PUF-Tree: A Compact Tree Structure for Frequent Pattern Mining of Uncertain Data

Carson Kai-Sang Leung[⋆] and Syed Khairuzzaman Tanbeer

Dept. of Computer Science, University of Manitoba, Canada
{kleung,tanbeer}@cs.umanitoba.ca

Abstract. Many existing algorithms mine frequent patterns from traditional databases of *precise* data. However, there are situations in which data are *uncertain*. In recent years, researchers have paid attention to frequent pattern mining from uncertain data. When handling uncertain data, UF-growth and UFP-growth are examples of well-known mining algorithms, which use the UF-tree and the UFP-tree respectively. However, these trees can be large, and thus degrade the mining performance. In this paper, we propose (i) a more compact tree structure to capture uncertain data and (ii) an algorithm for mining all frequent patterns from the tree. Experimental results show that (i) our tree is usually more compact than the UF-tree or UFP-tree, (ii) our tree can be as compact as the FP-tree, and (iii) our mining algorithm finds frequent patterns efficiently.

1 Introduction

Since the introduction of frequent pattern mining [1], there have been numerous studies on mining and visualizing *frequent patterns* (i.e., *frequent itemsets*) from *precise* data such as databases (DBs) of market basket transactions [8,11,12]. Users definitely know whether an item is present in, or is absent from, a transaction in these DBs of precise data. However, there are situations in which users are uncertain about the presence or absence of some items or events [3,4,5,10,14,16]. For example, a physician may highly suspect (but cannot guarantee) that a patient suffers from flu. The uncertainty of such suspicion can be expressed in terms of existential probability. For instance, a patient may have a 90% likelihood of having the flu, and a 20% likelihood of having a cold regardless of having the flu or not. With this notion, each item in a transaction t_j in DBs containing precise data can be viewed as an item with a 100% likelihood of being present in t_j.

To deal with uncertain data, the *U-Apriori algorithm* [6] was proposed in PAKDD 2007. As an Apriori-based algorithm, U-Apriori requires multiple scans of uncertain DBs. To reduce the number of DB scans (down to two), the tree-based *UF-growth algorithm* [13] was proposed in PAKDD 2008. In order to compute the expected support of each pattern *exactly*, paths in the corresponding UF-tree are shared only if tree nodes on the paths have the same item and same existential probability. Hence, the UF-tree may not be too compact. In

[⋆] Corresponding author.

an attempt to make the tree compact, the *UFP-growth algorithm* [2] groups *similar* nodes (with the same item x and similar existential probability values) into a cluster. However, depending on the clustering parameter, the corresponding UFP-tree may be as large as the UF-tree (i.e., no reduction in tree size). Moreover, UFP-growth returns not only the frequent patterns but also some infrequent patterns (i.e., false positives). As alternatives to trees, hyper-structures were used by the *UH-Mine algorithm* [2], which was reported [15] to outperform UFP-growth.

Recently, we studied fast tree-based mining of frequent patterns from uncertain data [14]. In the current paper, we study the following questions: Can we further reduce the tree size (e.g., smaller than the UFP-tree)? Can the resulting tree be as compact as the FP-tree? How to mine frequent patterns from such a compact tree? Would such a mining algorithm be faster than UH-Mine? Our **key contributions** of this paper are as follows:

1. a **p**refix-capped **u**ncertain **f**requent pattern **tree (PUF-tree)**, which can be as compact as the original FP-tree; and
2. a mining algorithm (namely, **PUF-growth**), which is guaranteed to find *all and only those* frequent patterns (i.e., *no* false negatives and *no* false positives) from uncertain data.

The remainder of this paper is organized as follows. The next section presents background and related works. We then propose our PUF-tree structure and PUF-growth algorithm in Sections 3 and 4, respectively. Experimental results are shown in Section 5, and conclusions are given in Section 6.

2 Background and Related Works

In this section, we first give some background information about frequent pattern mining of uncertain data (e.g., existential probability, expected support), and we then discuss some related works.

Let (i) $\texttt{Item}$ be a set of m domain items and (ii) $X = \{x_1, x_2, \ldots, x_k\}$ be a k-itemset (i.e., a pattern consisting of k items), where $X \subseteq \texttt{Item}$ and $1 \leq k \leq m$. Then, a transactional DB $= \{t_1, t_2, \ldots, t_n\}$ is the set of n transactions, where each transaction $t_j \subseteq \texttt{Item}$. The projected DB of X is the set of all transactions containing X.

Unlike precise DBs, each item x_i in a transaction $t_j = \{x_1, x_2, \ldots, x_h\}$ in an uncertain DB is associated with an ***existential probability value*** $\boldsymbol{P(x_i, t_j)}$, which represents the likelihood of the presence of x_i in t_j [9]. Note that $0 < P(x_i, t_j) \leq 1$. The ***existential probability*** $\boldsymbol{P(X, t_j)}$ **of a pattern** $\boldsymbol{X}$ **in** $\boldsymbol{t_j}$ is then the product of the corresponding existential probability values of items within X when these items are independent [9]: $P(X, t_j) = \prod_{x \in X} P(x, t_j)$. The ***expected support*** $\boldsymbol{expSup(X)}$ of X in the DB is the sum of $P(X, t_j)$ over all n transactions in the DB: $expSup(X) = \sum_{j=1}^{n} P(X, t_j)$. A pattern X is ***frequent*** in an uncertain DB if $expSup(X) \geq$ a user-specified minimum support threshold *minsup*. Given a DB and *minsup*, the research problem of **frequent pattern**

Table 1. A transactional DB (*minsup*=0.5)

TID	**Transactions**	**Sorted transactions (with infrequent items removed)**
t_1	{a:0.2, b:0.2, c:0.7, f:0.8}	{a:0.2, c:0.7, f:0.8}
t_2	{a:0.5, c:0.9, e:0.5}	{a:0.5, c:0.9, e:0.5}
t_3	{a:0.3, d:0.5, e:0.4, f:0.5}	{a:0.3, e:0.4, f:0.5, d:0.5}
t_4	{a:0.9, b:0.2, d:0.1, e:0.5}	{a:0.9, e:0.5, d:0.1}

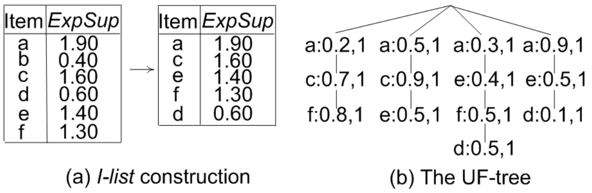

Fig. 1. The UF-tree for the DB shown in Table 1 when *minsup*=0.5

mining from uncertain data is to discover from the DB a complete set of frequent patterns having expected support $\geq$ *minsup*.

Recall from Section 1 that the tree-based **UF-growth algorithm** [13] uses **UF-trees** to mine frequent patterns from uncertain DBs in two DB scans. Each node in a UF-tree captures (i) an item x, (ii) its existential probability, and (iii) its occurrence count. Tree paths are shared if the nodes on these paths share the same ⟨item, existential probability⟩-value. In general, when dealing with uncertain data, it is not uncommon that the existential probability values of the same item vary from one transaction to another. As such, the resulting UF-tree may not be as compact as the FP-tree. See Fig. 1, which shows a UF-tree for the DB presented in Table 1 when *minsup*=0.5. The UF-tree contains four nodes for item a with different probability values as children of the root. Efficiency of the corresponding UF-growth algorithm, which finds all and *only those* frequent patterns, partially relies on the compactness of the UF-tree.

In an attempt to make the tree more compact, the **UFP-growth algorithm** [2] was proposed. Like UF-growth, the UFP-growth algorithm also scans the DB twice and builds a **UFP-tree**. As nodes for item x having similar existential probability values are clustered into a mega-node, the resulting mega-node in the UFP-tree captures (i) an item x, (ii) the *maximum* existential probability value (among all nodes within the cluster), and (iii) its occurrence count (i.e., the number of nodes within the cluster). Tree paths are shared if the nodes on these paths share the same item but *similar* existential probability values. In other words, the path sharing condition is less restrictive than that of the UF-tree. By extracting appropriate tree paths and constructing UFP-trees for subsequent projected DBs, UFP-growth finds all frequent patterns and some *false positives* at the end of the second DB scan. The third DB scan is then required to remove those false positive.

The **UH-Mine algorithm** [2] stores all frequent items in each DB transaction in a hyper-structure called **UH-struct**. As UH-Mine does not take advantage of prefix-sharing, the size of the resulting UH-struct is always as large as that of the DB for the frequent items. However, UH-Mine was reported [2,15] to be faster than UFP-growth.

3 Our PUF-Tree Structure

To reduce the size of the UF-tree and UFP-tree, we propose the **prefix-capped uncertain frequent pattern tree (PUF-tree)** structure, in which important information about uncertain data is captured so that frequent patterns can be mined from the tree. The PUF-tree is constructed by considering an upper bound of existential probability value for each item when generating a k-itemset (where $k > 1$). We call the upper bound of an item x_r in a transaction t_j the **(prefixed) item cap** of x_r in t_j, as defined below.

Definition 1. The **(prefixed) item cap $\boldsymbol{I^{Cap}(x_r, t_j)}$** of an item x_r in a transaction $t_j = \{x_1, \ldots, x_r, \ldots, x_h\}$, where $1 \leq r \leq h$, is defined as the product of $P(x_r, t_j)$ and the highest existential probability value M of items from x_1 to x_{r-1} in t_j (i.e., in the *proper prefix* of x_r in t_j):

$$I^{Cap}(x_r, t_j) = \begin{cases} P(x_r, t_j) \times M & \text{if } h > 1 \\ P(x_1, t_j) & \text{if } h = 1 \end{cases}, \text{ where } M = \max_{1 \leq q \leq r-1} P(x_q, t_j). \quad \square$$

Example 1. Consider an uncertain DB with four transactions as presented in the second column in Table 1. The item cap of c in t_1 can be computed as $I^{Cap}(c, t_1) = 0.7 \times \max\{P(a, t_1), P(b, t_1)\} = 0.7 \times \max\{0.2, 0.2\} = 0.7 \times 0.2 = 0.14$. Similarly, the item cap of f in t_1 is $I^{Cap}(f, t_1) = 0.8 \times \max\{P(a, t_1),\ P(b, t_1), P(c, t_1)\} = 0.8 \times max\{0.2, 0.2, 0.7\} = 0.8 \times 0.7 = 0.56$. □

Note that $I^{Cap}(x_r, t_j)$ provides us with an important property of covering the existential probabilities of all possible patterns containing x_r and its prefix in t_j, as stated in the following theorem.

Theorem 1. Let X be a k-itemset in transaction t_j (where $k > 1$). The existential probability of any of its non-empty proper subset Y that shares the same suffix x_r as X (i.e., $Y \subset X \subseteq t_j$) is always less than or equal to $I^{Cap}(x_r, t_j)$.

Proof. Let (i) $t_j = \{x_1, \ldots, x_r, \ldots, x_h\}$, (ii) $X = \{x_s, \ldots, x_r\}$ be a k-itemset in t_j, and (iii) $Y = \{x_t, \ldots, x_r\}$ be a k'-itemset (where $k' < k$) and a proper subset of X (i.e., $Y \subset X \subseteq t_j$). Then, recall from Section 2 that the existential probability $P(Y, t_j)$ of Y in t_j is defined as follows: $P(Y, t_j) = \prod_{x \in Y} P(x, t_j)$, which is equivalent to $P(x_r, t_j) \times \prod_{(x \in Y) \wedge (x \neq x_r)} P(x, t_j)$. Note that $0 < P(x, t_j) \leq 1$. So, $\prod_{(x \in Y) \wedge (x \neq x_r)} P(x, t_j) \leq \max_{1 \leq q \leq r-1} P(x_q, t_j)$. Hence, $P(Y, t_j) = P(x_r, t_j) \times \prod_{(x \in Y) \wedge (x \neq x_r)} P(x, t_j) \leq P(x_r, t_j) \times \max_{1 \leq q \leq r-1} P(x_q, t_j) = I^{Cap}(x_r, t_j)$. □

Since the expected support of X is the sum of all existential probabilities of X over all the transactions containing X, the *cap* of expected support of X can then be defined as follows.

Definition 2. The **cap of expected support** $\boldsymbol{expSup^{Cap}(X)}$ of a pattern $X = \{x_1, \ldots, x_k\}$ (where $k > 1$) is defined as the sum (over all n transactions in a DB) of all item caps of x_k in all the transactions that contain X: $expSup^{Cap}(X) = \sum_{j=1}^{n} \{I^{Cap}(x_k, t_j) \; | X \subseteq t_j\}$. □

Based on Definition 2, $expSup^{Cap}(X)$ for any k-itemset $X=\{x_1, \ldots, x_k\}$ can be considered as an *upper bound* to the expected support of X, i.e., $expSup(X) \leq expSup^{Cap}(X)$. So, if $expSup^{Cap}(X) < minsup$, then X cannot be frequent. Conversely, if X is a frequent pattern, then $expSup^{Cap}(X)$ must be $\geq minsup$. Hence, we obtain a *safe condition* with respect to $expSup^{Cap}(X)$ and *minsup*. This condition, which helps us to obtain an upper bound of the expected support for each pattern, could be safely applied for mining all frequent patterns.

As the *expected support* satisfies the downward closure property [1], all non-empty subsets of a frequent pattern are also frequent. Conversely, if a pattern is infrequent, then none of its supersets can be frequent. In other words, for $Y \subset X$, (i) $expSup(X) \geq minsup$ implies $expSup(Y) \geq minsup$ and (ii) $expSup(Y) < minsup$ implies $expSup(X) < minsup$.

Example 2. Consider two patterns X & Y. Then, $P(Y, t_j) \geq P(X, t_j)$ for any transaction t_j such that $Y \subset X \subseteq t_j$. Moreover, Y must be present in any transaction whenever X is present, but *not* vice versa. So, the number of transactions containing Y is at least the number of transactions containing X. Hence, $expSup(Y)=\sum_{j=1}^{n} P(Y, t_j) \geq \sum_{j=1}^{n} P(X, t_j)=expSup(X)$ for all n transactions in a DB. If X is frequent (i.e., its expected support $\geq minsup$), then Y is also frequent because $expSup(Y) \geq expSup(X) \geq minsup$. Conversely if Y is infrequent (i.e., its expected support $< minsup$), then X cannot be frequent because $expSup(X) \leq expSup(Y) < minsup$. □

In contrast, the *cap of expected support* generally does *not* satisfy the downward closure property because $expSup^{Cap}(Y)$ can be less than $expSup^{Cap}(X)$ for some proper subset Y of X. See Example 3.

Example 3. Let $t_3=\{a{:}0.3,\ e{:}0.4,\ f{:}0.5,\ d{:}0.5\}$ be the only transaction in the DB containing $X=\{a, e, f, d\}$ and its subset $Y=\{e, f\}$. Then, $expSup^{Cap}(Y) = P(f, t_3) \times \max\{P(a, t_3), P(e, t_3)\}=0.5\times0.4=0.20$ and $expSup^{Cap}(X) = P(d, t_3) \times \max\{P(a, t_3), P(e, t_3), P(f, t_3)\} = 0.5 \times 0.5 = 0.25$. This shows that $expSup^{Cap}(Y)$ can be $<$ $expSup^{Cap}(X)$. □

However, for some *specific* cases (e.g., when X & Y share the same suffix item x_r), the cap of expected support satisfies the downward closure property, as proved in Lemma 1. We call such property the ***partial downward closure property***: For any non-empty subset Y of X such that both X & Y ends with x_r, (i) $expSup^{Cap}(X) \geq minsup$ implies $expSup^{Cap}(Y) \geq minsup$ and (ii) $expSup^{Cap}(Y) < minsup$ implies $expSup^{Cap}(X) < minsup$.

Lemma 1. The cap of expected support of a pattern X satisfies the *partial* downward closure property.

Proof. Let (i) X & Y be two itemsets such that $Y \subset X$, and (ii) X & Y share the same suffix item x_r. Then, Y must be present in any transaction whenever X is present, but *not* vice versa. The number of transactions containing

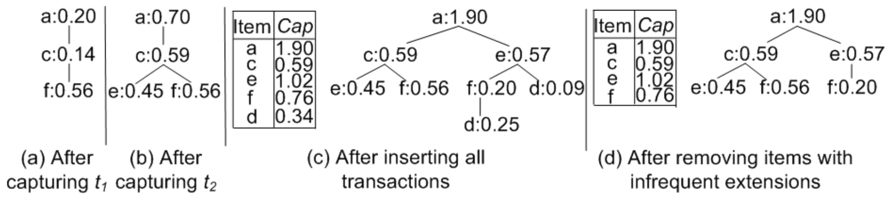

Fig. 2. Our PUF-tree for the DB in Table 1 when *minsup*=0.5

Y is higher than or equal to the number of transactions containing X. For any transaction t_j in which X & Y are present, they share the same suffix item x_r and thus share the same $I^{Cap}(x_r, t_j)$. So, $expSup^{Cap}(Y) = \sum_{j=1}^{n} I^{Cap}(x_r, t_j) | Y \subseteq t_j\} \geq \sum_{j=1}^{n} I^{Cap}(x_r, t_j) | X \subseteq t_j\} = expSup^{Cap}(X)$. As a result, if $expSup^{Cap}(X) \geq minsup$, then $expSup^{Cap}(Y) \geq expSup^{Cap}(X) \geq minsup$. Conversely, if $expSup^{Cap}(Y) < minsup$, then $expSup^{Cap}(X) \leq expSup^{Cap}(Y) < minsup$. In other words, the cap of expected support of a pattern satisfies the *partial* downward closure property. □

To address the limitation of path sharing of UF-tree, we avoid keeping multiple nodes for the same item having different existential probability values. The basic idea is that, instead of storing the *exact* existential probability value for a node in the transaction, we store in the PUF-tree node the *maximum* existential probability value of the prefix from that node up to the root (i.e., the item cap of the item). For any node in a PUF-tree, we maintain (i) an *item* and (ii) its prefixed *item cap* (i.e., the sum of all item caps for transactions that pass through or end at the node).

How to construct a PUF-tree? With the first scan of the DB, we find distinct frequent items in DB and construct a header table called *I-list* to store only frequent items in some consistent order (e.g., canonical order) to facilitate tree construction. Then, the actual PUF-tree is constructed with the second DB scan in a fashion similar to that of the FP-tree [7]. A key difference is that, when inserting a transaction item, we first compute its item cap and then insert it into the PUF-tree according to the *I-list* order. If that node already exists in the path, we update its item cap by adding the computed item cap to the existing item cap. Otherwise, we create a new node with this item cap value. For better understanding of the PUF-tree construction, see Example 4.

Example 4. Consider the DB in Table 1, and let the user-specified support threshold *minsup* be set to 0.5. Let the *I-list* follow the descending order of expected supports of items. After the first DB scan, the contents of the *I-list* after computing the expected supports of all items and after removing infrequent items (e.g., item b) are $\langle a$:1.9, c:1.6, d:0.6, e:1.4, f:1.3$\rangle$. After sorting, the *I-list* becomes $\langle a$:1.9, c:1.6, e:1.4, f:1.3, d:0.6$\rangle$, as shown in Fig. 1.

With the second DB scan, we insert only the frequent items of each transaction (with their respective item cap values) in the *I-list* order. For instance, when inserting transaction t_1=$\{a$:0.2, c:0.7, f:0.8$\}$, items a, c and f (with their respective item cap values 0.2, 0.7×0.2=0.14 and 0.8×0.7=0.56) are inserted in the PUF-tree as shown in Fig. 2(a). Fig. 2(b) shows the status of the PUF-tree after inserting t_2={a:0.5, c:0.9, e:0.5} with item cap values 0.5, 0.9×0.5=0.45 and 0.5×0.9=0.45 for items a, c and e. As t_2 shares a common prefix $\langle a, c\rangle$ with an existing path in the PUF-tree, (i) the item cap values of those items in the common prefix (i.e., a and c) are added to the corresponding nodes and (ii) the remainder of the transaction (i.e., a new branch for item e) is inserted as a child of the last node of the prefix (i.e., as a child of c). After capturing all transactions in the DB in Table 1, we obtain the PUF-tree shown in Fig. 2(c). Similar to the FP-tree, our PUF-tree maintains horizontal node traversal pointers, which are not shown in the figures for simplicity. □

Note that we are *not confined* to sorting and storing items in descending order of expected support. We could use other orderings such as descending order of item caps or of occurrence counts. An interesting observation is that, if we were to store items in descending order of occurrence counts, then *the number of nodes in the resulting PUF-tree would be the same as that of the FP-tree.*

Note that the sum of all item cap values (i.e., ***total item cap***) for all nodes of an item in a PUF-tree (which is computed when inserting each transaction) is maintained in the *I-list*. In other words, an item x_r in the *I-list* of the PUF-tree maintains $expSup^{Cap}(X)$ for $X = \{x_1, x_2, \ldots, x_r\}$.

Lemma 2. For a k-itemset $X = \{x_s, \ldots, x_r\}$ (where $k > 1$) in a DB, if $expSup^{Cap}(X) < minsup$, then any k''-itemset $Z = \{x_t, \ldots, x_r\}$ (where $k'' > k$) in the DB that contains X (i.e., $Z \supset X$) cannot be frequent.

Proof. Let X & Z be two itemsets such that (i) $X \subset Z$ and (ii) they share the same suffix item x_r. Following Lemma 1, if $expSup^{Cap}(X) < minsup$, then $expSup^{Cap}(Z) \leq expSup^{Cap}(X) < minsup$. As $expSup^{Cap}(Z)$ serves as an upper bound to $expSup(Z)$, we get $expSup(Z) \leq expSup^{Cap}(Z)$. Hence, if $expSup^{Cap}(X) < minsup$, then Z cannot be frequent because $expSup(Z) \leq expSup^{Cap}(Z) < minsup$. □

The above lemma allows us to prune the constructed PUF-tree further by removing any item having a total item cap (in the *I-list*) less than *minsup*. (The horizontal node traversal pointers allow us to visit such nodes in the PUF-tree in an efficient manner.) Hence, we can remove item d from the PUF-tree in Fig. 2(c) because $expSup^{Cap}(d) < minsup$. This results in a more compact PUF-tree, as shown in Fig. 2(d). This tree-pruning technique can save the mining time as it skips those items.

Let $F(t_j)$ be the set of frequent items in transaction t_j. Based on the aforementioned tree construction mechanism, the item cap in a node x in a PUF-tree maintains the sum of item caps (i.e., total item cap) of an item x for all transactions that pass through or end at x. Because common prefixes are shared, the PUF-tree becomes more compact and avoids having siblings containing the same item but having different existential probability values. However, as the item caps of different items in a transaction can be different, the item cap of a node in a PUF-tree does not necessarily need to be greater than or equal to that of all of its child nodes.

It is interesting to note that, although item caps of a parent and child(ren) are not related, a PUF-tree can be a highly compact tree structure. The number of tree nodes in a PUF-tree (i) can be the same to that of an FP-tree [7] (when the PUF-tree is constructed using the frequency-descending order of items) and (ii) is bounded above by $\sum_{t_j \in \text{ DB}} |F(t_j)|$. In addition, the complete set of mining results can be generated because a PUF-tree contains $F(t_j)$ for all transactions and it stores the total item cap for a node. Mining based on this item cap value ensures that no frequent k-itemset ($k > 1$) will be missed.

Furthermore, recall that the expected support of $X = \{x_1, \ldots, x_k\}$ is computed by summing the products of the existential probability value of x_k with those of all items in the proper prefix of X over all n transactions in the DB, i.e., $\mathit{expSup}(X) = \sum_{j=1}^{n}(P(x_k, t_j) \times (\prod_{i=1}^{k-1} P(x_i, t_j)))$. Hence, the item cap for X computed based on the existential probability value of x_k and the highest existential probability value in its prefix provides a *tighter* upper bound (than that based on highest existential probability value in transactions containing X) because the former involves only the existential probability values of items that are in X whereas the latter may involve existential probability values of items that are not even in X.

4 Our PUF-Growth Algorithm for Mining Frequent Patterns from PUF-Trees

Here, we propose a pattern-growth mining algorithm called **PUF-growth**, which mines frequent patterns from our PUF-tree structure. Recall that the construction of a PUF-tree is similar to that of the construction of an FP-tree, except that item caps (instead of occurrence frequencies) are stored. Thus, the basic operation in PUF-growth for mining frequent patterns is to construct a projected DB for each potential frequent pattern and recursively mine its potential frequent extensions.

Once an item x is found to be potentially frequent, the existential probability of x must contribute to the expected support computation for every pattern X constructed from $\{x\}$-projected DB (denoted as DB_x). Hence, the cap of expected support of x is guaranteed to be the upper bound of the expected support of the pattern. This implies that the complete set of patterns with suffix x can be mined based on the partial downward closure property stated in Lemma 1. Note that $\mathit{expSup}^{Cap}(X)$ is the upper bound of $\mathit{expSup}(X)$, and it satisfies the partial downward closure property. So, we can directly proceed to generate all potential frequent patterns from the PUF-tree based on the following corollary.

Corollary 1. Let (i) X be a k-itemset (where $k > 1$) with $\mathit{expSup}^{Cap}(X) \geq \mathit{minsup}$ in the DB and (ii) Y be an itemset in the X-projected DB (denoted as DB_X). Then, $\mathit{expSup}^{Cap}(Y \cup X)$ in the DB $\geq \mathit{minsup}$ if and only if $\mathit{expSup}^{Cap}(Y)$ in all the transactions in $DB_X \geq \mathit{minsup}$.

Proof. Let (i) X be a k-itemset (where $k > 1$) with $\mathit{expSup}^{Cap}(X) \geq \mathit{minsup}$ in the DB and (ii) Y be an itemset in the X-projected DB (i.e., $Y \in DB_X$).

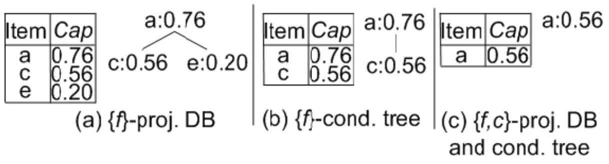

Fig. 3. Our PUF-growth mines frequent patterns from the PUF-tree in Fig. 2(d)

Then, due to the mining process (especially, the construction of projected DBs) in the PUF-growth algorithm, itemset $(Y \cup X)$ in the DB shares the same suffix (i.e., X) as itemset Y in DB_X. Moreover, due to the definition of projected DBs, the transactions that contain $(Y \cup X)$ in the DB are identical to those transactions that contain Y in DB_X. Hence, $expSup^{Cap}(Y \cup X)$ in the DB equals to $expSup^{Cap}(Y)$ in DB_X. Consequently, if $expSup^{Cap}(Y \cup X) \geq minsup$ in the DB, then $expSup^{Cap}(Y) \geq minsup$ in DB_X, and vice versa. □

Based on Lemma 1 and Corollary 1, we apply the PUF-growth algorithm to our PUF-tree for generating only those k-itemsets (where $k > 1$) with caps of expected support $\geq$ *minsup*. Similar to UFP-growth, this mining process may also lead to some false positives in the resulting set of frequent patterns at the end of the second DB scan, and all these false positives will be filtered out with the third DB scan. Hence, our PUF-growth is guaranteed to return the *exact* set of frequent patterns (i.e., *all* and *only those* frequent patterns with *neither* false positives *nor* false negatives).

Example 5. The PUF-growth algorithm starts mining from the bottom of the *I-list* (i.e., item f with $expSup^{Cap}(f) = 0.76$). The $\{f\}$-projected DB, as shown in Fig. 3(a), is constructed by accumulating tree paths $\langle a{:}0.56,\ c{:}0.56\rangle$ and $\langle a{:}0.20,\ e{:}0.20\rangle$. Note that the total item cap of f is the sum of item caps in each respective path. When projecting these paths, PUF-growth also calculates the cap of the expected support of each item in the projected DB (as shown in the *I-list* in the figure). Based on Lemma 2, PUF-growth then removes infrequent item e from the $\{f\}$-projected DB—because $expSup^{Cap}(e) = 0.20 < minsup$—and results in the $\{f\}$-conditional tree as shown in Fig. 3(b).

This $\{f\}$-conditional tree is then used to generate (i) all 2-itemsets containing item f and (ii) their further extensions by recursively constructing projected DBs from them. Consequently, pattern $\{f, c\}$:0.56 is generated first, and the $\{f, c\}$-projected DB is then constructed as shown in Fig. 3(c). Pattern $\{f, c, a\}$:0.56 is generated from the $\{f, c\}$-conditional tree. Pattern $\{f, a\}$:0.76 is then generated, which terminates the mining process for $\{f\}$ because no further extension of the projected DB can be generated. Patterns containing items such as e, c and a can then be mined in a similar fashion. The complete set of patterns generated by PUF-growth includes $\{f, c\}$:0.56, $\{f, c, a\}$:0.56, $\{f, a\}$:0.76, $\{e, a\}$:1.02 and $\{c, a\}$:0.59. □

As shown in Example 5, PUF-growth finds a complete set of patterns from a PUF-tree without any false negatives.

5 Experimental Results

We compared the performances of our PUF-growth algorithm with existing algorithms (e.g., UF-growth [13], UFP-growth [2] and UH-Mine [2]) on both real and synthetic datasets. The synthetic datasets, which are generally sparse, are generated within a domain of 1000 items by the data generator developed at IBM Almaden Research Center [1]. We also considered several real datasets such as mushroom, retail and connect4. We assigned a (randomly generated) existential probability value from the range (0,1] to each item in every transaction in the dataset. The name of each dataset indicates some characteristics of the dataset. For example, the dataset u100k10L_10_100 contains 100K transactions with average transaction length of 10, and each item in a transaction is associated with an existential probability value that lies within a range of [10%, 100%]. Due to space constraints, we present here the results on some of the above datasets.

All programs were written in C and run with UNIX on a quad-core processor with 1.3GHz. Unless otherwise specified, runtime includes CPU and I/Os for *I-list* construction, tree construction, mining, and false-positive removal (of PUF-growth). The results shown in this section are based on the average of multiple runs for each case. In all experiments, *minsup* was expressed in terms of the percentage of DB size, and the PUF-trees were constructed using the descending order of expected support of items.

5.1 Compactness of PUF-Trees

The UF-tree, UFP-tree and PUF-tree all arranged items in the same order (e.g., descending order of expected support). Hence, depending on the clustering parameter, the number of nodes of a UFP-tree (in its best case) could be similar to that of a PUF-tree. Note that the size of UFP-tree was larger than that of PUF-tree because the UFP-tree stored extra cluster information in nodes. The size of UF-tree was larger than that of the other three because the UF-tree may contain multiple nodes for the same item (under the same parent). So, in the first experiment, we compared the compactness of our PUF-trees with the UF-tree[1].

The node counts between PUF-trees and UF-trees, as presented in Table 2, show that PUF-trees were more compact than UF-trees for both sparse and dense datasets and for high and low *minsup* thresholds. The ratio $\frac{\#\text{nodes in PUF-tree}}{\#\text{nodes in UF-tree}}$ represents the gain of PUF-trees over UF-trees (e.g., for mushroom_50_60, the PUF-tree contained 8108 nodes, which was only 6.68% of the 121205 nodes in the UF-tree). The gain of PUF-trees was much promising in dense datasets (e.g., mushroom_50_60) than sparse datasets (e.g., u100k10L_10_100) because the PUF-tree is more likely to share paths for common prefixes in *dense* datasets. In contrast, the UF-tree contains a distinct tree path for each distinct ⟨item, existential probability⟩ pair, and thus *not* as compact as the PUF-tree.

[1] Because UH-Mine stores all transactions in the UH-struct, it is usually less compact than the PUF-tree. So, we avoid showing the comparison with UH-Mine.

Table 2. Compactness of PUF-trees

Dataset	*minsup*	$\frac{\#\text{PUF-tree nodes}}{\#\text{UF-tree nodes [13]}}$	*minsup*	$\frac{\#\text{PUF-tree nodes}}{\#\text{UF-tree nodes [13]}}$
retail_50_60	0.2	$\frac{159{,}504}{672{,}670} \approx 23.71\%$	2	$\frac{208}{17{,}708} \approx 1.17\%$
mushroom_50_60	0.1	$\frac{8{,}108}{121{,}205} \approx 6.68\%$	7	$\frac{2{,}982}{101{,}985} \approx 2.92\%$
u100k10L_50_60	0.05	$\frac{90{,}069}{807{,}442} \approx 11.15\%$	0.1	$\frac{90{,}007}{798{,}073} \approx 11.28\%$
u100k10L_10_100	0.05	$\frac{90{,}069}{881{,}010} \approx 10.22\%$	0.1	$\frac{90{,}013}{872{,}062} \approx 10.32\%$

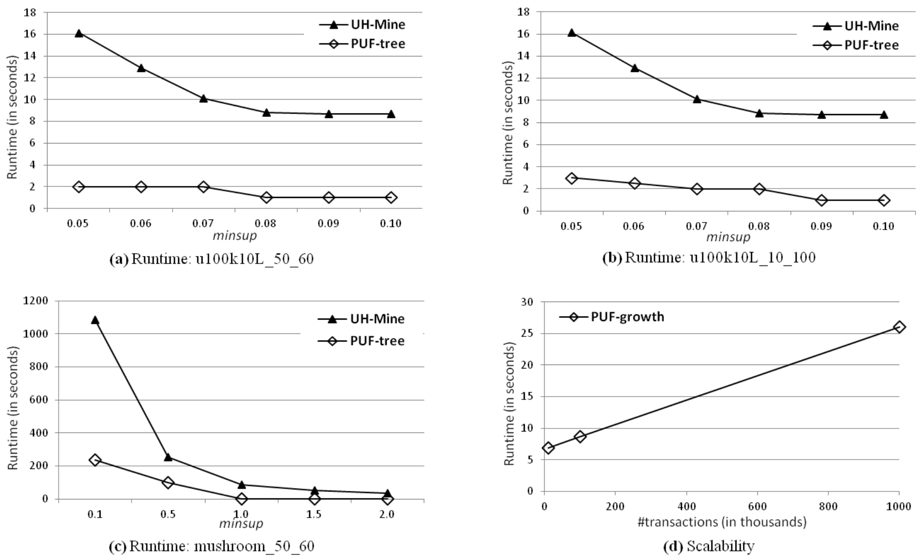

Fig. 4. Experimental results

5.2 Runtime

Recall that UH-Mine was shown to outperform the UFP-growth [2,15]. So, we also compared our PUF-growth with UH-Mine. Figs. 4(a)–(c) show that PUF-growth took shorter runtime than UH-Mine for datasets u100k10L_50_60, u100k10L_10_100 and mushroom_50_60. The primary reason is that, even though the UH-Mine finds the exact set of frequent patterns when mining an extension of X, it may suffer from the high computation cost of calculating the expected support of X on-the-fly for all transactions containing X. Such computation may become more costly when mining a large number of patterns (e.g., in mushroom_50_60) and long patterns (e.g., in u100k10L_50_60, u100k10L_10_100, retail_50_60).

5.3 Number of False Positives

In practice, although both UFP-tree and PUF-trees are compact, their corresponding algorithms generate some false positives. Hence, their overall

Table 3. Comparison on false positives (in terms of % of total #patterns)

Dataset	***minsup***	**UFP-growth [2]**	**PUF-growth**
u10k5L_80_90	0.1	61.20%	22.55%
u100k10L_50_60	0.07	89.32%	25.99%

performances depend on the number of false positives generated. In this experiment, we measured the number of false positives generated by UFP-growth and PUF-growth. Due to space constraints, we present results (in percentage) using one *minsup* value for each of the two datasets (i.e., u10k5L_80_90 and u100k10L_50_60) in Table 3. In general, PUF-growth was observed to remarkably reduce the number of false positives when compared with UFP-growth. The primary reason of this improvement is that upper bounds of expected support of patterns in clusters are *not* as tight as the upper bounds provided by PUF-growth. In a UFP-tree, if a parent has several children, then each child will use higher cluster values in the parent to generate the total expected support. If the total number of existential probability values of that child is still lower than that of the parent's highest cluster value, then the expected support of the path with this parent and child will be high. This results in more false positives in long run.

5.4 Scalability

To test the scalability of PUF-growth, we applied the algorithm to mine frequent patterns from datasets with increasing size. The experimental result presented in Fig. 4(d) indicates that our PUF-growth algorithm (i) is scalable with respect to the number of transactions and (ii) can mine large volumes of uncertain data within a reasonable amount of time.

The above experimental results show that our PUF-growth algorithm effectively mines frequent patterns from uncertain data irrespective of distribution of existential probability values (whether most of them have low or high values, whether they are distributed into a narrow or wide range of values).

6 Conclusions

In this paper, we proposed the **PUF-tree structure** for capturing important information of uncertain data. In addition, we presented the **PUF-growth algorithm** for mining frequent patterns. The algorithm uses the PUF-tree to obtain upper bounds to the expected support of frequent patterns (i.e., *item caps*, which are computed based on the highest existential probability of an item in the prefix); it guarantees to find *all* frequent patterns (with *no* false negatives). Experimental results show the effectiveness of our PUF-growth algorithm in mining frequent patterns from our PUF-tree structure.

Acknowledgements. This project is partially supported by NSERC (Canada) and University of Manitoba.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: VLDB 1994, pp. 487–499 (1994)
2. Aggarwal, C.C., Li, Y., Wang, J., Wang, J.: Frequent pattern mining with uncertain data. In: ACM KDD 2009, pp. 29–37 (2009)
3. Bernecker, T., Kriegel, H.-P., Renz, M., Verhein, F., Zuefle, A.: Probabilistic frequent itemset mining in uncertain databases. In: ACM KDD 2009, pp. 119–127 (2009)
4. Calders, T., Garboni, C., Goethals, B.: Approximation of frequentness probability of itemsets in uncertain data. In: IEEE ICDM 2010, pp. 749–754 (2010)
5. Calders, T., Garboni, C., Goethals, B.: Efficient pattern mining of uncertain data with sampling. In: Zaki, M.J., Yu, J.X., Ravindran, B., Pudi, V. (eds.) PAKDD 2010, Part I. LNCS (LNAI), vol. 6118, pp. 480–487. Springer, Heidelberg (2010)
6. Chui, C.-K., Kao, B., Hung, E.: Mining frequent itemsets from uncertain data. In: Zhou, Z.-H., Li, H., Yang, Q. (eds.) PAKDD 2007. LNCS (LNAI), vol. 4426, pp. 47–58. Springer, Heidelberg (2007)
7. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: ACM SIGMOD 2000, pp. 1–12 (2000)
8. Lakshmanan, L.V.S., Leung, C.K.-S., Ng, R.T.: Efficient dynamic mining of constrained frequent sets. ACM TODS 28(4), 337–389 (2003)
9. Leung, C.K.-S.: Mining uncertain data. WIREs Data Mining and Knowledge Discovery 1(4), 316–329 (2011)
10. Leung, C.K.-S., Hao, B.: Mining of frequent itemsets from streams of uncertain data. In: IEEE ICDE 2009, pp. 1663–1670 (2009)
11. Leung, C.K.-S., Irani, P.P., Carmichael, C.L.: FIsViz: a frequent itemset visualizer. In: Washio, T., Suzuki, E., Ting, K.M., Inokuchi, A. (eds.) PAKDD 2008. LNCS (LNAI), vol. 5012, pp. 644–652. Springer, Heidelberg (2008)
12. Leung, C.K.-S., Jiang, F.: RadialViz: an orientation-free frequent pattern visualizer. In: Tan, P.-N., Chawla, S., Ho, C.K., Bailey, J. (eds.) PAKDD 2012, Part II. LNCS (LNAI), vol. 7302, pp. 322–334. Springer, Heidelberg (2012)
13. Leung, C.K.-S., Mateo, M.A.F., Brajczuk, D.A.: A tree-based approach for frequent pattern mining from uncertain data. In: Washio, T., Suzuki, E., Ting, K.M., Inokuchi, A. (eds.) PAKDD 2008. LNCS (LNAI), vol. 5012, pp. 653–661. Springer, Heidelberg (2008)
14. Leung, C.K.-S., Tanbeer, S.K.: Fast tree-based mining of frequent itemsets from uncertain data. In: Lee, S.-g., Peng, Z., Zhou, X., Moon, Y.-S., Unland, R., Yoo, J. (eds.) DASFAA 2012, Part I. LNCS, vol. 7238, pp. 272–287. Springer, Heidelberg (2012)
15. Tong, Y., Chen, L., Cheng, Y., Yu, P.S.: Mining frequent itemsets over uncertain databases. PVLDB 5(11), 1650–1661 (2012)
16. Zhang, Q., Li, F., Yi, K.: Finding frequent items in probabilistic data. In: ACM SIGMOD 2008, pp. 819–832 (2008)