by combining two UP-Lists or IMCUP-Lists. The Index1 and Index2 illustrate the indexes of the last processed tuples of the first and second lists, respectively. Thus, in the next steps, the proposed algorithm can process only the unprocessed tuples of these lists using these indexes. This intelligent approach increases the efficiency of the proposed algorithm and avoids reprocessing these tuples in the next steps. The algorithm constructs all lists in the same way and updates T^k and *minSup* whenever the *expSup* of an IMCUP-Lists is greater than the *minSup*. Once the mining process is complete, the algorithm considers the patterns in T^k as output and stores all lists for the next steps.

Thereafter, each time the user changes *K*, the algorithm extracts the Top-*K* patterns as follows.

The algorithm considers a new T^k with a length *K* and an internal *minSup* that is initially set to zero. All UP-Lists and some IMCUP-Lists were created in the previous steps; therefore, unlike the TUFP algorithm, the proposed algorithm does not need to re-scan the database and rebuild the lists. It suffices that the algorithm stores the first *K* items of the lists in T^k and updates the *minSup* if T^k contains *k* elements. It then processes the UP-Lists and combines each frequent UP-List with its subsequent frequent lists to construct or update the IMCUP-Lists while the PUFP and *ppf* conditions are regularly checked.

Indeed, if a new IMCUP-List is constructed, the proposed algorithm adds it at the beginning of the lists; otherwise, if the IMCUP-List is available from the previous steps, the proposed algorithm only updates it if needed. The proposed algorithm suggests an efficient method called PUT technique to update IMCUP-Lists. Moreover, the proposed algorithm can use this technique to quickly determine whether the existing IMCUP-List needs to be updated or not.

Definition 6. (***P**rocessing **U**nprocessed **T**uples* (***PUT** technique*)). *Let U_1 and U_2 be two Lists given for the construction of an IMCUP-List. If Index*1 *or Index*2 *of the U_1U_2's IMCUP-List or both are the same as the last tuples indexes of U_1 and U_2, respectively, the U_1U_2's IMCUP-List need not be updated; otherwise, the U_1U_2's IMCUP-List is updated. To update the U_1U_2's IMCUP-List, only the unprocessed tuples of U_1 and U_2 are processed using Index*1 *and Index*2 *of the U_1U_2's IMCUP-List. For this purpose, Index*1 *and Index*2 *of the U_1U_2's IMCUP-List are read by the algorithm and their subsequent tuples are processed.*



(a) General architecture of UP-List (b) General architecture of IMCUP-List

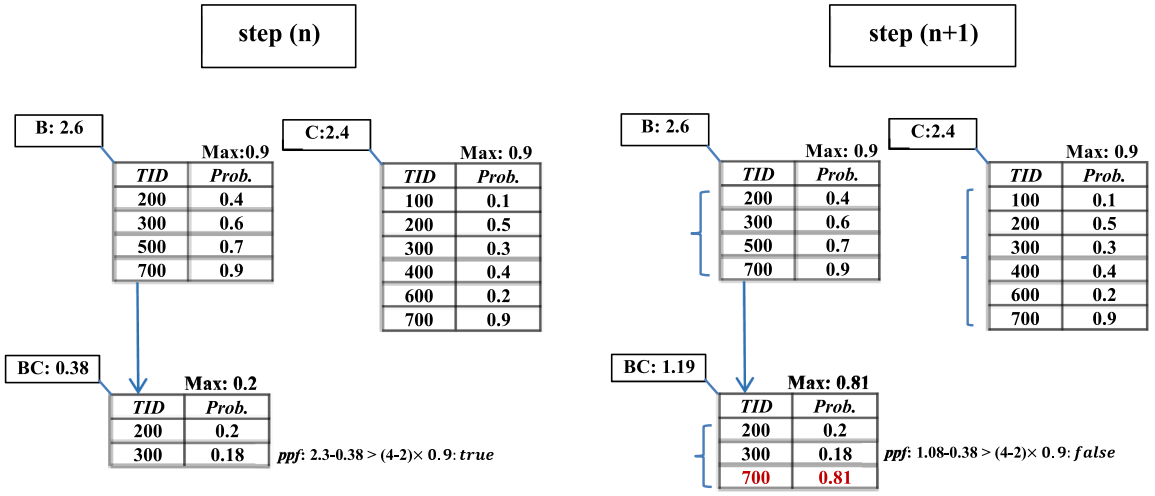
Fig. 2. The general architecture of UP-List and IMCUP-List.

Example 1 describes how the IMCUP-Lists are updated by the proposed algorithm.

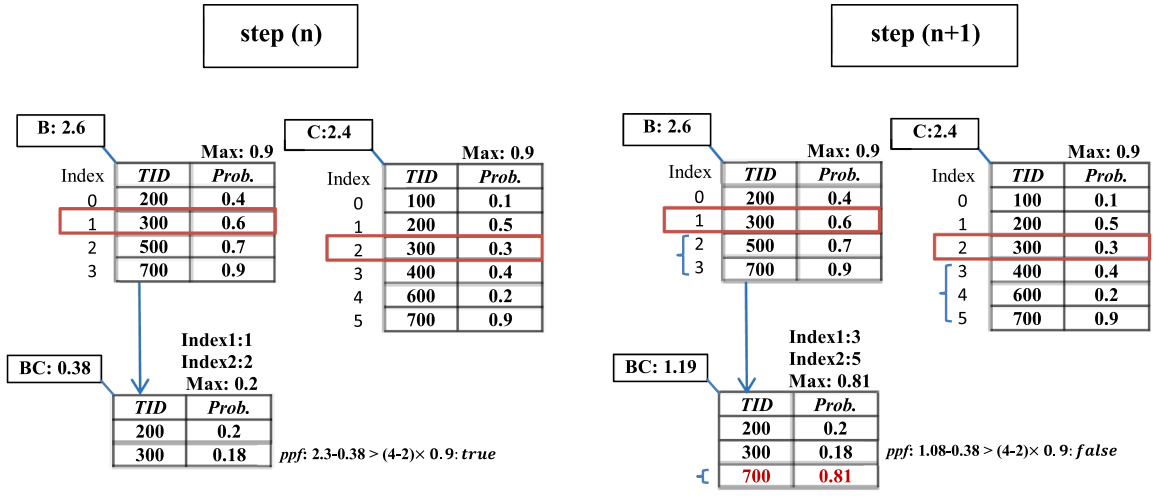
Example 1. *For step (n) of Fig. 3(a), consider K = 4 and internal minSup = 2.3. As shown in Fig. 3(a), the IMCUP-List for BC is constructed by combining the B's UP-List and C's UP-List. When generating the BC's IMCUP-List, the algorithm checks the ppf condition after processing the second tuple of B's UP-List (i.e., TID: 300). Since the ppf condition is satisfied, the algorithm ceases processing the remaining tuples and stores the infrequent BC's IMCUP-List. The user changes the K value to 6 and the internal minSup value becomes 1.08. As shown in step (n + 1) of Fig. 3(a), the proposed algorithm processes the B's UP-List and C's UP-List. Since the IMCUP-List for BC was created in the previous step. Now the algorithm needs to update it. A naive approach is to check all the tuples set of B's UP-List and C's UP-List from the beginning. According to this approach, every time the algorithm finds a common tuple, it must check all tuples of BC's IMCUP-List. If the BC's IMCUP-List does not contain this common tuple, the algorithm should add it to the BC's IMCUP-List. In step (n + 1) of Fig. 3(a), it can be seen that this approach is costly because the tuples processed in the previous steps need to be processed again.*

In contrast, the algorithm can update BC's IMCUP-List by processing only the unprocessed tuples of B's UP-List and C's UP-List. As explained earlier, Index1 and Index2 help in this task. Indeed, the algorithm can easily find unprocessed tuples through Index1 and Index2 to update BC's IMCUP-List. The Index1 = 1 and Index2 = 2 of BC's IMCUP-List in step (n) of Fig. 3(b) represent the indexes of the last processed tuples of B's UP-List and C's UP-List, respectively. To update the BC's IMCUP-List, as shown in step (n + 1) of Fig. 3(b), the algorithm first checks the *ppf* condition for TID:300 from the BC's IMCUP-List. Since the result is false, the algorithm processes only the subsequent tuples of TID: 300 from B's UP-List and C's UP-List. Indeed, Index1 = 1 and Index2 = 2 of the BC's IMCUP-List are read by the algorithm. Then, the algorithm finds Index1 = 1 on B's UP-List and Index2 = 2 on C's UP-List and processes tuples of B's UP-List whose index is greater than 1 (i.e., TID:500 and TID:700) and tuples of C's UP-List whose index is greater than 2 (i.e., TID:400, TID:600, and TID:700). The algorithm adds the new common tuple (i.e., TID: 700) to BC's IMCUP-List, and updates Index1 and Index2 of BC's IMCUP-List for the next steps (i.e., Index1 = 3 and Index2 = 5).

Moreover, Index1 and Index2 can help the proposed algorithm to



(a) Naïve approach for updating IMCUP-Lists



(b) Proposed approach for updating IMCUP-Lists

Fig. 3. Comparison of naïve and proposed methods for updating IMCUP-Lists.

identify lists that do not need to be updated. When all the tuples of an IMCUP-List are fully processed, this list does not need to be updated in the next steps. The proposed algorithm can quickly identify these lists. If Index1 or Index2 of BC's IMCUP-List or both are the same as the last tuples indexes of B's UP-List and C's UP-List, respectively, the proposed algorithm detects that all tuples set of B's UP-List and C's UP-List have been processed in the previous steps to create BC's IMCUP-List. Therefore, there is no need to process them and update BC's IMCUP-List. For example, consider the BC's IMCUP-List in step (n + 1) of Fig. 3(b). Since the Index1 = 3 and Index2 = 5 of BC's IMCUP-List are equal to the last tuples indexes of B's UP-List and C's UP-List, respectively, there is no need to update the BC's IMCUP-List in the next steps.

Consequently, Index1 and Index2, as described in Example 1, can significantly improve the performance of the algorithm.

Consider U_1 and U_2 as the current frequent lists that are merged to create or update IMCUP-Lists. When the user changes *K*, the algorithm may encounter the following three cases when updating the IMCUP-Lists.

Case 1. *There is no IMCUP-List for U_1U_2. Therefore, the proposed algorithm combines U_1 and U_2 to construct a new IMCUP-List for U_1U_2. Then, it adds the constructed U_1U_2's IMCUP-List to the beginning of the lists.*

Case 2. *The IMCUP-List for U_1U_2 has been created in the previous steps. The Index1 or Index2 of the U_1U_2's IMCUP-List or both are equal to the last tuples indexes of U_1 and U_2, respectively. In this case, the U_1U_2's IMCUP-List does not need to be updated (as explained in Example 1).*

Case 3. *The IMCUP-List for U_1U_2 has been created in the previous steps. However, Index1 and Index2 of the U_1U_2's IMCUP-List are not equal to the last tuples indexes of U_1 and U_2, respectively. In this case, the algorithm must update the U_1U_2's IMCUP-List. Therefore, based on the PUT technique, only the unprocessed tuples of U_1 and U_2 are processed. Then, the proposed algorithm adds new commen tuples to the existing tuples of the U_1U_2's IMCUP-List (as explained in Example 1).*

The following examples well illustrate the performance of the proposed algorithm in interactive environments.

Example 2. *Consider the UDB in Table 1 with K = 4. The proposed algorithm considers an array T^k of length 4, in which the current Top-K UFPs are stored, and an internal minSup that is initially set to zero. The proposed algorithm scans the database, constructs UP-Lists for all items in the UDB, and then sorts them in descending order of expSup. Therefore, items are ordered as A-B-C-D (Fig. 4). Next, the proposed algorithm inserts the first four items in UP-Lists into T^k. Since T^k has 4 elements and the expSup value of the last element is 2.3, the minSup value is updated to 2.3.*

When processing the frequent item A, the algorithm checks whether item A can be combined with the subsequent items B, C, and D. Since B, C, and D are frequent, the A's UP-List can be combined with them to create the AB's IMCUP-List, AC's IMCUP-List, and AD's IMCUP-List. For all lists, the proposed algorithm first checks the PUFP condition. If this condition is satisfied, the lists are combined. When creating AB's IMCUP-List, the proposed algorithm checks the condition of *ppf* after processing the second tuple of A's UP-List (i.e., TID: 200). Since the condition is satisfied, the algorithm ceases processing the remaining tuples and stores the infrequent AB's IMCUP-List with Index1 = 1 and Index2 = 0. Similarly, the proposed algorithm processes the infrequent AC's IMCUP-List and stores it with Index1 = 1 and Index2 = 1. To create the AD's IMCUP-List, the algorithm processes all the tuples of the A's UP-List and D's UP-List and stores the AD's IMCUP-List. Since its *expSup* value is less than the *minSup* value, T^k and *minSup* are not updated. The

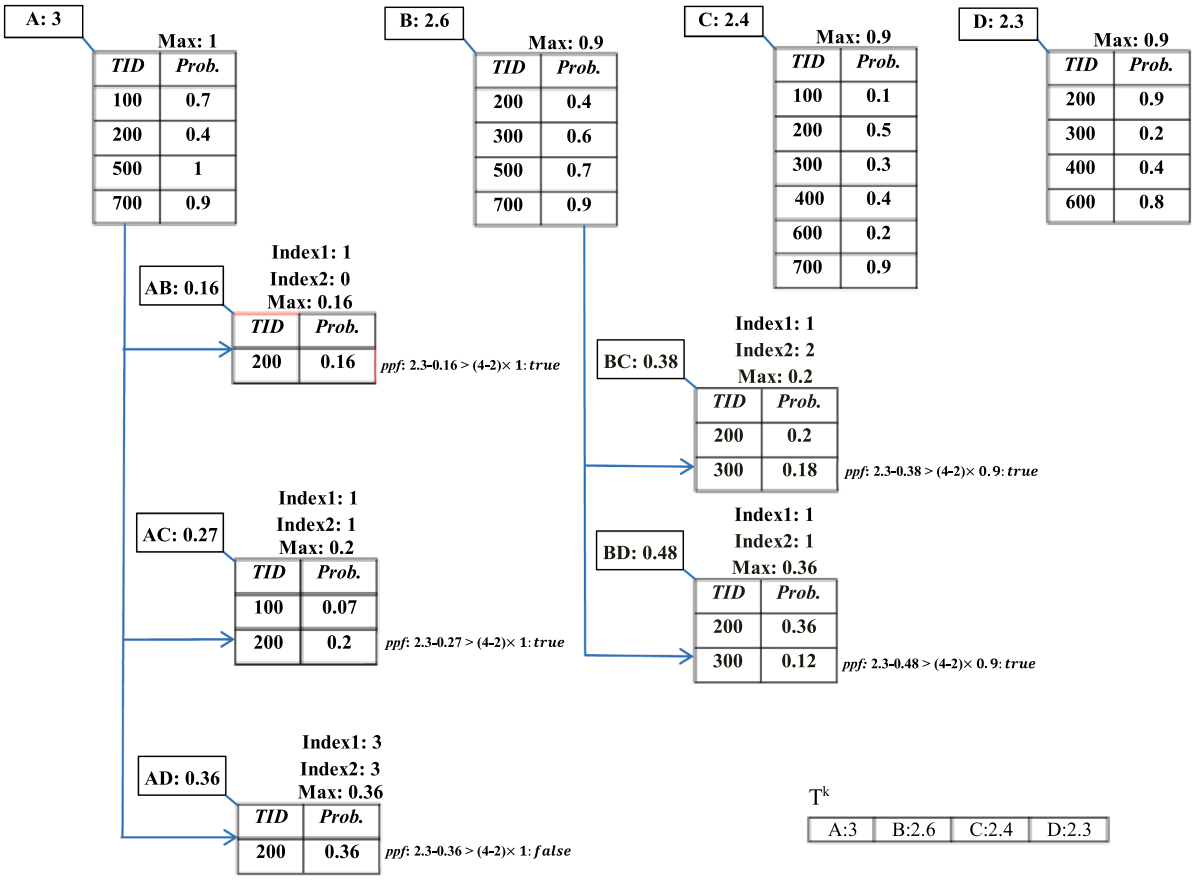


Fig. 4. The Interactive mining of Top-*K* UFPs when *K* = 4.

algorithm cannot combine the patterns AB, AC, and AD to create IMCUP-lists for longer patterns because they are infrequent. Therefore, the algorithm continues its work by processing the B's UP-List.

To construct the BC's IMCUP-List, the algorithm checks the condition of *ppf* after processing the second tuple of the B's UP-List (i.e., TID: 300). Since the condition is satisfied, the algorithm finishes processing the remaining tuples and stores the infrequent BC's IMCUP-List with Index1 = 1 and Index2 = 2. Similarly, the algorithm processes the infrequent BD's IMCUP-List and stores it with Index1 = 1 and Index2 = 1. The next item to be processed is C. Although the items C and D are frequent, after checking the PUFP condition, the algorithm does not generate the CD's IMCUP-List. This is because the overestimated *expSup* value for the CD's IMCUP-List is 2.16 (=*expSup*(C) * Max(D) = 2.4 * 0.9), while the *minSup* value is 2.3. Finally, the algorithm processes D, but after that there is no more list. Therefore, the algorithm terminates and the extracted patterns are {A:3}, {B:2.6}, {C:2.4}, and {D:2.3}.

Example 3. *Consider the user changes the value K to* 6. *Therefore, the proposed algorithm considers a new array* T^k *of length* 6 *and an internal minSup that is initially set to zero. Since the UP-Lists were generated in the previous step, there is no need to scan the database and generate UP-Lists from scratch* (Fig. 5). *The algorithm inserts the first four items in UP-Lists into* T^k. *Since* T^k *does not yet have* 6 *elements, minSup is* 0.

By processing the A's UP-List, the algorithm recognizes that the AB's IMCUP-List has been generated in the previous step. Therefore, its Index1 and Index2 are checked by the algorithm. Index1 and Index2 of the AB's IMCUP-List are not equal to the last tuples indexes of A's UP-List and B's UP-List, respectively. Therefore, the algorithm must update AB's IMCUP-List. To update, the algorithm reads the Index1 = 1 and Index2 = 0 of the AB's IMCUP-List based on the PUT technique. It compares the subsequent tuples of index = 1 in A's UP-List (i.e., TID: 500 and TID: 700) and index = 0 in B's UP-List (i.e., TID: 300, TID: 500, and TID: 700). The result is two new tuples (i.e., TID: 500 and TID: 700), which the algorithm adds to the AB's IMCUP-List, and updates Index1 and Index2 of the AB's IMCUP-List for the next steps (i.e., Index1 = 3 and Index2 = 3). Then, the algorithm adds the pattern AB to the T^k. Similarly, it updates the AC's IMCUP-List and adds the AC pattern to T^k. Since the number of T^k elements is six, the algorithm sorts the T^k and updates the *minSup* value to 1.08 (Fig. 5). The algorithm processes the AD's IMCUP-List and quickly recognizes that the AD's IMCUP-List does not need to be updated because its Index1 = 3 and Index2 = 3 are equal to the last tuples indexes of A's UP-List and D's UP-List, respectively. The patterns AB and AC are frequent at this step; therefore, the algorithm recursively generates an IMCUP-List for ABC, by processing the AB's IMCUP-List. The algorithm checks the *ppf* condition after processing the second tuple of the AB's IMCUP-List (i.e., TID: 500). Since the condition is satisfied, the algorithm ceases processing the remaining tuples and stores the infrequent ABC's IMCUP-List with Index1 = 1 and Index2 = 1.

Thereafter, the algorithm processes the B's UP-List and updates the BC's IMCUP-List and BD's IMCUP-List. The *expSup* value of the BC

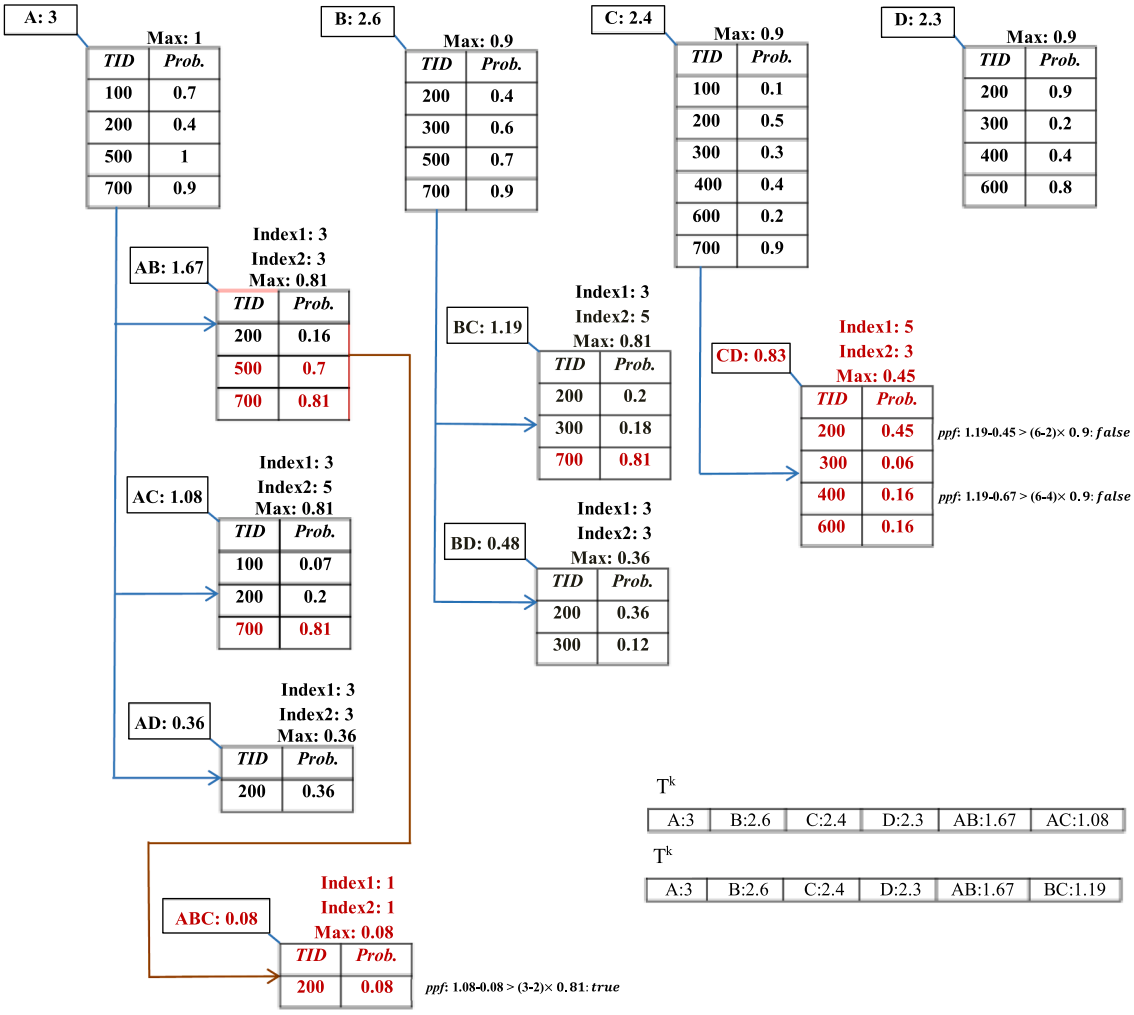


Fig. 5. The Interactive mining of Top-*K* UFPs when *K* = 6.

pattern is greater than the *minSup* value; therefore, AC is removed from the T^k, and the BC is added. The algorithm sorts the T^k and updates the *minSup* value to 1.19 as shown in Fig. 5. The algorithm cannot combine the BC and BD patterns to create the BCD's IMCUP-lists because the BD pattern is infrequent. Therefore, the algorithm continues its work by processing the C's UP-List.

The overestimated *expSup* value for the CD's IMCUP-List is 2.16 by checking the PUFP condition while the *minSup* value in this step is 1.19; therefore, the algorithm generates the CD's IMCUP-List, as shown in Fig. 5. The *expSup* value of the CD's IMCUP-List is less than the *minSup* value, so T^k is not updated. Therefore, the extracted patterns in this step are {A:3}, {B:2.6}, {C:2.4}, {D:2.3}, {AB:1.67}, and {BC:1.19}.

Example 4. *Consider the user changes K to* 8. *The proposed algorithm considers a new array* T^k *of length* 8 *and an internal minSup that is initially set to zero. It inserts the first four items in UP-Lists into* T^k. *Since* T^k *does not yet have* 8 *elements, minSup is* 0.

The A's UP-List is processed, and since the IMCUP-Lists for AB, AC and AD do not need to be updated, the algorithm only adds them to T^k. Then the algorithm recursively checks the ABC's IMCUP-List by processing the AB's IMCUP-List. The ABC's IMCUP-List is updated and added to T^k. Since the T^k has eight elements, the algorithm sorts the T^k and updates the *minSup* to 0.36 (Fig. 6). The AD's IMCUP-List is frequent in this step; therefore, the algorithm generates the IMCUP-Lists for ABD and ACD as shown in Fig. 6. Since their *expSups* are less than the *minSup*, the T^k is not updated.

Next, the algorithm processes the B's UP-List, BC's IMCUP-List, and BD's IMCUP-List. The BC's IMCUP-List and BD's IMCUP-List do not need to be updated. The *expSup* value of the BC's IMCUP-List is greater than the *minSup* value; therefore, AD is removed from the T^k and the BC is added. Therefore, the T^k is sorted and the *minSup* is updated to 0.809, as shown in Fig. 6.

Finally, the algorithm processes the C's UP-List and CD's IMCUP-List. The *expSup* value of the CD's IMCUP-List is greater than the *minSup*

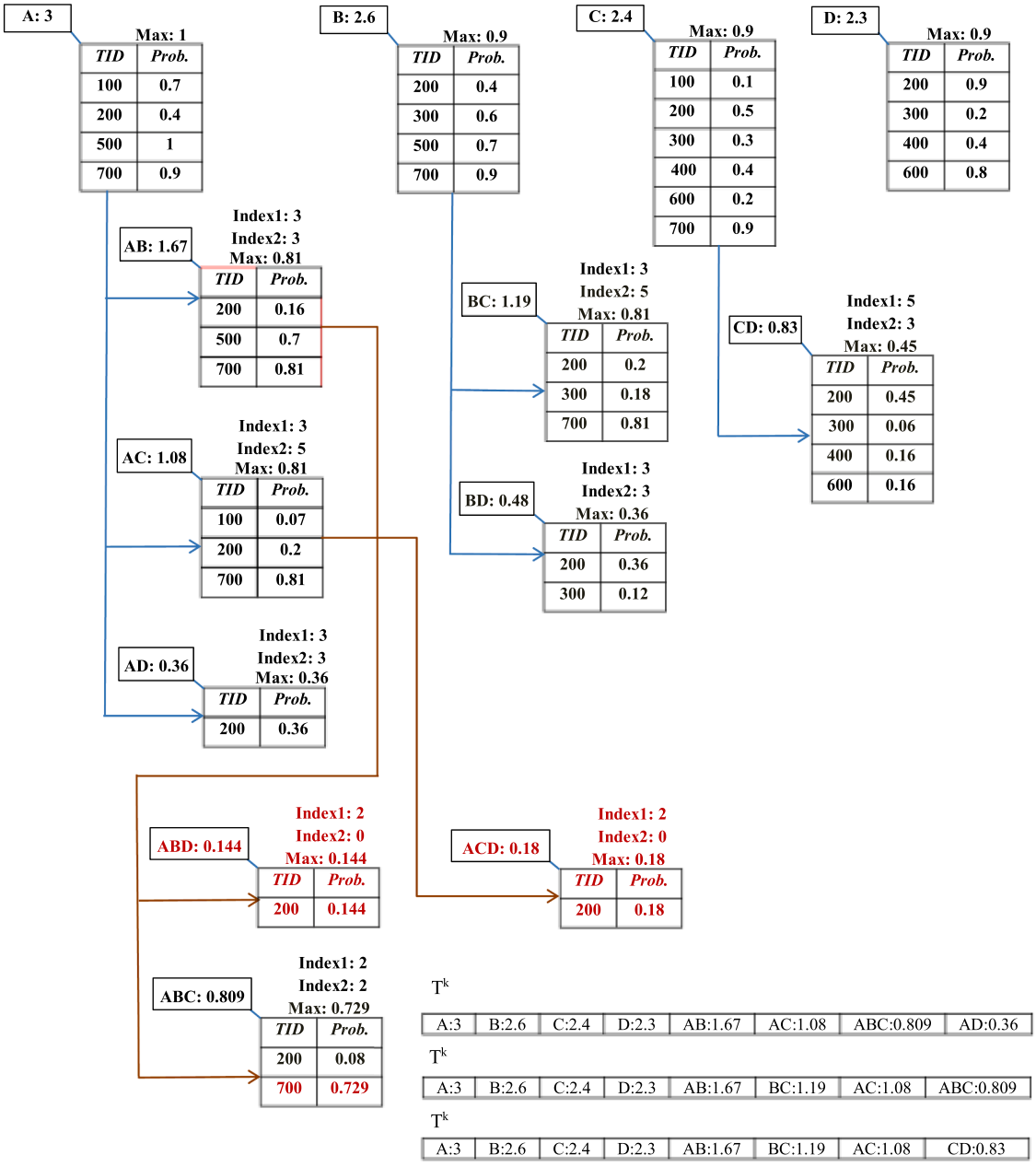


Fig. 6. The Interactive mining of Top-*K* UFPs when $K = 8$.

value; therefore, ABC is removed from the T^k, CD is added, and the *minSup* value is updated to 0.83, as shown in Fig. 6. Finally, the algorithm succeeds in extracting the patterns {A:3}, {B:2.6}, {C:2.4}, {D:2.3}, {AB:1.67}, {BC:1.19}, {AC:1.08}, and {CD:0.83}.

3.3.3. ITUFP algorithm

Fig. 7 presents the overall pseudo-code of the ITUFP method. The UP-Lists for all items are created by scanning the UDB (Line 1). Then the algorithm sorts the UP-Lists in their *expSup* descending order (Line 2). The user makes a mining request with a *K* value (Line 3), the algorithm defines an array of length *K*, T^k, and a *minSup* with initial value zero (Line 4). It inserts the first *K* items in UP-Lists into T^k (Line 5). When T^k has *K* elements, the algorithm updates *minSup* with the *expSup* value of the last element of T^k (Lines 6 to 8). The sub-procedure Mine_Patterns is called for mining the Top-*K* UFPs and storing them in T^k (Line 9). Each time a mining request is made by the user with a new *K*, the part between lines 3 to 10 is called for mining Top-*K* UFPs. The proposed algorithm consists of sub-procedures, which are shown in Fig. 8.

The first sub-procedure, Mine_patterns, describes how to create or update IMCUP-Lists from UP-Lists (Lines 1 to 32). For each UP-List, u_x (Line 1), if the *expSup* of u_x is greater than the *minSup* (Line 2), the combination of u_x with each subsequent UP-List, u_l, is checked by the algorithm. If u_x has no set of IMCUP-Lists, a set of IMCUP-Lists, IMC, and a prefix itemset, pref, are initialized for u_x (Line 3), and the item of u_x is set to pref (Line 4). Then, for each UP-List, u_l, located after u_x (Line 5), the algorithm checks whether the u_l has an *expSup* value greater than the *minSup* value (Line 6). The algorithm then proceeds with subsequent tasks to create or update an IMCUP-List from u_x and u_l. Accordingly, the PUFP condition is checked first (Line 7). If u_xu_l is a PUFP and no IMCUP-List for u_xu_l was generated in the previous steps (Line 8), an IMCUP-List is generated from u_x and u_l, c, while the *ppf* condition is regularly checked by the algorithm (Line 9). Next, the Index1 value of c is set to the index value of the u_x's last processed tuple and Index2 value of c is set to the index value of the u_l's last processed tuple (Line 10), and c is added into a set of IMCUP-Lists for the next steps of mining (Line 11). Otherwise (Line 12), if the IMCUP-List for u_xu_l, c, already exists and Index1 and Index2 of c are not equal to the indexes of the u_x's last tuple and the u_l's last tuple, respectively (Line 13), the existing u_xu_l's IMCUP-List is updated according to the technique PUT, while the *ppf* condition is regularly checked by the algorithm (Line 14).

Thereafter, the Index1 and Index2 values of c are updated for the next steps (line 16). Now, if *expSup* of c is greater than *minSup* (Line 18), the algorithm removes the last element of T^k and inserts the pattern of c into T^k, and sorts the T^k (Line 19). If T^k has *K* elements, the algorithm updates the *minSup* value with the *expSup* value of the last element of T^k (Lines 21 to 23). After the algorithm finishes processing u_x, the sub-procedure ITUFP_growth is called for mining longer patterns when the number of IMCUP-Lists generated from u_x is more than 1 (Lines 27 to 29).

The second sub-procedure, ITUFP_growth, describes how to mine Top-*K* UFPs with 3 or more lengths (Lines 33 to 63). It works in the same way as the part between lines 1 to 32 in the sub-procedure Mine_Patterns. The only difference is that the prefix information is constantly updated each time the sub-procedure is called (Line 36). The IMCUP-Lists are recursively generated or updated from previous IMCUP-Lists until there is no more list to combine.

4. Analysis of time complexity

This section compares the time complexities of the proposed ITUFP algorithm and the TUFP algorithm (Le et al., 2020) in an interactive environment where the user is constantly changing *K*. To evaluate the time complexity, the symbol definitions are given in Table 2.

The main processes for mining patterns by the TUFP and ITUFP algorithms are as follows (i) scanning the UDB, (ii) constructing UP-Lists, and (iii) constructing CUP-Lists/IMCUP-Lists. In an interactive environment, when the user changes *K*, the TUFP algorithm needs to scan the UDB and create the UP-Lists and CUP-Lists from scratch. Therefore, the time required for interactive mining of Top-*K* UFPs by the TUFP algorithm is as follows.

T_{TUFP} = Time cost for scanning the UDB + Time cost for generating UP-Lists from scratch + Time cost for generating CUP-Lists from scratch.

$$T_{TUFP} = T_{UDB} + T_{up} + T_{cup} \tag{3}$$

The ITUFP algorithm constructs the UP-Lists by scanning the UDB, extracts the Top-*K* UFPs by constructing IMCUP-Lists, and finally stores the lists for the next steps. When *K* changes, the proposed algorithm does not need to re-scan the UDB and rebuild the UP-Lists. It only updates the IMCUP-Lists. Therefore, the time required for interactive mining of Top-*K* UFPs by the ITUFP algorithm is as follows.

T_{ITUFP} = Time cost for updating IMCUP-Lists.

$$T_{ITUFP} = T_{u-IMCUP} \tag{4}$$

As shown in Equations (3) and (4), in an interactive environment where *K* changes, the TUFP algorithm needs to re-scan the UDB and rebuild the UP-Lists and CUP-Lists. While the ITUFP algorithm can extract the Top-*K* UFPs only by updating the IMCUP-Lists. Since the ITUFP algorithm does not require T_{UDB} and T_{up} and $T_{u\text{-}IMCUP} \leq T_{cup}$, it is

Main_Procedure: ITUFP

Input: An Uncertain DataBase, ***UDB***
A set of user-defined *K*, ***K***= *{K1, K2, K3, ..., Ki}*

Output: an array of Top-*K* UFP results, T^k

Variable: A set of UP-Lists, **U**
A set of IMCUP-Lists, **IMC**
A prefix itemset, **pref**
A *minSup* with initial value zero, *minSup*

```
1.  Generate an ux for each item in UDB by a database scan and U ← U ∪ ux
2.  Sort UP-Lists in their expSup descending order
3.  For each mining request with Ki
4.      Define an array of length K, T^k, and a minSup with initial value zero
5.      T^k ← T^k ∪ the first K items in UP-Lists
6.      IF Tk has K elements
7.        Update minSup with the expSup value of the last element of T^k
8.      End if
9.      Call Mine_Patterns (U, Ki, T^k, minSup)
10. End For
```

Fig. 7. Main procedure of ITUFP algorithm.

```
Sub_procedure: Mine_Patterns (a set of UP-Lists, U, K, T^k, minSup)
1.  For each u_x in U
2.    If (u_x.expSup > minSup)
3.      If no set of IMCUP-Lists exists for u_x, then initialize a set of IMCUP-Lists, IMC and pref
4.      Insert u_x.item into pref
5.      For each u_l in U // x<l
6.        If (u_l.expSup > minSup)
7.          If ((u_x.expSup * max of u_l) > minSup)  // Checking PUFP condition
8.            If IMCUP-List from u_x and u_l does not exist
9.              Generate an IMCUP-List, c, from u_x and u_l while checking the ppf condition regularly
10.             Set c.Index1 = index of the u_x's last processed tuple and c.Index2 = index of the u_l's last
                processed tuple
11.             IMC ← IMC ∪ c
12.           Else
13.             If (c.Index1 != index of the u_x's last tuple and c.Index2 != index of the u_l's last tuple)
14.               Update existing u_xu_l's IMCUP-List according to the technique PUT while checking the ppf
                  condition regularly
15.             End if
16.             Set c.Index1 = index of u_x's last processed tuple and c.Index2 = index of u_l's last processed tuple
17.           End if
18.           If (c.expSup > minSup)
19.           T^k ← T^k ∪ c.pattern and Sort T^k
20.           End if
21.           IF T_k has K elements
22.             Update minSup with the expSup value of the last element of T^k
23.           End if
24.         End if
25.       End if
26.     End for
27.     If (# of elements in IMC>1)
28.       Call ITUFP_growth (IMC, pref, K, T^k, minSup)
29.     End if
30.   End if
31. End for
32. Return R

Sub_procedure:  ITUFP_growth (a set of IMCUP-Lists, IMC, a prefix itemset, pref, K, T^k, minSup)
33. For each c_x in IMC
34.   If (c_x.expSup > minSup)
35.     If no set of IMCUP-Lists exists for c_x, then initialize a set of IMCUP-Lists, IMC'
36.     Initialize a prefix itemset for the next step, pref', with pref and c_x
37.     For each c_l in IMC // x<l
38.       If (c_l.expSup > minSup)
39.         If ((c_x.expSup * max of c_l) > minSup)  // Checking PUFP condition
40.           If IMCUP-List from c_x and c_l does not exist
41.             Generate an IMCUP-List, c', from c_x and c_l while checking the ppf condition regularly
42.             Set c'.Index1= index of c_x's last processed tuple and c'.Index2 = index of c_l's last processed tuple
43.             IMC' ← IMC' ∪ c'
44.           Else
45.             If ( c'.Index1!= index of c_x's last tuple and c'.Index2 != index of c_l's last tuple)
46.               Update the existing c_xc_l's IMCUP-List according to the technique PUT while checking the ppf
                  condition regularly
47.             End if
48.             Set c'.Index1= index of c_x's last processed tuple and c'.Index2 = index of c_l's last processed tuple
49.           End if
50.           If (c'.expSup > minSup)
51.           T^k ← T^k ∪ c'.pattern and Sort T^k
52.           End if
53.           IF T_k has K elements
54.             Update minSup with the expSup value of the last element of T^k
55.           End if
56.         End if
57.       End if
58.     End for
59.     If (# of elements in IMC'>1)
60.       Call  ITUFP_growth (IMC', pref',  K, T^k, minSup )
61.     End if
62.   End if
63. End for
```

Fig. 8. Sub procedures of ITUFP algorithm.

clear that $T_{ITUFP} < T_{TUFP}$.

5. Performance evaluation

To evaluate the ITUFP algorithm in interactive environments, the ITUFP algorithm was compared with the TUFP algorithm in terms of newly constructed lists, runtime, and scalability concerning the number of transactions. A complete set of evaluations was performed on several real and synthetic datasets. The u12I5D300K, T40I10D100K, T10I4D100K, and T10I5D1M are synthetic sparse datasets generated by the IBM Quest Synthetic Data Generator (http://www.almaden.ibm.com/cs/quest/syndata.html). The mushroom, Chess, and Connect-4 are dense real datasets, and Retail, Pumsb*, BMS-Webview-1, and BMS-POS are sparse real datasets obtained from the UCI repository (http://archive.ics.uci.edu/ml/index.php) and the FIMI repository (http://fimi.ua.ac.be/data/).

Table 3 summarizes some statistical information about these datasets. The fifth column shows the average length of the transactions and

Table 2
Symbol definitions for the time complexity.

Symbol	Meaning
T_{UDB}	Time cost for scanning the UDB
T_{up}	Time cost for generating UP-Lists from scratch
T_{cup}	Time cost for generating CUP-Lists from scratch
$T_{u\text{-}IMCUP}$	Time cost for updating IMCUP-Lists
T_{TUFP}	Time required for interactive mining of Top-K UFPs by the TUFP
T_{ITUFP}	Time required for interactive mining of Top-K UFPs by the proposed ITUFP

Table 3
Dataset characteristics.

Dataset	Dense/Sparse	#Trans	#Items	Trans Len	
				Ave	Max
Chess	Dense	3,196	75	37	37
mushroom	Dense	8,124	119	23	23
Connect-4	Dense	67,557	129	43	43
Pumsb*	Sparse	49,046	2088	50.48	63
BMS-Webview-1	Sparse	59,602	497	2.50	267
Retail	Sparse	88,162	16,470	10.31	76
BMS-POS	Sparse	515,597	1,657	6.53	164
T10I4D100K	Sparse	100,000	870	10.10	29
T40I10D100K	Sparse	100,000	942	39.61	77
u12I5D300K	Sparse	300,000	1000	12.25	31
T10I5D1M	Sparse	1,000,000	1000	10	32

the sixth column shows the maximal length of the transactions. The existential probability (randomly generated) of each item in these datasets is a unique value between (0, 1] (Davashi, 2021a; Leung & Tanbeer, 2012; Leung & Tanbeer, 2013; Leung et al., 2014). The algorithms were implemented in C#. The programs run with Windows 10 OS on a personal 64-bit laptop with Intel(R) Core(TM) i7-6700 HQ CPU 2.60 GHz, 16 GB RAM. In each case, multiple runs were performed, and the average of these runs was reported as the result. To determine the effect of the number of newly constructed lists on runtime, the experiments on the number of newly constructed lists and runtime were performed with the same K values.

5.1. Evaluating the number of newly constructed lists

In an interactive environment, whenever the K changes by the user, the TUFP and ITUFP algorithms must extract the Top-K UFPs by generating lists. Therefore, the overall performance of TUFP and ITUFP depends on the number of newly constructed lists. The number of newly constructed lists is evaluated for different values of K on mushroom, Chess, connect-4, Retail, Pumsb*, BMS-Webview-1, BMS-POS, T10I4D100K, T40I10D100K, and u12I5D300K datasets. In Fig. 9(a to j), the results of the evaluations are given only when K values increase. This is because in the opposite case, when the K values decrease, the set of Top-K UFPs is obtained from the previous result set with a higher K value. When the user makes the first mining request, both algorithms produce the same number of lists, as shown in Fig. 9(a to j).

After that, whenever the user changes the K value, the TUFP algorithm must rebuild the UP-Lists and CUP-Lists from scratch. In contrast, the ITUFP algorithm does not need to rebuild the UP-Lists and IMCUP-Lists. This is because the ITUFP algorithm has stored the lists constructed in the previous steps. It updates the existing IMCUP-Lists if necessary and creates new IMCUP-Lists only for new patterns. When the K value is increased, the TUFP algorithm generates more new lists. In contrast, the proposed ITUFP algorithm generates significantly fewer new lists because most of the lists have been generated and stored in the previous steps. For example, as shown in Fig. 9(f), the TUFP algorithm generates 16,548 new lists in the Retail dataset for $K = 20$, while the proposed ITUFP algorithm generates only 30 new lists.

5.2. Runtime evaluation

The runtime for the TUFP and ITUFP algorithms in interactive environments is evaluated using the mushroom, Chess, connect-4, Retail, BMS-Webview-1, Pumsb*, BMS-POS, T10I4D100K, T40I10D100K, and u12I5D300K datasets. Fig. 10(a to j) illustrate the results of these assessments. The x-axis indicates the increasing values of K and the y-axis indicates the runtime required for scanning the database and mining the Top-K UFPs. In an interactive environment, each time K changes, the TUFP algorithm must re-scan the database and generate all UP-Lists and CUP-Lists from scratch to extract the Top-K UFPs with the new K value. Therefore, as the K value is increased and more Top-K UFPs are extracted, the TUFP algorithm takes more time for mining the patterns.

In contrast, the proposed algorithm stores all UP-Lists and IMCUP-Lists after the first mining process. Since the database is fixed, these UP-Lists are used unchanged in the next steps. In other words, whenever the K changes, the proposed ITUFP algorithm does not need to re-scan the database and rebuild UP-Lists. Moreover, the ITUFP algorithm does not need to rebuild IMCUF-Lists from scratch and only updates them. Therefore, as the evaluations in Fig. 10(a to j) show, the proposed algorithm requires less time for interactive mining of Top-K UFPs than the TUFP algorithm. On the other hand, the gap between the performance curves of the ITUFP and TUFP algorithms is increased when the number of K is increased. Indeed, after several steps of mining, the proposed ITUFP algorithm has generated and stored more lists in the previous steps. Therefore, the ITUFP algorithm does not need to generate or update many lists. In this case, the TUFP algorithm needs to create all lists from scratch. For example, in the Retail dataset for $K = 20$ (Fig. 10(f)), the proposed ITUFP algorithm can extract the Top-K UFPs about 87 % faster than TUFP. As expected in interactive environments, the ITUFP algorithm dramatically improves runtime compared to the TUFP algorithm for all datasets.

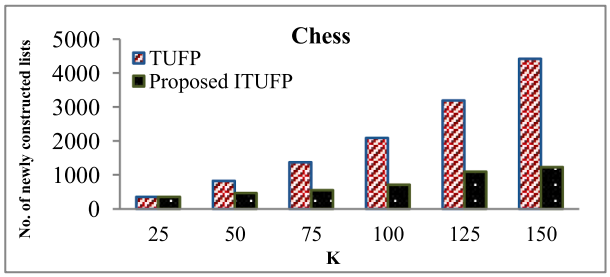
As can be seen from the evaluations on the number of newly constructed lists and the runtime, the proposed algorithm works well for all data. Indeed, there is no noticeable difference in the efficiency of the proposed algorithm for large and small datasets.

5.3. Scalability evaluation

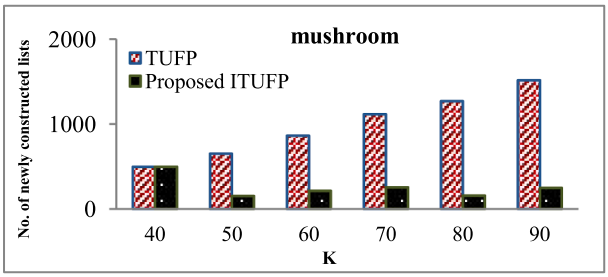
The scalability evaluation was performed with regard to the number of transactions for dataset T10I5D1M in Fig. 11. In this evaluation, for 0.2 million transactions, the user initially sets the value of K to 5. Then, the user changes the value of K to 10 and 15 and the runtime is measured. Similarly, each time the database size is increased, the runtime is measured in an interactive environment where the user changes K from 5 to 10 and 15. As can be seen in Fig. 11, the overall runtime for mining Top-K UFPs is increased as the size of the database is increased. From the performance evaluation, the ITUFP is more scalable than the TUFP for large datasets.

6. Conclusion and future work

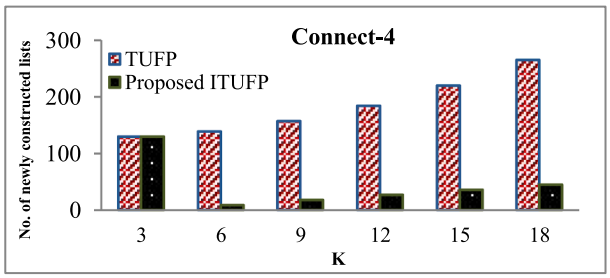
In this paper, a fast method called ITUFP was proposed for interactive mining of Top-K UFPs. In interactive environments where the user constantly changes the K threshold, the static TUFP algorithm needs to scan the UDB and create the UP-Lists and CUP-Lists from scratch. However, the proposed ITUFP is able to store lists information well by providing a new data structure called IMCUP-List. The proposed ITUFP stores all UP-Lists and IMCUP-Lists constructed after each step. When K changes, the proposed ITUFP does not need to re-scan the database for building UP-Lists and can easily update the IMCUP-Lists. The runtime and scalability evaluation results show the efficiency of the proposed ITUFP algorithm in interactive environments compared to the TUFP algorithm. However, the proposed algorithm is not very efficient in incremental environments where new databases are constantly added. Indeed, databases change frequently in real-world applications. When a



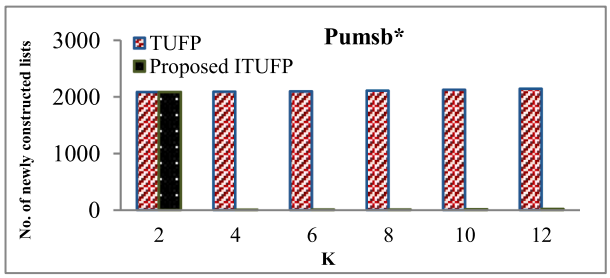
(a) Chess dataset



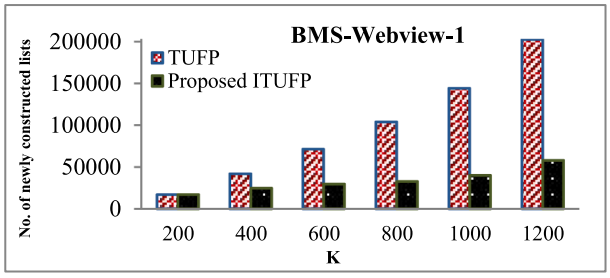
(b) mushroom dataset



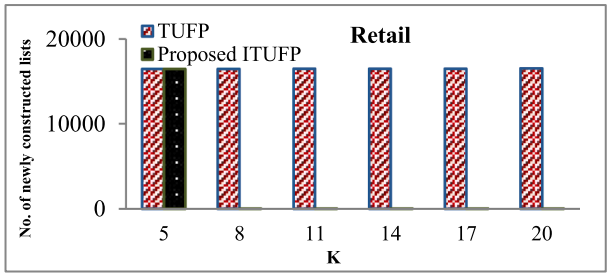
(c) Connect-4 dataset



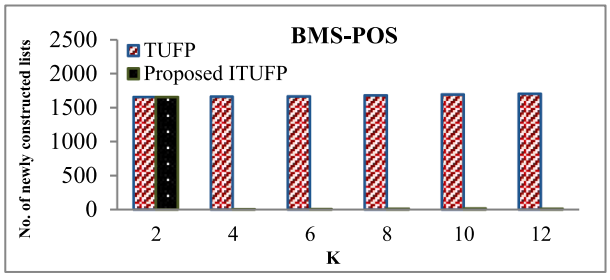
(d) Pumsb* dataset



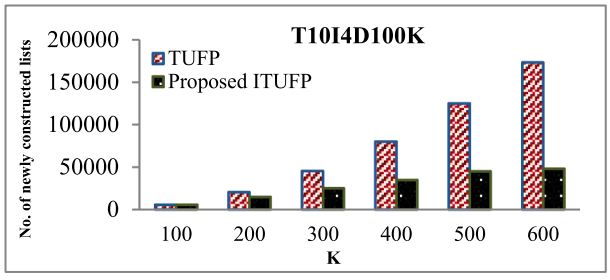
(e) BMS-Webview-1 dataset



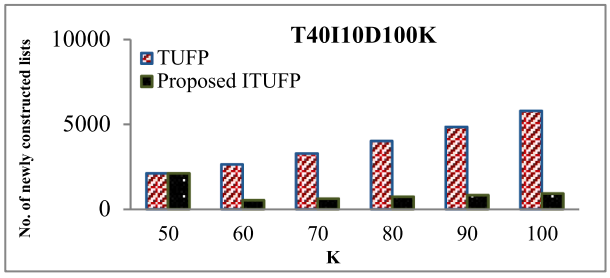
(f) Retail dataset



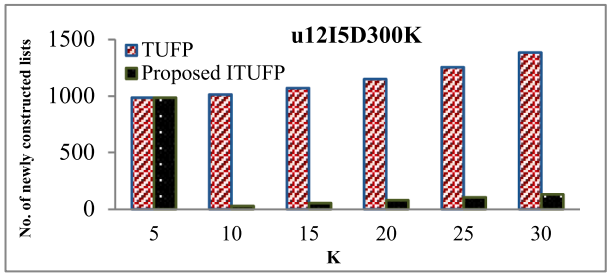
(g) BMS-POS dataset



(h) T10I4D100K dataset



(i) T40I10D100K dataset



(j) u12I5D300K dataset

Fig. 9. Experimental results: The number of newly constructed lists.

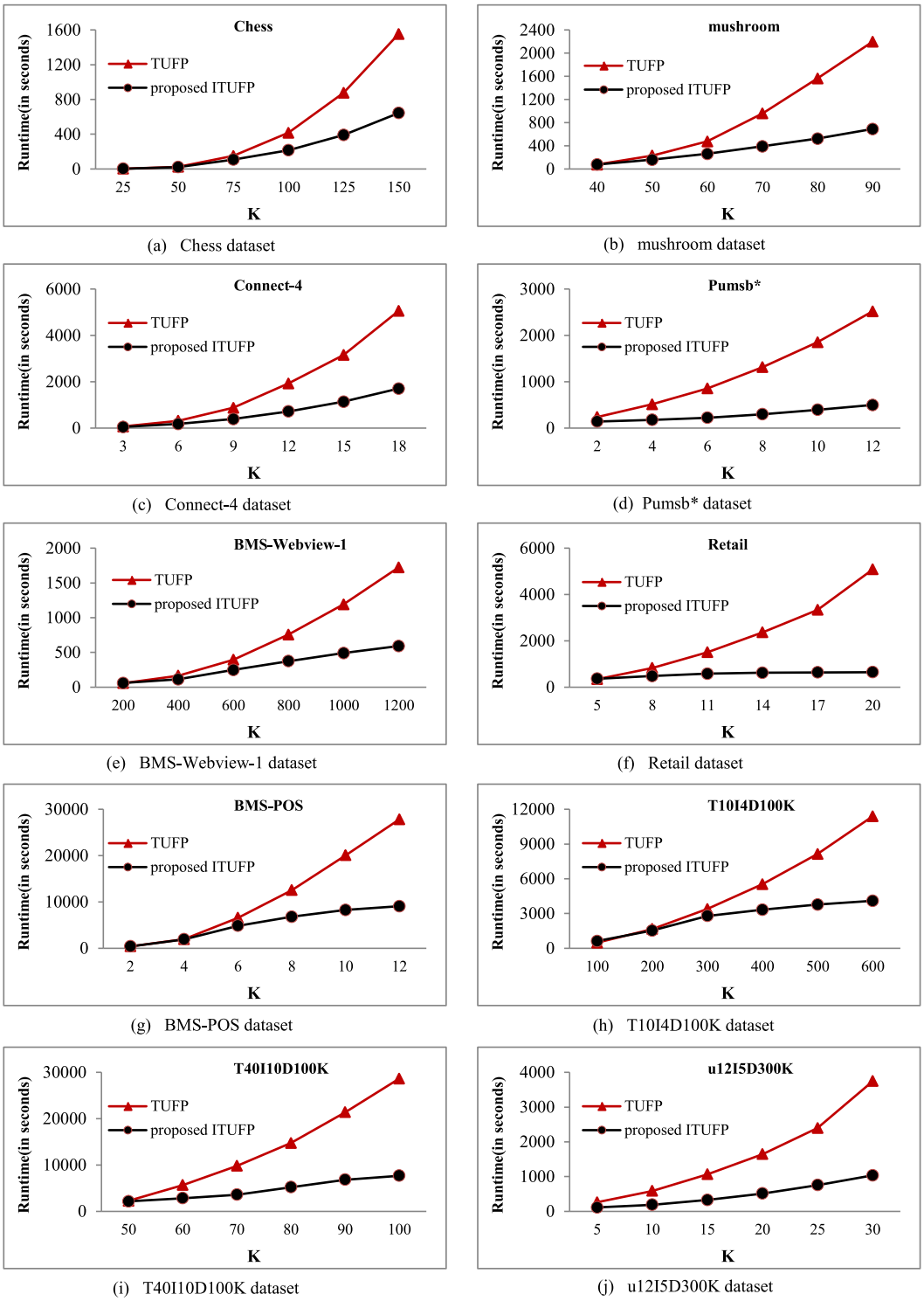


Fig. 10. Experimental results: The RunTime.

new database is added, the proposed algorithm has to construct the lists from scratch, which is very time consuming. Future work will add incremental capability to the proposed algorithm so that lists do not need to be reconstructed each time the database is updated.

CRediT authorship contribution statement

Razieh Davashi: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing.

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 paper.

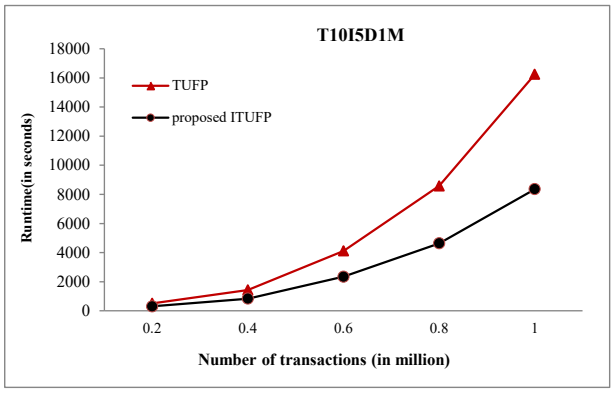


Fig. 11. Experimental results: Scalability with No. of transactions.

Data availability

The address of the data repository is available in the paper

References

Abd-Elmegid, L. A., El-Sharkawi, M. E., El-Fangary, L. M., & Helmy, Y. K. (2010). Vertical mining of frequent patterns from uncertain data. *Computer and Information Science, 3*(2), 171–179.

Aggarwal, C.C., Li, Y., Wang, J., & Wang, J. (2009). Frequent pattern mining with uncertain data. In Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 29–38).

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. In Proceedings of the 20th International Conference on Very Large Data Bases, VLDB (pp. 487–499).

Ahmed, C. F., Tanbeer, S. K., Jeong, B. S., & Choi, H. J. (2012). Interactive mining of high utility patterns over data streams. *Expert Systems with Applications, 39*(15), 11979–11991.

Ahmed, C. F., Tanbeer, S. K., Jeong, B.-S., Lee, Y.-K., & Choi, H. J. (2012). Single-pass incremental and interactive mining for weighted frequent patterns. *Expert Systems with Applications, 39*(9), 7976–7994.

Bhadoria, R. S., Kumar, R., & Dixit, M. (2011). Analysis on probabilistic and binary datasets through frequent itemset mining. In *In 2011 World Congress on Information and Communication Technologies* (pp. 263–267). IEEE.

Calders, T., Garboni, C., & Goethals, B. (2010). Efficient pattern mining of uncertain data with sampling. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (pp. 480–487). Springer.

Cheung, Y.-L., & Fu, A. W. C. (2004). Mining frequent itemsets without support threshold: With and without item constraints. *IEEE transactions on knowledge and data engineering, 16*(9), 1052–1069.

Chui, C. K., & Kao, B. (2008). A decremental approach for mining frequent itemsets from uncertain data. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (pp. 64–75). Springer.

Chui, C. K., Kao, B., & Hung, E. (2007). Mining frequent itemsets from uncertain data. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (pp. 47–58). Springer.

Davashi, R. (2021a). ILUNA: Single-pass incremental method for uncertain frequent pattern mining without false positives. *Information Sciences, 564*, 1–26.

Davashi, R. (2021b). UP-tree & UP-Mine: A fast method based on upper bound for frequent pattern mining from uncertain data. *Engineering Applications of Artificial Intelligence, 106*, Article 104477.

Davashi, R., & Nadimi-Shahraki, M. H. (2019). EFP-tree: An efficient FP-tree for incremental mining of frequent patterns. *International Journal of Data Mining, Modelling and Management, 11*(2), 144–166.

Dong, X., Qiu, P., Lü, J., Cao, L., & Xu, T. (2019). Mining Top-k Useful Negative Sequential Patterns via Learning. *IEEE transactions on neural networks and learning systems, 30*(9), 2764–2778.

Fu, A., & W.c., Kwong, R. W.w., & Tang, J.. (2000). Mining n-most interesting itemsets. In *International symposium on methodologies for intelligent systems* (pp. 59–67). Springer.

Han, J., Pei, J., & Yin, Y. (2000). Mining frequent patterns without candidate generation. *ACM sigmod record, 29*(2), 1–12.

Han, J., Wang, J., Lu, Y., & Tzvetkov, P. (2002). Mining top-k frequent closed patterns without minimum support. In *In 2002 IEEE International Conference on Data Mining, 2002. Proceedings* (pp. 211–218). IEEE.

Han, X., Liu, X., Li, J., & Gao, H. (2021). Efficient top-k high utility itemset mining on massive data. *Information Sciences, 557*, 382–406.

Hirate, Y., Iwahashi, E., & Yamana, H. (2004). TF2P-growth: An efficient algorithm for mining frequent patterns without any thresholds. In Proc. of ICDM.

Krishnamoorthy, S. (2019). Mining top-k high utility itemsets with effective threshold raising strategies. *Expert Systems with Applications, 117*, 148–165.

Le, T., Vo, B., Huynh, V. N., Nguyen, N. T., & Baik, S. W. (2020). Mining top-k frequent patterns from uncertain databases. *Applied Intelligence, 50*(5), 1487–1497.

Lee, G., & Yun, U. (2017). A new efficient approach for mining uncertain frequent patterns using minimum data structure without false positives. *Future Generation Computer Systems, 68*, 89–110.

Lee, G., & Yun, U. (2018). Single-pass based efficient erasable pattern mining using list data structure on dynamic incremental databases. *Future Generation Computer Systems, 80*, 12–28.

Leung, C. K. S., Carmichael, C. L., & Hao, B. (2007). Efficient mining of frequent patterns from uncertain data. In *Seventh IEEE International Conference on Data Mining Workshops (ICDMW 2007)* (pp. 489–494). IEEE.

Leung, C. K. S., Khan, Q. I., Li, Z., & Hoque, T. (2007). CanTree: A canonical-order tree for incremental frequent-pattern mining. *Knowledge and Information Systems, 11*(3), 287–311.

Leung, C. K. S., & MacKinnon, R. K. (2014). BLIMP: A compact tree structure for uncertain frequent pattern mining. In *International Conference on Data Warehousing and Knowledge Discovery* (pp. 115–123). Springer.

Leung, C. K. S., & MacKinnon, R. K. (2015). Balancing tree size and accuracy in fast mining of uncertain frequent patterns. In *International Conference on Big Data Analytics and Knowledge Discovery* (pp. 57–69). Springer.

Leung, C. K. S., Mateo, M. A. F., & Brajczuk, D. A. (2008). A tree-based approach for frequent pattern mining from uncertain data. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (pp. 653–661). Springer.

Leung, C.K.S., & Sun, L. (2011). Equivalence class transformation based mining of frequent itemsets from uncertain data. In Proceedings of the 2011 ACM Symposium on Applied Computing (pp. 983–984).

Leung, C. K. S., & Tanbeer, S. K. (2012). Fast tree-based mining of frequent itemsets from uncertain data. In *International Conference on Database Systems for Advanced Applications* (pp. 272–287). Springer.

Leung, C. K. S., & Tanbeer, S. K. (2013). In *PUF-tree: a compact tree structure for frequent pattern mining of uncertain data* (pp. 13–25). Springer.

Leung, C. K. S., Tanbeer, S. K., Budhia, B. P., & Zacharias, L. C. (2012). Mining probabilistic datasets vertically. In *In Proceedings of the 16th International Database Engineering & Applications Sysmposium* (pp. 199–204).

Leung, C. K., MacKinnon, R. K., & Tanbeer, S. K. (2014). Tightening upper bounds to the expected support for uncertain frequent pattern mining. *Procedia Computer Science, 35*, 328–337.

Lin, C. W., & Hong, T. P. (2012). A new mining approach for uncertain databases using CUFP trees. *Expert Systems with Applications, 39*(4), 4084–4093.

Liu, X., Niu, X., & Fournier-Viger, P. (2021). Fast top-k association rule mining using rule generation property pruning. *Applied Intelligence, 51*(4), 2077–2093.

MacKinnon, R. K., Strauss, T. D., & Leung, C. K. S. (2014). In *DISC: efficient uncertain frequent pattern mining with tightened upper bounds* (pp. 1038–1045). IEEE.

Nguyen, L. T., Vo, B., Nguyen, L. T., Fournier-Viger, P., & Selamat, A. (2018). ETARM: An efficient top-k association rule mining algorithm. *Applied Intelligence, 48*(5), 1148–1160.

Pham, T.-T., Do, T., Nguyen, A., Vo, B., & Hong, T. P. (2020). An efficient method for mining top-K closed sequential patterns. *IEEE Access, 8*, 118156–118163.

Pyun, G., & Yun, U. (2014). Mining top-k frequent patterns with combination reducing techniques. *Applied Intelligence, 41*(1), 76–98.

Quang, T. M., Oyanagi, S., & Yamazaki, K. (2006). In *ExMiner: An efficient algorithm for mining top-k frequent patterns* (pp. 436–447). Springer.

Shen, L., Hong, S., & Prithard, P. (1998). Finding the N Largest Itemsets in Data Mining. In *In Proc. of International Conference on Data Mining* (pp. 211–222).

Song, C., Liu, X., Ge, T., & Ge, Y. (2019). Top-k frequent items and item frequency tracking over sliding windows of any size. *Information Sciences, 475*, 100–120.

Sun, X., Lim, L., & Wang, S. (2012). An approximation algorithm of mining frequent itemsets from uncertain dataset. *International Journal of Advancements in Computing Technology, 4*(3), 42–49.

Tanbeer, S. K., Ahmed, C. F., Jeong, B. S., & Lee, Y. K. (2009). Efficient single-pass frequent pattern mining using a prefix-tree. *Information Sciences, 179*(5), 559–583.

Wang, J., Fang, S., Liu, C., Qin, J., Li, X., & Shi, Z. (2020). Top-k closed co-occurrence patterns mining with differential privacy over multiple streams. *Future Generation Computer Systems, 111*, 339–351.

Wang, L., Cheung, D. W. L., Cheng, R., Lee, S. D., & Yang, X. S. (2011). Efficient mining of frequent item sets on large uncertain databases. *IEEE transactions on knowledge and data engineering, 24*(12), 2170–2183.

Wang, L., Feng, L., & Wu, M. (2013). AT-mine: An efficient algorithm of frequent itemset mining on uncertain dataset. *journal of computers, 8*(6), 1417–1427.

Zhang, C., Du, Z., Gan, W., & Philip, S. Y. (2021). TKUS: Mining top-k high utility sequential patterns. *Information Sciences, 570*, 342–359.