The Hong Kong Polytechnic University

Department of Computing

Iterative Uncertain Frequent Pattern Mining with Trees

WANG Shu

A thesis submitted in partial fulfilment of the requirements

for the degree of Master of Philosophy

August 2011

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

WANG Shu

Abstract

Many frequent-pattern mining algorithms were designed to handle precise data, such as the FP-tree structure and the FP-growth algorithm. In data mining research, attention has been turned to mining frequent patterns in uncertain data recently. A common way to represent the uncertainty of a data item in transactional databases is to associate it with an existential probability. In this thesis, we propose two solutions for uncertain frequent pattern mining.

One solution is a novel uncertain-frequent-pattern discovery structure, the mUF-tree, for storing summarized and uncertain information about frequent patterns. With the mUF-tree, the UF-Evolve algorithm can utilize the shuffling and merging techniques to generate iterative versions of the tree. Its main purpose is to discover new uncertain frequent patterns from these iterative versions.

The other solution is the mUF-trie structure and the UF-Prune algorithm. In the mUF-trie, the uncertain information about frequent patterns is summarized in the lexicographic order, which facilitates mining uncertain frequent patterns separately for each item. With the mUF-trie, the UF-Prune algorithm can continuously generate a sub-trie for each item, utilize the shuffling and merging techniques to generate iterative versions of the sub-trie, and prune away the processed items in the mUF-trie. As in the mUF-tree, the new structure can support the discovery of new uncertain frequent patterns relating to each item from iterative versions of its sub-trie.

Our preliminary performance study shows that the UF-Evolve and UF-Prune algorithms are efficient and scalable for mining additional uncertain frequent patterns. We have also proposed an application and some extended work of the two solutions. The uncertain frequent pattern mining for rural systems can find out special patterns relating to productivity and sustainability to improve profitability or environmental gain for valuable crops, and the extensions are related to incremental uncertain frequent pattern mining with the mUF-tree and mUF-trie.

Publications arising from the thesis

1. Wang, S., and Ng, V. *UF-tree: Uncertain Frequent Pattern Mining for Rural Systems.* Proceedings of the Knowledge Discovery for Rural Systems 2010, PAKDD 2010, Hyderabad, India, pp. 115-128, 2010.
2. Wang, S., and Ng, V. *UF-Evolve: Uncertain Frequent Pattern Mining.* ICEIS 2011 - 13th International Conference on Enterprise Information Systems, Volume 1, pp. 74-84, Beijing, China, 8 - 11 June, 2011.
   (Best Student Paper Award)
3. Wang, S., and Ng, V. *UF-Evolve: Uncertain Frequent Pattern Mining.* Selected Papers submitted for Springer-Verlag LNBIP Series Book, Lecture Notes, Business Information Processing (LNBIP) published by Springer-Verlag.

Acknowledgements

I would like to express my sincere thanks to my Chief Supervisor, Dr. Vincent Ng, who spent much time on discussing the research work with me. He gave me constructive suggestions and valuable guidance through the whole period of research study. His professional advice, enthusiastic brainstorming and useful critiques helped me to develop new ideas and solve problems during the research study.

My special thank is extended to the BoE Chair and external examiners for kindly reading and evaluating my thesis.

I would like to thank Department of Computing, who provided support in many aspects during my research study.

Last but not least, I want to thank my parents and friends, who provided me encouragements and supports in tangible and intangible ways.

WANG Shu

Table of contents

List of figures, tables and abbreviations

Chapter 1 : Introduction

Data uncertainty is often found in real-world applications because of measurement inaccuracy, sampling discrepancy, outdated data sources, or other errors. One type of data uncertainty is existential uncertainty. In existential uncertainty, it is uncertain about the presence or absence of some items or events. For example, we may highly suspect, while cannot guarantee, that a patient suffers from an illness based on a few symptoms. The uncertainty of such suspicion can be expressed in terms of existential probability. If R_i represents a patient record, then each item within R_i represents an illness and is associated with an existential probability expressing the likelihood of the patient suffering from that illness. As an example, a patient can have an 80% likelihood of suffering from fever, and a 60% likelihood of suffering from H1N1.

Many frequent-pattern mining algorithms were designed to handle precise data. Recently, attention has been turned to mining frequent patterns in uncertain data. Han et al. [16] proposed the FP-tree structure, which is an extended prefix-tree structure for storing compressed, crucial information about frequent patterns, and developed an efficient FP-growth algorithm. In our work, we extend the FP-tree for mining uncertain data. Our key contributions are (i) the development of the mUF-tree structure to summarize the content of records consisting of uncertain data, (ii) the idea of shuffling and merging nodes of mUF-tree, whose difference in existential probabilities is small, to derive more uncertain frequent patterns by the UF-Evolve algorithm, (iii) the development of the mUF-trie structure to summarize the content of records consisting of uncertain data in the lexicographic order, and (iv) the idea of mining uncertain frequent patterns for each item with better efficiency by the UF-Prune algorithm.

Some existing models compute expected supports and mine frequent patterns in the form of an itemset with the expected support. This calculation incurs much information loss since the mined frequent patterns do not contain any probability information. In our model, we aim to keep the probability information in our problem settings and definitions, and display the probability information in the mined frequent patterns, which are the significance and also the advantage over the others.

The rest of the thesis is organized as follows. Chapter 2 gives a literature review and Chapter 3 describes the problem statement and our approaches. Chapter 4 introduces the mUF-tree, the UF-Evolve algorithm and other supporting algorithms to discover uncertain frequent patterns iteratively. Chapter 5 describes the mUF-trie and the UF-Prune algorithm. Chapter 6 presents our performance study and its results. Next, Chapter 7 describes an application of our algorithm for rural data and the approach for incremental mining with uncertain data. Finally, Chapter 8 concludes our work and suggests some future related research.

Chapter 2 : Literature Review

2.1 Uncertain Data Representations

Antova et al. [5] proposed U-relations, a succinct and purely relational representation system for uncertain databases. U-relations support attribute-level uncertainty using vertical partitioning. When considering positive relational algebra extended by an operation for computing possible answers, a query on the logical level can be translated into, and evaluated as, a single relational algebra query on the U-relational representation. Positive relational algebra queries are evaluated purely relationally on U-relations, and this query evaluation approach takes full advantage of query evaluation and optimization techniques on vertical partitions.

Antova et al. [6] proposed I-SQL, which is an analog to SQL for the case of incomplete information. They also proposed World-set Algebra, which is a language that supports the contemplation on alternatives and can map from a complete database to an incomplete one comprising several possible worlds. World-set Algebra formalizes a clean fragment of I-SQL, which is similar as relational algebra formalizes SQL.

Hunter et al. [18] extended a logic-based framework to modeling and merging uncertain information that is defined at different levels of granularity of XML textentries, and to modeling and reasoning with XML documents that contain semantically heterogeneous uncertain information on more complex elements in XML sub-trees. One advantage of the uncertainty XML model is that it focuses on multiple XML datasets and provides a set of means to merge options with uncertainty from different sources on textentries and sub-trees.

Keijzer [19] proposed to use probabilistic XML in data integration to facilitate unattended integration. Instead of asking a user, the system should decide itself if elements refer to the same real-world object. The integration result with conflicts is stored, rather than resolving conflicts at integration time. At query time, the conflicts are resolved and the user's feedback to query results can be used to reduce the uncertainty in the database.

2.2 Data Mining on Precise Data

In 1994, Agrawal et al. [4] developed two algorithms, Apriori and AprioriTid, for solving the problem of discovering all significant association rules between items in a large database of sales transactions. Since then, there are many interests to improve the performance of association mining. Malik et al. [27] proposed HDO (Hamming-Distance-based greedy transaction reordering scheme), and aHDO, a linear-time approximation to HDO. These two data mining algorithms use trie and bitmap-based representations to optimize the support (i.e. frequency) counting performance.

Han et al. [16] proposed the FP-tree structure and the FP-growth algorithm for efficiently mining frequent patterns without generation of candidate itemsets for precise data. It consists of two phases. The first phase focuses on constructing the FP-tree from the database, and the second phase focuses on applying FP-growth to derive frequent patterns from the FP-tree. Each node in the FP-tree consists of three attributes, item-name, count and node-link. In the FP-tree, each entry in the associated header table consists of two fields: (1) item-name, and (2) head of node-links (a pointer pointing to the first node in the FP-tree carrying the item-name). Suppose there is a precise transaction database as shown in Table 1, and the minimum support threshold is 3. Then, the FP-tree together with the associated node-links is shown in Figure 1.

TID	Items
1	f, a, c, d, g, i, m, p
2	a, b, c, f, l, m, o
3	b, f, h, j, o
4	b, c, k, s, p
5	a, f, c, e, l, p, m, n

Table 1. A precise transaction database

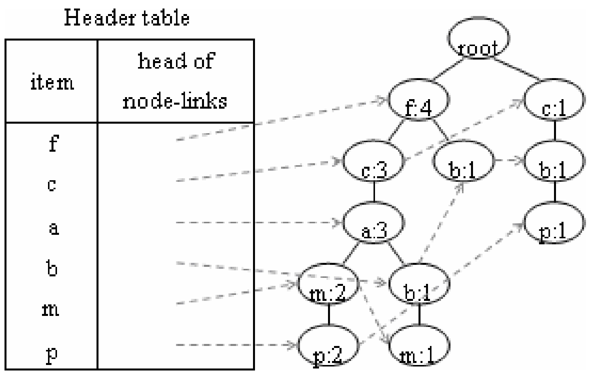


Figure 1. The FP-tree for data in Table 1

The first advantage of using the FP-tree is related to its size. It is highly compact and is usually substantially smaller than the original database. Another advantage is that the FP-growth algorithm applies a pattern growth method and never generates any nonexistent candidate itemsets because the itemset in each transaction is always included in the corresponding path of the FP-tree. The third advantage is that the FP-growth algorithm applies a partitioning-based divide-and-conquer method and recursively constructs conditional FP-trees with conditional pattern-bases.

The FP-tree structure and the FP-growth algorithm are so popular that they have been adopted or extended many times [3, 8, 21, 29, 33]. For example, Borgelt [8] described a C implementation of the FP-growth algorithm, which contains two variants of the core operation of computing a projection of an FP-tree, and Krakovetskiy [21] proposed the conception of strong itemsets, which solves the problem of huge number of candidate generation in mining association rules.

2.3 Data Mining on Uncertain Data

Chau et al. [10] considered data uncertainty when data mining is performed on uncertain data, in order to obtain high quality data mining results. They proposed a framework for possible research directions in the area of uncertain data mining. They also proposed the UK-means clustering algorithm, which modifies the traditional K-means algorithm to handle data uncertainty in data mining.

Chui et al. [14] proposed the U-Apriori algorithm, which processes transactions whose items are associated with existential probabilities. The advantage of the existential uncertain data model is that it allows more information to be captured by the dataset. The disadvantage of retaining such information is that the size of the dataset would be much larger. They also introduced a trimming strategy to reduce the number of candidates that need to be counted based on the Apriori approach. The Apriori heuristic achieves good performance gained by (possibly significantly) reducing the size of candidate sets. However, in situations with a large number of frequent patterns, long patterns, or low minimum support thresholds, an Apriori-like algorithm may suffer from the two nontrivial issues. It is costly to handle a huge number of candidate sets; and it is tedious to repeatedly scan the database and check a large set of candidates by pattern matching, which is especially true for mining long patterns.

Leung et al. [22, 23] proposed the UF-tree structure, a different tree structure than the FP-tree for capturing the content of transactions consisting of uncertain data, and the UF-growth algorithm, a mining algorithm for finding frequent patterns from the UF-tree. Each node in the UF-tree stores item, expected support and occurrence (i.e. the number of transactions containing such an item). UF-growth computes expected supports of itemsets and finds frequent patterns from the UF-tree. The expected support of an itemset in a transaction is the expected probability (over all "possible worlds") of coexistence of all the items in the itemset. The expected support of an itemset in a database is the sum of the expected probabilities of the itemset over all transactions. This calculation incurs much information loss.

Sun et al. [30] proposed two algorithms to discover frequent patterns and association rules from probabilistic data under the Possible World Semantics in bottom-up and top-down manners, and extended these two algorithms to discover maximal frequent patterns. The tuple-uncertainty model adopted in [30] is used for probabilistic databases. Each tuple is associated with a measuring equipment or sensor, and the probability for each tuple is determined by sensor measurement errors, as well as the uncertainty caused by the information extraction in the system. Therefore, each tuple carries an existential probability attribute, which denotes the confidence that the tuple exists. This is a simple database design. Yet this semantics is not adopted in our research, since we have different semantics which is described next.

There are different models for uncertain data. In one model adopted in [14, 22, 23], each item is associated with an existential probability. Another model in [30] is to associate an existential probability to each transaction instead of individual items in a transaction. In our research, we use the first model, in which each item is associated with an existential probability. Since for the applications and data we used, such as patient records and rural data, each item is associated with a measuring equipment or sensor, the probabilities of different items in the same record are independent. But we aim to avoid the information loss incurred by computing expected supports, and display the probability information in the mined frequent patterns. Therefore, in Chapter 6, we do not compare our methods with the existing uncertain data mining methods.

2.4 Incremental Data Mining on Precise Data

Adnan et al. [2] proposed an incremental algorithm that can update the FP-tree incrementally without scanning the old database and with a minimal scanning of the incremental database. First, they extended the FP-tree to include every attribute that occurs at least once in the database. This facilitates mining frequent patterns with different support thresholds without constructing several FP-trees to satisfy the purpose. Second, they extended the FP-tree to reflect updates to the corresponding database by scanning only the updated portion. This reduces execution time in general.

Chang et al. [9] proposed the NFUP (New Fast UPdate) algorithm for incrementally mining association rules from large transaction databases. NFUP uses the information available from a following partition to avoid the rescanning of the original database, and only requires the scanning of the incremental database. For some newly generated frequent itemsets in the incremental database, NFUP does not need to rescan the original database, but accumulates their occurrence counts and deletes obviously infrequent itemsets. Thus, NFUP can determine new frequent itemsets at the latest time intervals.

Cheung et al. [12] proposed the FUP_2 algorithm, which is a general incremental updating technique for maintaining the association rules discovered in a database in the cases of insertion, deletion and modification of transactions in the database. FUP_2 can update the discovered association rules when new transactions are added to the database, and old transactions are removed from it. FUP_2 makes use of the previous mining results to reduce the amount of work that has to be done to discover the association rules in the updated database.

Cheung et al. [13] proposed the CATS Tree structure, which extends the idea of FP-tree to improve storage compression and allow frequent pattern mining without generation of candidate itemsets. There are some advantages of CATS Tree. The first advantage is that once a CATS Tree is built, frequent pattern mining with different supports can be performed without rebuilding the tree structure. The second advantage is that CATS Tree allows single pass frequent pattern mining over the database and thus can mine transaction streams. The third advantage is that CATS Tree allows insertion and deletion of transactions or even a single transaction at any time. This makes CATS Tree suitable for real time transactional frequent pattern mining with modifications.

Ezeife et al. [15] proposed two algorithms for incrementally maintaining association rules in the updated database. One is the DB-tree algorithm, which stores all the database information in an FP-tree structure and requires no rescan of the original database for all update cases. The other is the PotFP-tree (Potential Frequent Pattern tree) algorithm, which uses a prediction of future possible frequent itemsets to reduce the number of times the original database needs to be scanned when previous small itemsets become large after database update. The two algorithms are based on a generalized FP-tree structure that store more items on the tree rather than only those that are frequent, and thus reduce the required number of database scans.

Hong et al. [17] proposed the FUFP-tree (Fast Updated FP-tree) structure, which modifies the FP-tree construction algorithm for handling new transactions, and makes the tree update process easier. They also proposed the incremental FUFP-tree maintenance structure and algorithm, which handle new transaction insertion in data mining.

Chapter 3 : Iterative Mining of Uncertain Data

Given an uncertain database and a minimum support threshold, we propose to solve the problem of discovering a possible set of uncertain frequent patterns in two phases.

- In the first phase, a tree is constructed for storing summarized and uncertain information about a given database.
- In the second phase, algorithms are applied for iteratively discovering new uncertain frequent patterns from the tree with shuffling and merging.

3.1 Problem Statement

In this section, we state some definitions before introducing our target problem. First, we define what an uncertain record and an uncertain database are.

Definition 1. Uncertain Record (R). Let I = $\{a_1, a_2, \ldots, a_k\}$ be a set of items. An uncertain record R = $\{(a_1{:}p_1), (a_2{:}p_2), \ldots, (a_m{:}p_m)\}$, where ($a_i \in$ I) $\wedge$ (each a_i is unique with p_i as the probability indicating its existence).

Definition 2. Uncertain Database (UDB). An uncertain database UDB consists of multiple uncertain records, i.e. UDB = $<R_1, R_2, \ldots, R_n>$.

An example of an uncertain database is shown in Table 2.

Record ID	**Item:probability pairs**
1	(a:0.2), (f:0.8), (g:0.1), (d:0.3), (c:0.2)
2	(a:0.2), (d:0.3), (c:0.2), (f:0.8)
3	(b:0.3), (f:0.8), (a:0.2)
4	(a:0.4), (c:0.7), (d:0.6), (f:0.8)
5	(f:0.8)
6	(a:0.7), (c:0.4), (b:0.5)
7	(a:0.7)
8	(f:0.8), (a:0.4), (c:0.7), (d:0.6)
9	(b:0.5), (a:0.7), (c:0.4)
10	(d:0.4), (e:0.5), (f:1.0), (a:0.1)
11	(e:0.6), (c:0.3), (d:0.5), (a:0.1), (f:1.0)
12	(a:0.1), (f:1.0), (c:0.3), (d:0.5)
13	(a:0.1), (f:1.0)
14	(f:1.0)
15	(d:0.4), (c:0.4), (b:0.2)
16	(b:0.2)
17	(b:0.2), (f:0.6)
18	(b:0.2)

Table 2. An uncertain database.

In an uncertain database, it is often that an item with a probability appears in a record while the same item with a different probability appears in another record. For example, in Table 2, (d:0.4) appears in the 10th record while (d:0.5) appears in the 11th record. Given a predefined minimum support threshold, it can be observed that an item with a specific probability may not have sufficient support (i.e. the number of records containing it in UDB) to be a frequent pattern.

In order to have more and longer frequent patterns, one can consider merging the same items with only small differences in their respective probabilities for satisfying the support requirement. For instance, we can consider (d:0.4) and (d:0.5) as a group with a probability range [0.4-0.5]. Hence, for a group of item:probability pairs in which all items are the same, a [lowerbound-upperbound] can be used to represent the spread of probabilities for the item. Once the item:probability pairs are grouped, more frequent patterns would be found. With this idea, we now define maximum merging threshold and uncertain frequent pattern.

Definition 3. Maximum Merging Threshold (γ). A maximum merging threshold γ is used to determine whether two item:probability pairs can be merged. Suppose an item a with probability (a:p_1) appears in a record while the same item with another probability (a:p_2) appears in another record. If abs($p_1 - p_2$) $\leq \gamma$, then (a:p_1) and (a:p_2) can be merged and represented as (a:[l-u]), where (l = min(p_1, p_2)) $\wedge$ (u = max(p_1, p_2)). Here, (a:[l-u]) means it is of item a with l as its lowerbound and u as its upperbound of their respective existential probabilities.

The maximum merging threshold can be defined by users. Here, we define a uniform maximum merging threshold for all items, and for all probability values of the same item. In fact, a different user can define different maximum merging thresholds for different items, and for different probability intervals of the same item to suit particular applications. There is only a small change in the proposed algorithms.

Definition 4. Uncertain Frequent Pattern (UFP). An uncertain frequent pattern UFP is represented as $(a_1:[l_1\text{-}u_1])(a_2:[l_2\text{-}u_2])\ldots(a_k:[l_k\text{-}u_k])$:s, where (s is the support for UFP) $\wedge$ (s $\geq$ a predefined minimum support threshold). A record R contains an UFP if for each $(a_i:[l_i\text{-}u_i]) \in$ UFP, $\exists$ $(a_j:p_j) \in$ R, such that $(a_i = a_j) \wedge (l_i \leq p_j \leq u_i)$.

3.2 Our Approaches

Given an uncertain database, a minimum support threshold and a maximum merging threshold, we are interested in mining a possible set of uncertain frequent patterns. We propose two methods. The first method is the mUF-tree structure with the UF-Evolve algorithm. It solves the problem in two phases.

- In the first phase, an mUF-tree is constructed for storing summarized and uncertain information about frequent patterns.
- In the second phase, the UF-Evolve algorithm, which utilizes the shuffling and merging techniques to generate iterative versions of the mUF-tree, is applied for discovering new uncertain frequent patterns. It can either shuffle and merge the most suitable pair of paths each time, or shuffle and merge many pairs of paths at once.

However, UF-Evolve shuffles the whole mUF-tree iteratively, which demands much computational effort. Hence, we further develop a trie structure in the lexicographic order that can facilitate the generation of a sub-trie for each item, and an algorithm that can shuffle each sub-trie rather than the whole trie and discover as many uncertain frequent patterns as possible for each item.

So we propose the second method, the mUF-trie structure with the UF-Prune algorithm, which also solves the problem in two phases. The two phases are similar to that of the mUF-tree, except a pruning step to improve its discovery efficiency.

Chapter 4 : mUF-tree

In this chapter, we introduce the mUF-tree, which is for storing summarized and uncertain information about frequent patterns, and its construction algorithm. Next, we discuss the mUF-tree based mining algorithm, UF-Evolve, which utilizes the shuffling and merging techniques to generate iterative versions of the mUF-tree for discovering new uncertain frequent patterns.

4.1 mUF-tree: Design and Construction

Given an uncertain database, we propose to use an mUF-tree to store summarized and uncertain information about frequent patterns. An mUF-tree with its header table is shown in Figure 2.

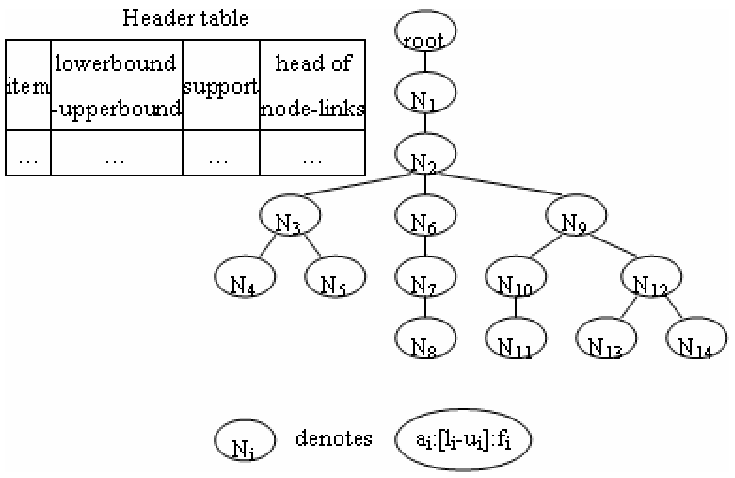


Figure 2. mUF-tree

Definition 5. Uncertain Frequent Pattern Tree for Merging (mUF-tree). An mUF-tree has the following characteristics.

1. An mUF-tree has a virtual root. Each node in the mUF-tree consists of five

attributes, item, lowerbound, upperbound, frequency and node-link. Lowerbound and upperbound register the spread of existential probabilities of the corresponding item. To facilitate tree traversal, nodes with the same item:[lowerbound-upperbound] are linked in sequence via node-links. In Figure 2, a_i:[l_i-u_i]:f_i in node N_i represents item:[lowerbound-upperbound]:frequency.

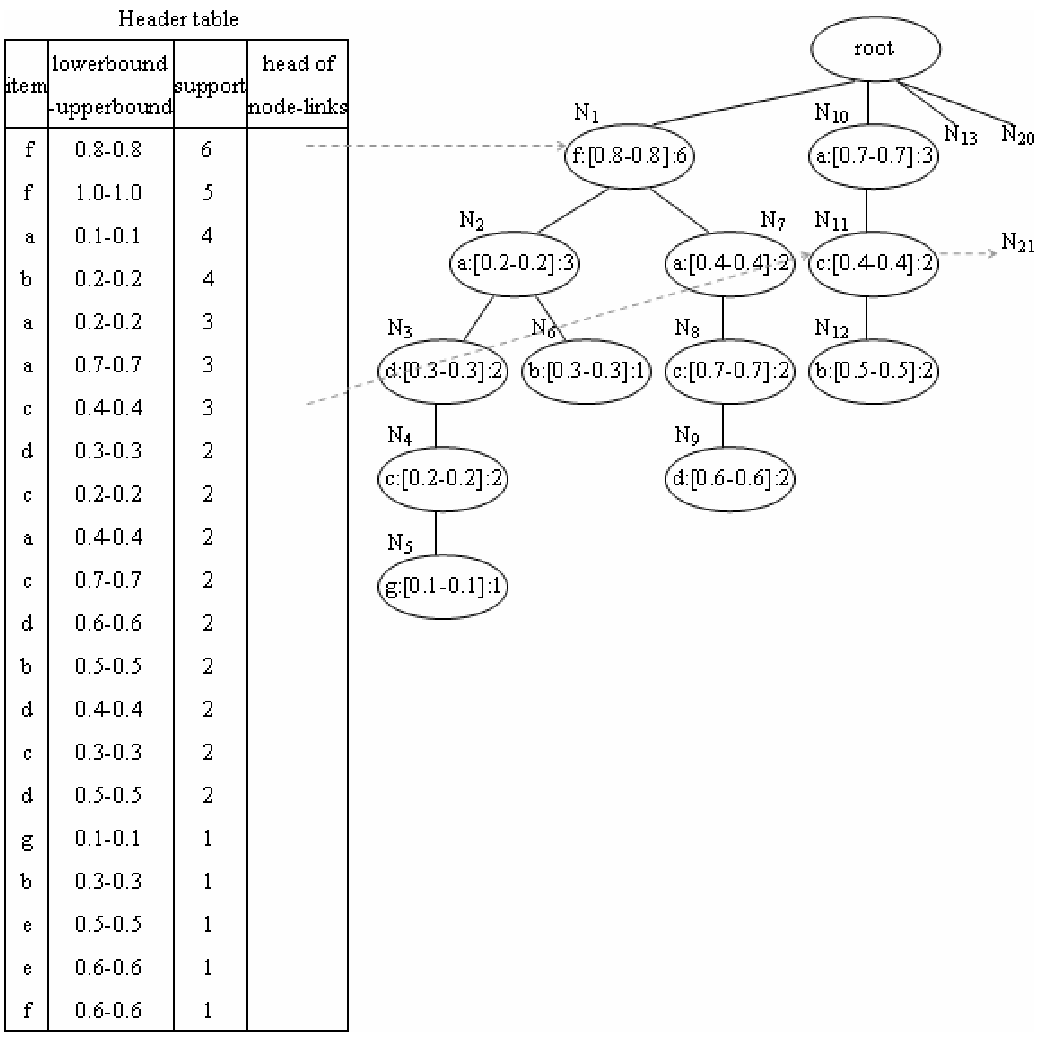
2. In the mUF-tree, each entry in the header table consists of four values: (1) item, (2) lowerbound-upperbound, (3) support (the cumulative frequency of the item:[lowerbound-upperbound] in the mUF-tree), and (4) head of node-links, which is a pointer pointing to the first node in the mUF-tree carrying the item:[lowerbound-upperbound].
3. In the header table, item:[lowerbound-upperbound] pairs are in the descending order of supports.

The UF-Construct algorithm is used to construct an initial mUF-tree from an uncertain database. Unlike the FP-tree construction algorithm [16], UF-Construct scans the database without specifying a minimum support threshold. Hence, the mUF-tree constructed contains all information. Figure 3 shows the steps of the UF-Construct algorithm.

```
UF-Construct algorithm.
Input: A UDB.
Output: An mUF-tree T.

Procedure UF-Construct(UDB)
(1)  scan UDB once and collect a_set = the set of item:probability pairs with
       their respective supports in descending order;
(2)  create T with a root;
(3)  for each record R in UDB do {
(4)    R' = item:probability pairs in R putting in the same order as in a_set;
(5)    r = root(T);
(6)    for i = 1 to length(R') do {
(7)      $(a_i:p_i)$ = the $i^{th}$ item:probability pair of R';
(8)      find $N_c$ = child(r), where $(N_c.a_c = a_i) \wedge (N_c.l_c = p_i)$;
(9)      if $\exists N_c$ then $N_c.f_c$ ++;
(10)     else {
(11)       create node $N_c$ with $(N_c.a_c = a_i) \wedge (N_c.l_c = N_c.u_c = p_i) \wedge$
             $(N_c.f_c = 1)$;
(12)       insert $N_c$ as the rightmost child of r;
         }
(13)     update header table and node-links within T;
(14)     r = $N_c$;
       }
     }
(15) return T;
```

Figure 3. UF-Construct

With the records in Table 2 of Chapter 3, the mUF-tree together with the associated node-links is shown in Figure 4. Since the constructed mUF-tree is large, we split it into two parts. Figure 4(a) shows the header table with the left half of the mUF-tree, and Figure 4(b) shows the right half of the mUF-tree. For the ease of understanding, we only show some of the node-links.



Header table

item	lowerbound -upperbound	support	head of node-links
f	0.8-0.8	6	
f	1.0-1.0	5	
a	0.1-0.1	4	
b	0.2-0.2	4	
a	0.2-0.2	3	
a	0.7-0.7	3	
c	0.4-0.4	3	
d	0.3-0.3	2	
c	0.2-0.2	2	
a	0.4-0.4	2	
c	0.7-0.7	2	
d	0.6-0.6	2	
b	0.5-0.5	2	
d	0.4-0.4	2	
c	0.3-0.3	2	
d	0.5-0.5	2	
g	0.1-0.1	1	
b	0.3-0.3	1	
e	0.5-0.5	1	
e	0.6-0.6	1	
f	0.6-0.6	1	

Figure 4(a). mUF-tree(left)

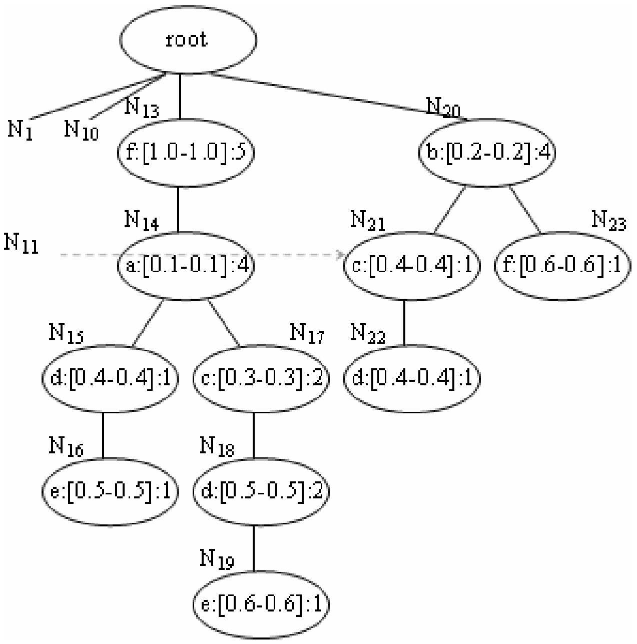


Figure 4(b). mUF-tree(right)

Figure 4. The mUF-tree for data in Table 2

4.2 Discovering New Uncertain Frequent Patterns

The mUF-tree constructed in Figure 4 has items with their probability ranges in which lowerbound and upperbound are equal to the original probability. If we want to use the FP-growth algorithm [16] to mine frequent patterns with mUF-tree(right) in Figure 4(b), we can consider a_i:[l_i-u_i] as an item. Then, FP-growth will discover a set of four frequent patterns {(b:[0.2-0.2]):4, (a:[0.1-0.1]):4, (f:[1.0-1.0])(a:[0.1-0.1]):4, (f:[1.0-1.0]):5} with the minimum support threshold be 3.

In an mUF-tree, it is often that an item with a [lowerbound-upperbound] appears in a node while the same item with another [lowerbound-upperbound] appears in another node. For example, in Figure 4(b), d:[0.4-0.4] appears in N_{15} while d:[0.5-0.5] appears in N_{18}. One can consider merging these items when there are only small differences in their various [lowerbound-upperbound] for satisfying the support. For example, we can merge d:[0.4-0.4] and d:[0.5-0.5] as d:[0.4-0.5]. Hence, for a group of item:[lowerbound-upperbound] pairs of the same item, we can use a combined [lowerbound-upperbound] to represent it. Once the item:[lowerbound-upperbound] pairs are grouped, more general frequent patterns can be found.

Here, we are proposing to further discover new uncertain frequent patterns by utilizing shuffling and merging of the mUF-tree. Nodes can be merged to evolve into another mUF-tree for further pattern mining. The steps can be repeated until merging is not possible.

4.2.1 Preliminary Definitions

For the ease of later discussion, some definitions are stated before presenting the algorithms.

Definition 6. Within Range. Given a maximum merging threshold γ, two nodes N_b and N_c are within range if $(N_b.a_b = N_c.a_c) \wedge (\text{abs}(\max(N_b.u_b, N_c.u_c) - \min(N_b.l_b, N_c.l_c)) \leq \gamma)$.

In Figure 4(b), given a maximum merging threshold 0.3, N_{15} and N_{18} are within range. Also, N_{16} and N_{19} are within range.

Definition 7. Common Items (CI). Given a pair of paths PB = $<N_{b1}N_{b2}\ldots N_{bk}>$ and PC = $<N_{c1}N_{c2}\ldots N_{cm}>$, CI(PB, PC) = $\{a_1, a_2, \ldots, a_n\}$ is a sequence of ordered items, where $(n \leq k) \wedge (n \leq m) \wedge (\forall\ i \in \{1, 2, \ldots, n\}$, there is a node in PB and a node in PC, such that these two nodes contain a_i and are within range).

In Figure 4(b), given a pair of paths PB = <$N_{15}N_{16}$> and PC = <$N_{17}N_{18}N_{19}$>, then CI(PB, PC) = {d, e}. We intend to merge the nodes corresponding to common items in the two paths. However, it is difficult to merge N_{15} with N_{18} and N_{16} with N_{19} directly. These nodes should be moved in the same order first. Here, we define the Maximum Attainable Peak (MAP) of a node in a path, and how to shuffle a node to its MAP.

Definition 8. Maximum Attainable Peak (MAP). Given a_set = {a_{11}, a_{12}, …, a_{1k}} as a sequence of ordered items, and a path PB = <$N_{21}N_{22}…N_{2m}$>, where the items in a_set is a subset of the items in PB. Each item a_{1i} in a_set is taken out sequentially and is traced along PB. If the position of a_{1i} in PB is not as in a_set, the node N_{2j} containing a_{1i} will be moved upward until the position of a_{1i} in PB is as in a_set. The final position is called the MAP of N_{2j} in PB.

Definition 9. Shuffling a Node to MAP. Given a node N_{2j} and its MAP in a path PB, N_{2j} can be swapped with its parent node and do this repeatedly until it reaches the MAP in PB. There are two possible cases for shuffling the node N_{2j} with its parent N_q.

- Case 1: $N_{2j}.f_{2j} = N_q.f_q$. Swap N_{2j} and N_q.
- Case 2: $N_{2j}.f_{2j} < N_q.f_q$. Create a new node, N_r, with the same property of N_q except $N_r.f_r = N_q.f_q – N_{2j}.f_{2j}$. Make N_r as another child of the parent of N_q. Set $N_q.f_q = N_{2j}.f_{2j}$. Update the header table and node-links within the mUF-tree, and then follow the step of Case 1 for N_{2j} and N_q.

In Figure 4(b), given a_set = {d, e}, then the MAP of N_{18} in the path <$N_{17}N_{18}N_{19}$> is the 1st position. We shuffle N_{18} with N_{17}, and follow the step of Case

1 afterwards. mUF-tree(right) evolves into mUF-tree(right)$_2$, as shown in Figure 5.

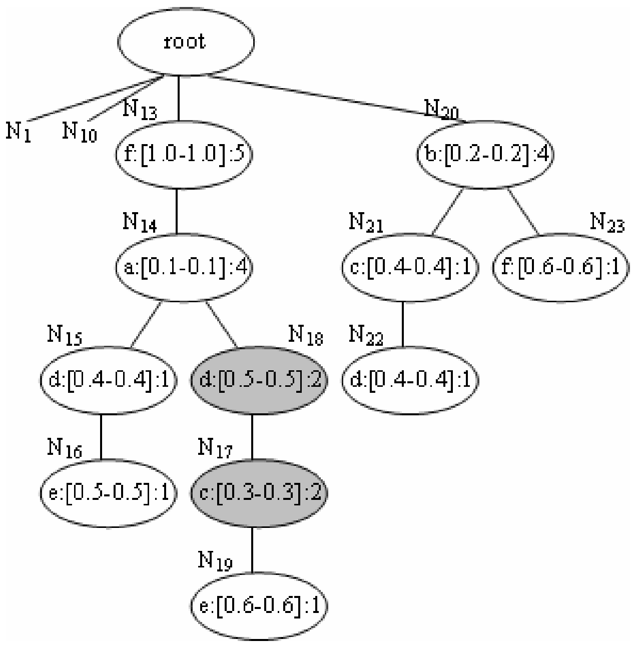


Figure 5. mUF-tree(right)$_2$

After shuffling N_{18} to its MAP, the MAP of N_{19} in the updated path <$N_{18}N_{17}N_{19}$> is the 2nd position. N_{19} will be shuffled with N_{17}, which follows the steps of Case 2. A new node N_{24} is created and N_{17} is modified, which evolves into mUF-tree(right)$_3$, as shown in Figure 6. Then N_{19} is shuffled with N_{17}, while the tree will be evolved into mUF-tree(right)$_4$, as shown in Figure 7.

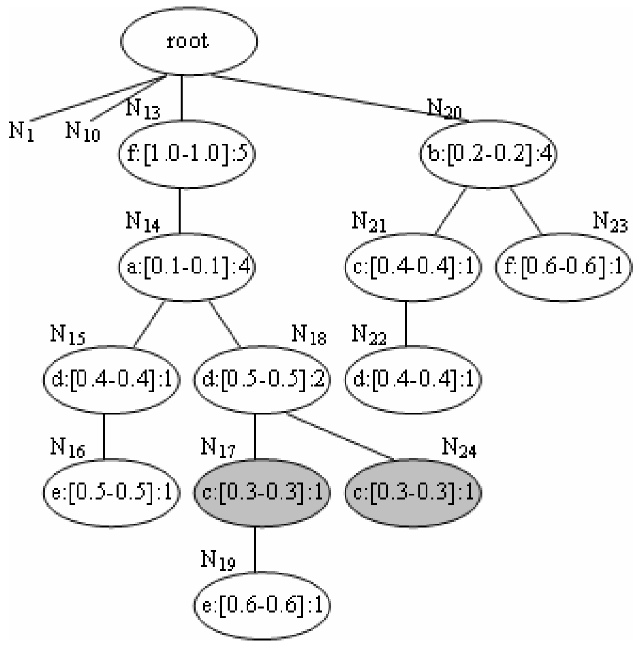


Figure 6. mUF-tree(right)$_3$

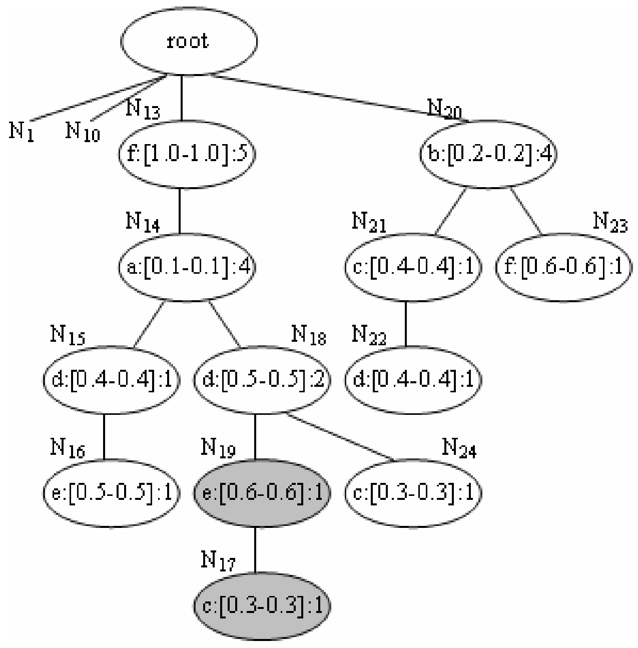


Figure 7. mUF-tree(right)$_4$

After shuffling, the nodes corresponding to common items are above other nodes in the two paths. It is straightforward now to merge N$_{15}$ with N$_{18}$ and N$_{16}$ with

N_{19}. Next, we define how two candidate nodes can be merged to form a new node.

Definition 10. Merging Nodes. If two nodes N_b and N_c are within range, they can be merged as a new node N_q, where $(N_q.a_q = N_b.a_b) \wedge (N_q.l_q = \min(N_b.l_b, N_c.l_c)) \wedge (N_q.u_q = \max(N_b.u_b, N_c.u_c)) \wedge (N_q.f_q = N_b.f_b + N_c.f_c)$.

N_{15} and N_{18} are merged as a new node N_{25}, which evolves into mUF-tree(right)$_5$, as shown in Figure 8. Next, N_{16} is merged with N_{19} as a new node N_{26}, which evolves into mUF-tree(right)$_6$, as shown in Figure 9.

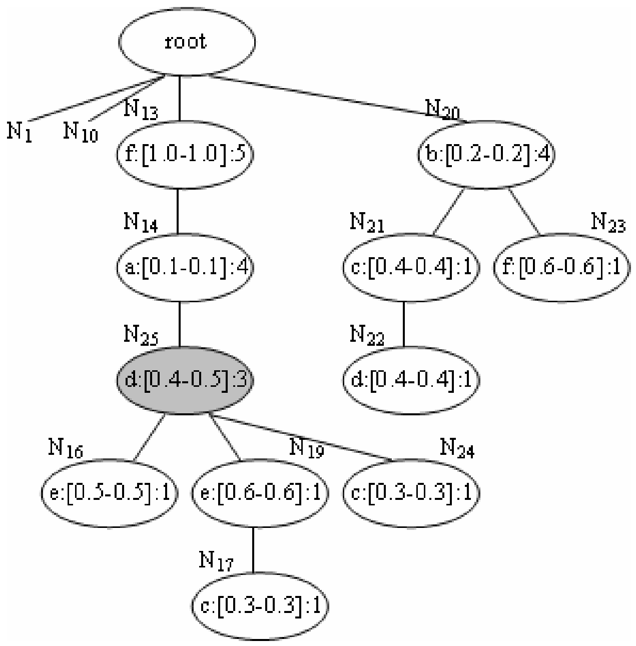


Figure 8. mUF-tree(right)$_5$

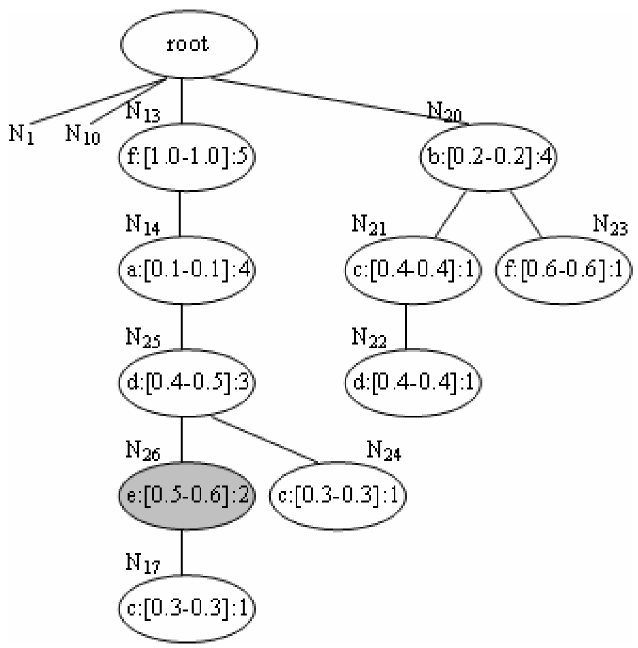


Figure 9. mUF-tree(right)$_6$

Definition 11. Overlap. Given a path PB, overlap(PB) is true if any node in PB has more than one child**.**

In Figure 4(b), given PB = <$N_{13}N_{14}N_{15}N_{16}$> and PC = <$N_{17}N_{18}N_{19}$>, then overlap(PB) since N_{14} in PB has two children, and !overlap(PC) (i.e. not overlap(PC)) since each node in PC has no more than one child.

Definition 12. Above. Given a_set = {a_{11}, a_{12}, …, a_{1k}} and a path PB = <$N_{21}N_{22}\ldots N_{2m}$>, above(a_set , PB) is true if $(k \leq m) \wedge (\forall\ i \in \{1, 2, \ldots, k\}, a_{1i} = N_{2i}.a_{2i})$.

In Figure 4(b), given a_set = {d, e}, PB = <$N_{15}N_{16}$> and PC = <$N_{17}N_{18}N_{19}$>, then above(a_set, PB) since d and e are above other items in PB, and !above(a_set, PC) since d and e are not above other items in PC.

Other than the shuffling cases presented in Definition 9, there can be five cases for shuffling a pair of paths PB and PC. In Case 1, both PB and PC do not share common nodes with any other paths in the mUF-tree. Therefore, PB and PC can be shuffled without affecting others. In Case 2, PB does not share common nodes with any other paths, but PC does. However, the nodes with common items in PC are above other nodes. Therefore, we do not need to shuffle PC. Case 3 is similar to Case 2 with the situations in PB and PC interchanged. In Case 4, both PB and PC share common nodes with other paths, and there are nodes having common items above other nodes. Therefore, neither PB nor PC needs to be shuffled. For all the four cases, PB and PC can be merged after shuffling if necessary.

All other conditions besides the above are considered as Case 0. In Case 0, shuffling is not performed as it may induce too much effort in restructuring of the whole mUF-tree. Here, we define the five cases formally below.

Definition 13. Shuffle Case (SC). Given a pair of paths PB and PC, and a_set = CI(PB, PC), there are five shuffle cases.

- SC(PB, PC) = 1 if !overlap(PB) $\wedge$!overlap(PC).
- SC(PB, PC) = 2 if !overlap(PB) $\wedge$ overlap(PC) $\wedge$ above(a_set, PC).
- SC(PB, PC) = 3 if overlap(PB) $\wedge$ above(a_set, PB) $\wedge$!overlap(PC).
- SC(PB, PC) = 4 if overlap(PB) $\wedge$ above(a_set, PB) $\wedge$ overlap(PC) $\wedge$ above(a_set, PC).
- SC(PB, PC) = 0 for other conditions.

For the ease of understanding, the different shuffle cases and their respective criteria are summarized in Table 3.

SC(PB, PC)	!overlap(PC)	overlap(PC) ∧ above(a_set, PC)
!overlap(PB)	1	2
overlap(PB) ∧ above(a_set, PB)	3	4

Table 3. Shuffle cases

Take the mUF-tree in Figure 4 as an example. Given a maximum merging threshold 0.3, then the shuffle cases with the corresponding path pairs are shown in Table 4.

PB	**PC**	**CI(PB, PC)**	**SC(PB, PC)**
$<N_{15}N_{16}>$	$<N_{17}N_{18}N_{19}>$	{d, e}	1
$<N_{10}N_{11}N_{12}>$	$<N_{20}N_{21}N_{22}>$	{b, c}	2
$<N_2N_3N_4N_5>$	$<N_7N_8N_9>$	{a, d}	3
$<N_1N_2N_3N_4N_5>$	$<N_{13}N_{14}N_{15}N_{16}>$	{f, a, d}	4
$<N_{10}N_{11}N_{12}>$	$<N_{13}N_{14}N_{17}N_{18}N_{19}>$	{c}	0
$<N_1N_7N_8N_9>$	$<N_{20}N_{23}>$	{f}	0
$<N_1N_2N_6>$	$<N_{10}N_{11}N_{12}>$	{b}	0
$<N_1N_2N_6>$	$<N_{20}N_{23}>$	{b, f}	0
$<N_1N_7N_8N_9>$	$<N_{20}N_{21}N_{22}>$	{c, d}	0

Table 4. Shuffle cases with corresponding path pairs

Definition 14. Common Ancestor Path (CAP). Given a pair of paths PB = $<N_{b1}N_{b2}\ldots N_{bk}>$ and PC = $<N_{c1}N_{c2}\ldots N_{cm}>$, CAP(PB, PC) is $<N_{b1}N_{b2}\ldots N_{bn}>$, where

$(n \leq k) \wedge (n \leq m) \wedge (\forall\ i \in \{1, 2, \ldots, n\}, N_{bi}$ and N_{ci} are the same node).

In Figure 4(b), given a pair of paths PB = $<N_{13}N_{14}N_{15}N_{16}>$ and PC = $<N_{13}N_{14}N_{17}N_{18}N_{19}>$, then CAP(PB, PC) = $<N_{13}N_{14}>$.

Definition 15. Single Prefix-Path Part and Multipath Part. The single prefix-path part of an mUF-tree consists of a single path from the root to N_k, the first node may contain more than one child. The multipath part of an mUF-tree consists of the descendants of N_k, with a virtual root connecting to the children of N_k as the parent.

For the mUF-tree shown in Figure 2, the single prefix-path part and the multipath part are shown in Figure 10.

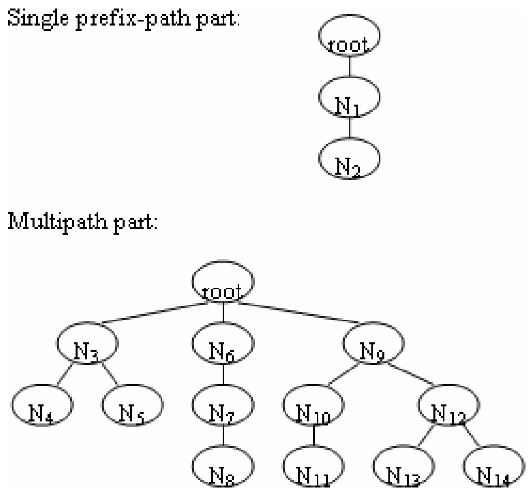


Figure 10. Single prefix-path part and multipath part of an mUF-tree

4.2.2 The UF-Evolve Algorithm

With the aforementioned definitions, we present the UF-Evolve algorithm for mining frequent patterns in an uncertain database by using an mUF-tree. The algorithm integrates the UF-Mine algorithm for finding out the possible frequent patterns and the UF-Shuffle algorithm for moving and merging the nodes in the mUF-tree iteratively. Figure 11 shows the steps of the UF-Evolve algorithm.

```
UF-Evolve algorithm.
Input: An mUF-tree T, a minimum support threshold $\sigma$, and a maximum merging
   threshold $\gamma$.
Output: A set of uncertain frequent patterns.

Procedure UF-Evolve(T, $\sigma$, $\gamma$)
(1)    FPS = $\phi$;   // Frequent pattern set
(2)    do {
(3)       FPS = FPS $\cup$ UF-Mine(T, null, $\sigma$);
(4)       T = UF-Shuffle(T, $\gamma$);
(5)    } while T has been modified;
(6)    return FPS;
```

Figure 11. UF-Evolve

The UF-Mine algorithm is a variant of the FP-growth algorithm [16]. It has two phases. It first attempts to find frequent patterns in the single prefix-path part of the given mUF-tree, and then continues to work out the multipath part. The details are shown in Figure 12.

```
UF-Mine algorithm.
Input: An mUF-tree T, an uncertain frequent pattern $\alpha$ =
   $(a_1:[l_1-u_1])(a_2:[l_2-u_2])\ldots(a_k:[l_k-u_k])$:s, and a minimum support threshold $\sigma$.
Output: A set of uncertain frequent patterns.

Procedure UF-Mine(T, $\alpha$, $\sigma$)
(1)   P = the single prefix-path part of T;
(2)   Q = the multipath part of T;
(3)   FPS(P) = $\phi$;   // Frequent pattern set in P
(4)   FPS(Q) = $\phi$;   // Frequent pattern set in Q
(5)   for each pattern $\beta$ formed from P where all nodes in $\beta$ have sufficient
         supports do {
(6)      $\beta$.support = minimum support of nodes in $\beta$;
(7)      append $\alpha$ to the end of $\beta$;
(8)      FPS(P) = FPS(P) $\cup$ $\beta$;
      }
(9)   for each $(a_i:[l_i-u_i])$ in Q with sufficient support do {
(10)     $\beta$ = $(a_i:[l_i-u_i])$;
(11)     $\beta$.support = $(a_i:[l_i-u_i])$.support;
(12)     append $\alpha$ to the end of $\beta$;
(13)     FPS(Q) = FPS(Q) $\cup$ $\beta$;
(14)     find cond_tree from Q constructed with the conditional pattern-base of
            $\beta$;   // Conditional mUF-tree of $\beta$
(15)     if $\exists$ cond_tree then
(16)        FPS(Q) = FPS(Q) $\cup$ UF-Mine(cond_tree, $\beta$, $\sigma$);
      }
(17)  return FPS(P) $\cup$ FPS(Q) $\cup$ (FPS(P) $\times$ FPS(Q));
```

Figure 12. UF-Mine

Note that in line (17) of Figure 12, (FPS(P) $\times$ FPS(Q)) means concatenating each frequent pattern FP_i in FPS(P) with each frequent pattern FP_j in FPS(Q), with support equal to FP_j.support. In order to illustrate how the algorithms work, we continue to use the running example in Figure 4(b). Suppose the minimum support threshold is 3, and the maximum merging threshold is 0.3. Initially, UF-Mine returns a set of uncertain frequent patterns as {(b:[0.2-0.2]):4, (a:[0.1-0.1]):4, (f:[1.0-1.0])(a:[0.1-0.1]):4, (f:[1.0-1.0]):5}.

An important part of our work is the UF-Shuffle algorithm, which is shown in Figure 13. When it is called by UF-Evolve, it collects the set of paths under the root and finds the most suitable pair of paths to shuffle and merge. The most suitable pair of paths is the one with the largest number of common items, and with a shuffle case 1, 2, 3 or 4. This is because we are more interested in mining longer frequent patterns. If there is more than one pair of paths, the leftmost pair of paths is selected to be the most suitable pair. At present, we do not aim to mine all possible frequent patterns. There are two reasons. One reason is that the shuffle cases 1, 2, 3 and 4 limit the possible number of shuffles. The other reason is that the characteristic of probability values limits the possible number of frequent patterns. For example, there are several item:probability pairs in an uncertain database, but in different records, i.e. (a:0.1), (a:0.3) and (a:0.5). Suppose the maximum merging threshold is 0.3. If we merge (a:0.1) with (a:0.3), then we will miss the patterns arising from merging (a:0.3) with (a:0.5). Therefore, finding all patterns is impossible for us.

```
UF-Shuffle algorithm.
Input: An mUF-tree T, and a maximum merging threshold γ.
Output: A shuffled mUF-tree T.

Procedure UF-Shuffle(T, γ)
(1)   scan T once and collect P_set = the set of paths under root(T);
(2)   for each pair of paths PB and PC in P_set do {
(3)       P = CAP(PB, PC);
(4)       PB = PB excludes P;
(5)       PC = PC excludes P;
(6)       get CI(PB, PC) and SC(PB, PC);
      }
(7)   suppose (PB', PC') is a pair of paths that (has maximum length(CI(PB,
          PC))) ∧ (length(CI(PB', PC')) > 0) ∧ (SC(PB', PC') > 0);
(8)   if ∃ (PB', PC') then {
(9)       switch SC(PB', PC') {
(10)          case 1: shuffle the nodes corresponding to CI(PB', PC') to their MAP
                  in PB' and PC';
(11)          case 2: shuffle the nodes corresponding to CI(PB', PC') to their MAP
                  in PB';
(12)          case 3: shuffle the nodes corresponding to CI(PB', PC') to their MAP
                  in PC';
(13)          case 4: do nothing;
          }
(14)      T = UF-Merge(T, PB', PC', length(CI(PB', PC')));
      }
(15)  return T;
```

Figure 13. UF-Shuffle

There can be different versions of the UF-Shuffle algorithm. Figure 14 shows UF-Shuffle_2, which is a variant of UF-Shuffle. When it is called by UF-Evolve, it collects the set of paths under the root and attempts to shuffle and merge each pair of paths. If a pair of paths are shuffled and merged, they will be removed from the set of paths since they have been modified and no longer exist. UF-Shuffle shuffles and merges the most suitable pair of paths in each iteration, which would mine longer and intermediate frequent patterns; while UF-Shuffle_2 shuffles and merges many pairs of paths in each iteration, which will save some computational effort. In our experiments presented in Chapter 6, we have adopted the UF-Shuffle algorithm only. The reason is that UF-Shuffle returns the intermediate results after each step of shuffling and merging, and UF-Shuffle_2 only returns the results after the final merge for mining.

```
Procedure UF-Shuffle_2(T, γ)
(1)    scan T once and collect P_set = the set of paths under root(T);
(2)    for each pair of paths PB and PC in P_set do {
(3)       P = CAP(PB, PC);
(4)       PB = PB excludes P;
(5)       PC = PC excludes P;
(6)       if (length(CI(PB, PC)) > 0)  ∧  (SC(PB, PC) > 0) then {
(7)          switch SC(PB, PC) {
(8)             case 1: shuffle the nodes corresponding to CI(PB, PC) to their MAP
                   in PB and PC;
(9)             case 2: shuffle the nodes corresponding to CI(PB, PC) to their MAP
                   in PB;
(10)            case 3: shuffle the nodes corresponding to CI(PB, PC) to their MAP
                   in PC;
(11)            case 4: do nothing;
             }
(12)         T = UF-Merge(T, PB, PC, length(CI(PB, PC)));
(13)         remove PB and PC from P_set;
                // Updates will be used in loop at line (2)
          }
       }
(14)   return T;
```

Figure 14. UF-Shuffle_2

After shuffling, candidate nodes can then be merged with respective updating. The algorithm of UF-Merge is shown in Figure 15.

```
UF-Merge algorithm.
Input: An mUF-tree T, a pair of paths PB = <N_b1 N_b2...N_bk> and PC =
   <N_c1 N_c2...N_cm>, and n, the number of nodes to be merged in each path.
Output: A merged mUF-tree T.

Procedure UF-Merge(T, PB, PC, n)
(1)    for i = 1 to n do {
(2)        N_q = the new node formed by merging N_bi with N_ci and updating its item,
               lowerbound, upperbound, frequency, node-link, parent and children;
(3)        remove N_bi and N_ci from T;
(4)        update header table and node-links within T;
       }
(5)    return T;
```

Figure 15. UF-Merge

In continuing the running example, UF-Shuffle is called to shuffle mUF-tree(right) in Figure 4(b), and it generates mUF-tree(right)$_4$, as shown in Figure 7. UF-Merge is then invoked to recommend the pair of paths to be merged (i.e. <$N_{15}N_{16}$> and <$N_{18}N_{19}N_{17}$>). It merges the pair of paths, and returns the generated mUF-tree(right)$_6$, as shown in Figure 9.

After having a new mUF-tree, UF-Mine is invoked again and returns a new set of uncertain frequent patterns as {(d:[0.4-0.5]):3, (a:[0.1-0.1])(d:[0.4-0.5]):3, (f:[1.0-1.0])(d:[0.4-0.5]):3, (f:[1.0-1.0])(a:[0.1-0.1])(d:[0.4-0.5]):3, (b:[0.2-0.2]):4, (a:[0.1-0.1]):4, (f:[1.0-1.0])(a:[0.1-0.1]):4, (f:[1.0-1.0]):5}. Then, UF-Evolve combines the original set of uncertain frequent patterns with the new set. The discovered uncertain frequent patterns in the two iterations are shown in Table 5. The algorithms continue until no new mUF-tree can be constructed.

Uncertain frequent patterns mined in 1st iteration	**New uncertain frequent patterns mined in 2nd iteration**
(b:[0.2-0.2]):4	(d:[0.4-0.5]):3
(a:[0.1-0.1]):4	(a:[0.1-0.1])(d:[0.4-0.5]):3
(f:[1.0-1.0])(a:[0.1-0.1]):4	(f:[1.0-1.0])(d:[0.4-0.5]):3
(f:[1.0-1.0]):5	(f:[1.0-1.0])(a:[0.1-0.1])(d:[0.4-0.5]):3

Table 5. Discovered uncertain frequent patterns after two iterations with mUF-tree

Chapter 5 : mUF-trie

In Chapter 4, the UF-Evolve algorithm collects the set of paths under the root of the mUF-tree and finds the pairs of paths to shuffle and merge. It shuffles the whole mUF-tree repeatedly, which demands much computational effort. What is more, the candidate paths in the whole mUF-tree are long, and the shuffle cases 1, 2, 3 and 4 limit the possible number of shuffles. Therefore, we want to improve these aspects and find uncertain frequent patterns more efficiently.

In this chapter, we introduce the mUF-trie, which is for storing summarized and uncertain information about frequent patterns in the lexicographic order. Its construction algorithm facilitates the generation of sub-tries and mining uncertain frequent patterns for each item. The mUF-trie utilizes the UF-Prune algorithm, which continuously generates a sub-trie for each item, applies the shuffling and merging techniques to generate iterative versions of the sub-trie for discovering new patterns, and prunes away the item in the mUF-trie. UF-Prune can shuffle each sub-trie rather than the whole mUF-trie and discover as many uncertain frequent patterns as possible for each item.

5.1 mUF-trie: Design and Construction

Given an uncertain database, an mUF-trie is used to store summarized and uncertain information about frequent patterns. It has a header table as shown in Figure 16. The attributes in each node and the columns in the header table are the same as that in the mUF-tree, but the uncertain information is summarized and stored in a different order.

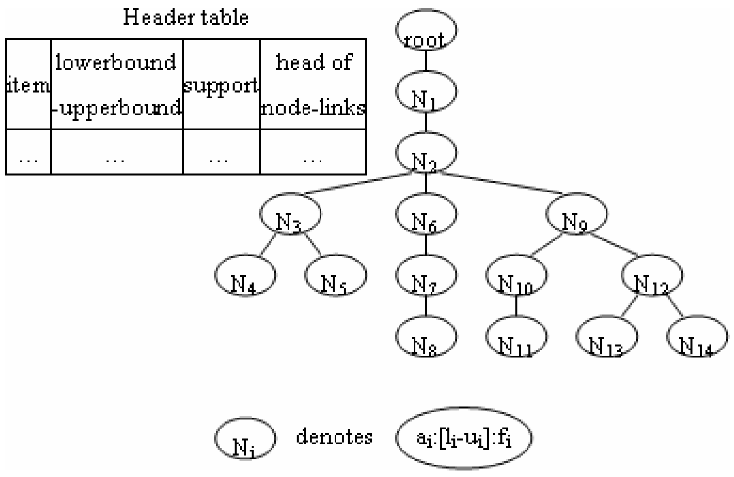


Figure 16. mUF-trie

Definition 16. Uncertain Frequent Pattern Trie for Merging (mUF-trie). An mUF-trie has the following characteristics.

1. An mUF-trie has a virtual root. Each node in the mUF-trie consists of five attributes, item, lowerbound, upperbound, frequency and node-link. Lowerbound and upperbound register the spread of existential probabilities of the corresponding item. To facilitate trie traversal, nodes with the same item:[lowerbound-upperbound] are linked in sequence via node-links. In Figure 16, a_i:[l_i-u_i]:f_i in node N_i represents item:[lowerbound-upperbound]:frequency.
2. In the mUF-trie, each entry in the header table consists of four values: (1) item, (2) lowerbound-upperbound, (3) support (the cumulative frequency of the item:[lowerbound-upperbound] in the mUF-trie), and (4) head of node-links, which is a pointer pointing to the first node in the mUF-trie carrying the item:[lowerbound-upperbound].
3. In the mUF-trie, items are in the lexicographic order from left to right and from top to bottom. If siblings are of the same item, they are in the ascending order of lowerbounds (if lowerbounds are the same, use upperbounds instead) from left

to right. In the header table, items are in the lexicographic order from top to bottom. If the item:[lowerbound-upperbound] pairs are of the same item, they are in the ascending order of lowerbounds (if lowerbounds are the same, use upperbounds instead) from top to bottom. This ordering is called mUF-trie-ordering.

The UF-Build algorithm is used to construct an initial mUF-trie from an uncertain database. UF-Build scans the database once and collects items whose supports in the database are no less than a predefined threshold. Here, we do not only collect frequent item:probability pairs, but also frequent items with different probabilities, since these item with different probabilities may be further grouped. Unlike the FP-tree construction algorithm [16], UF-Build sorts the frequent items in each record. In the constructed mUF-trie, the items are ordered in mUF-trie-ordering. This ordering facilitates the generation and mining of the sub-trie for each item. Compared with mUF-tree, the mUF-trie has a predefined minimum support threshold. The mUF-tree collects the set of item:probability pairs and sorts them in the descending order of supports, while the mUF-trie collects the set of frequent items. Figure 17 shows the steps of the UF-Build algorithm.

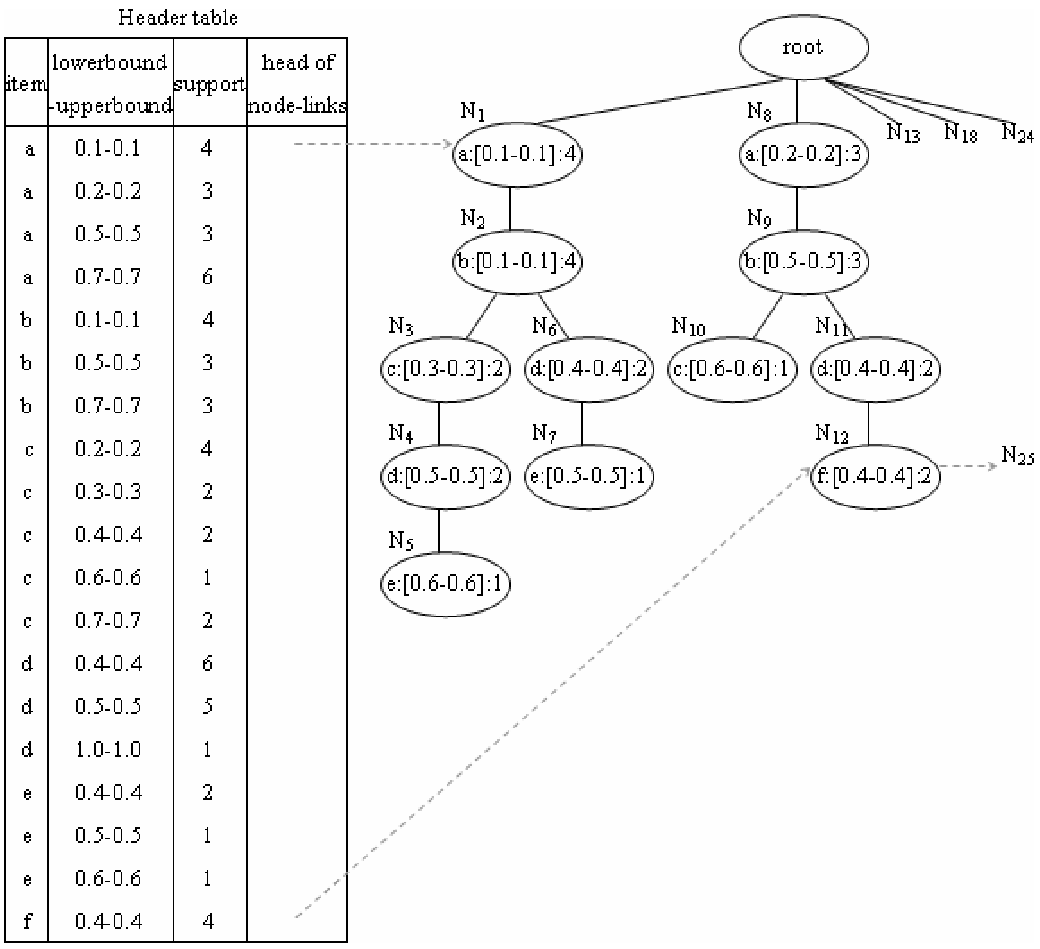
```
UF-Build algorithm.
Input: A UDB, and a minimum support threshold  $\sigma$.
Output: An mUF-trie T.

Procedure UF-Build(UDB,  $\sigma$)
(1)   scan UDB once and collect a_set = the set of items with their respective
          supports  $\geq \sigma$;
(2)   create T with a root;
(3)   for each record R in UDB do {
(4)       R' = item:probability pairs in R with items of a_set in lexicographic order;
(5)       r = root(T);
(6)       for i = 1 to length(R') do {
(7)           $(a_i:p_i)$ = the $i^{th}$ item:probability pair of R';
(8)           find $N_c$ = child(r), where $(N_c.a_c = a_i) \wedge (N_c.l_c = p_i)$;
(9)           if $\exists$ $N_c$ then $N_c.f_c$ ++;
(10)          else {
(11)              create node $N_c$ with $(N_c.a_c = a_i) \wedge (N_c.l_c = N_c.u_c = p_i) \wedge$
                      $(N_c.f_c = 1)$;
(12)              insert $N_c$ as a child of r according to mUF-trie-ordering;
              }
(13)          update header table and node-links within T;
(14)          r = $N_c$;
          }
      }
(15)  return T;
```

Figure 17. UF-Build

With the records in Table 6, the mUF-trie together with the associated node-links is shown in Figure 18 with the minimum support threshold be 4. Since the constructed mUF-trie is large, we split it into two parts. Figure 18(a) shows the header table with the left half of the mUF-trie, and Figure 18(b) shows the right half of the mUF-trie. For the ease of understanding, we only show some of the node-links.

Record ID	**Item:probability pairs**
1	(c:0.7), (f:0.4)
2	(d:0.4), (e:0.4), (a:0.5), (c:0.2)
3	(c:0.2), (g:0.2), (d:0.4), (e:0.4), (a:0.5)
4	(b:0.1), (a:0.1), (c:0.3), (e:0.6), (d:0.5)
5	(f:0.4), (c:0.7)
6	(d:0.5), (a:0.5)
7	(b:0.5), (a:0.2), (c:0.6)
8	(f:0.4), (d:0.4), (b:0.5), (g:0.5), (a:0.2)
9	(a:0.7)
10	(b:0.1), (d:0.4), (a:0.1)
11	(d:0.5), (b:0.7), (g:0.3), (c:0.4), (a:0.7)
12	(a:0.7), (c:0.4), (b:0.7), (d:0.5)
13	(d:0.4), (e:0.5), (a:0.1), (b:0.1)
14	(a:0.7), (b:0.7)
15	(f:0.4), (a:0.2), (b:0.5), (d:0.4)
16	(a:0.7), (d:1.0), (c:0.2)
17	(c:0.2), (a:0.7)
18	(b:0.1), (a:0.1), (d:0.5), (c:0.3)

Table 6. An uncertain database for mUF-trie



Header table

item	lowerbound -upperbound	support	head of node-links
a	0.1-0.1	4	
a	0.2-0.2	3	
a	0.5-0.5	3	
a	0.7-0.7	6	
b	0.1-0.1	4	
b	0.5-0.5	3	
b	0.7-0.7	3	
c	0.2-0.2	4	
c	0.3-0.3	2	
c	0.4-0.4	2	
c	0.6-0.6	1	
c	0.7-0.7	2	
d	0.4-0.4	6	
d	0.5-0.5	5	
d	1.0-1.0	1	
e	0.4-0.4	2	
e	0.5-0.5	1	
e	0.6-0.6	1	
f	0.4-0.4	4	

Figure 18(a). mUF-trie(left)

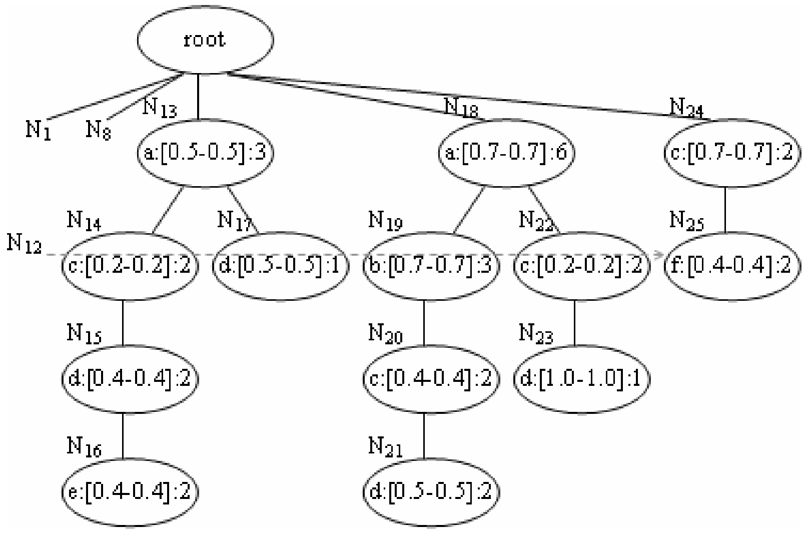


Figure 18(b). mUF-trie(right)

Figure 18. The mUF-trie for data in Table 6

5.2 Discovering New Uncertain Frequent Patterns

The mUF-trie constructed in Figure 18 has items with their probability ranges in which lowerbound and upperbound are equal to the original probability. If we want to use the FP-growth algorithm [16] to mine frequent patterns with the mUF-trie in Figure 18, we can consider a_i:[l_i-u_i] as an item. Then, FP-growth will discover a set of six frequent patterns {(a:[0.1-0.1])(b:[0.1-0.1]):4, (a:[0.7-0.7]):6, (c:[0.2-0.2]):4, (d:[0.4-0.4]):6, (d:[0.5-0.5]):5, (f:[0.4-0.4]):4} with the minimum support threshold be 4.

As with the mUF-tree, in an mUF-trie, there is an item with a [lowerbound-upperbound] appears in a node while the same item with another [lowerbound-upperbound] appears in another node. Similarly, one can consider merging the same items when there are only small differences in their various [lowerbound-upperbound] for satisfying the support. For example, in Figure 18(a), d:[0.4-0.4] appears in N_6 while d:[0.5-0.5] appears in N_4; we can merge d:[0.4-0.4] and d:[0.5-0.5] as d:[0.4-0.5]. Once the item:[lowerbound-upperbound] pairs are grouped, more general frequent patterns can be found.

Again, we propose to continuously generate a sub-trie for each item from the mUF-trie, further discover new uncertain frequent patterns relating to the item by utilizing shuffling and merging of the sub-trie, and prune away the item in the mUF-trie afterwards. In each sub-trie, nodes can be merged to evolve into another sub-trie for further pattern mining, and the steps can be repeated until merging is not possible.

5.2.1 Preliminary Definitions

For the ease of later discussion, some definitions are stated in this section.

Definition 17. Sub-trie (ST). With an mUF-trie T, given an item $a_i \in T$ and a minimum support threshold σ, we define ST(a_i) as a newly generated mUF-trie that consists of the nodes corresponding to a_i, a virtual root connecting to these nodes as the parent, and these nodes' descendants. Scan ST(a_i) once and collect a_set as the set of items with their respective supports $< \sigma$. In ST(a_i), prune away the nodes with items in a_set, and order the remaining nodes in mUF-trie-ordering. Let N_j be the j^{th} child of the root of ST(a_i). ST$(a_i)_j$ is defined as an mUF-trie that consists of N_j and its descendents, with N_j as the root. Note that ST$(a_i) \notin$ T, but ST$(a_i)_j \in$ ST(a_i).

Definition 18. Pruning Away a Node in mUF-trie. A node N_i in an mUF-trie can be pruned away by removing it from the mUF-trie, reconnecting its parent with its children, if any, and updating the header table and node-links within the mUF-trie.

With the mUF-trie shown in Figure 18, given an item a and a minimum support threshold 4, ST(a) is shown in Figure 19. The node corresponding to item f is pruned away since the support of f in ST(a) is 2. ST(a) includes ST$(a)_1$, ST$(a)_2$, ST$(a)_3$ and ST$(a)_4$.

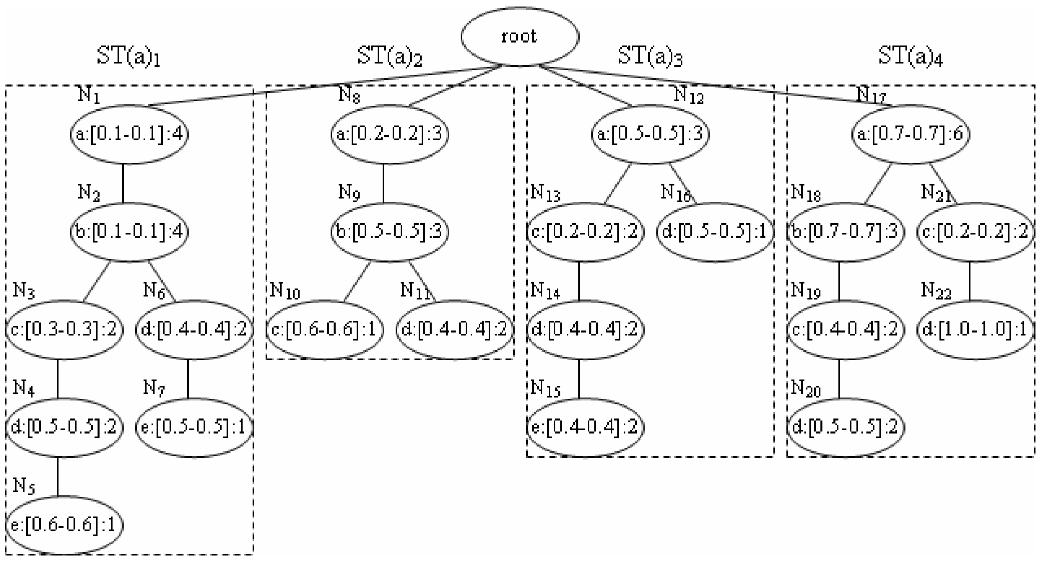


Figure 19. ST(a)

5.2.2 The UF-Prune Algorithm

With the aforementioned definitions, we present the UF-Prune algorithm for mining frequent patterns in an uncertain database by using an mUF-trie. The algorithm integrates the UF-Mine algorithm for finding out the possible frequent patterns and the UF-Reorganize algorithm for moving and merging the nodes in sub-tries iteratively. Figure 20 shows the steps of the UF-Prune algorithm.

```
UF-Prune algorithm.
Input: An mUF-trie T, a minimum support threshold $\sigma$, and a maximum merging
   threshold $\gamma$.
Output: A set of uncertain frequent patterns.

Procedure UF-Prune(T, $\sigma$, $\gamma$)
(1)    FPS = $\phi$;   // Frequent pattern set
(2)    for each item $a_i \in$ T in lexicographic order do {
(3)       generate ST($a_i$);   // Definition 17
(4)       for each ST($a_i$)$_j \in$ ST($a_i$) do   // Mine each ST($a_i$)$_j$
(5)          FPS = FPS $\cup$ UF-Mine(ST($a_i$)$_j$, null, $\sigma$);
(6)       ST($a_i$)' = UF-Reorganize(ST($a_i$), $\sigma$, $\gamma$);
(7)       if ST($a_i$)' $\neq$ ST($a_i$) then
(8)          for each ST($a_i$)$_j$' $\in$ ST($a_i$)' do {
                // Mine each ST($a_i$)$_j$' after reorganization
(9)             FPS = FPS $\cup$ UF-Mine(ST($a_i$)$_j$', null, $\sigma$);
             }
(10)      T = T prunes away the nodes corresponding to $a_i$, and ordered in
          mUF-trie-ordering;
       }
(11)   return FPS;
```

Figure 20. UF-Prune

In order to illustrate how the algorithms work, we continue to use the running example in Figure 18. Suppose the minimum support threshold is 4, and the maximum merging threshold is 0.3. UF-Prune generates ST(a), as shown in Figure 19. Then, it calls UF-Mine to mine ST(a)$_1$, ST(a)$_2$, ST(a)$_3$ and ST(a)$_4$, and UF-Mine returns a set of uncertain frequent patterns, which is {(a:[0.1-0.1])(b:[0.1-0.1]):4, (a:[0.7-0.7]):6}.

The core algorithm is the UF-Reorganize algorithm, which is shown in Figure 21. When it is called by UF-Prune with ST(a_i), it first attempts to merge the nodes corresponding to a_i, and then attempts to shuffle and merge pairs of paths under each

a_i. In the continuing example, UF-Prune calls UF-Reorganize to shuffle ST(a) in Figure 19. UF-Reorganize first attempts to merge the nodes corresponding to item a, which are N_1, N_8, N_{12} and N_{17}, and generates ST(a)', as shown in Figure 22.

UF-Reorganize algorithm.

Input: A sub-trie ST(a_i), a minimum support threshold σ, and a maximum merging threshold γ.

Output: A shuffled sub-trie ST(a_i)'.

```
Procedure UF-Reorganize(ST($a_i$), $\sigma$, $\gamma$)
        // Merge nodes corresponding to $a_i$
(1)     N_set = {$N_1$, $N_2$, …, $N_k$}, where $N_j$ contains $a_i$;
(2)     for each adjacent two nodes $N_j$ and $N_{j+1}$ in N_set do
(3)         if $N_i$ and $N_{i+1}$ are within range then {
(4)             ST($a_i$) = UF-Merge(ST($a_i$), <$N_j$>, <$N_{j+1}$>, 1);    // Merge two paths
(5)             update N_set;
            }
(6)     ST($a_i$)' = the resulted ST($a_i$) with nodes ordered in mUF-trie-ordering;
        // Shuffle and merge pairs of paths under each $a_i$
(7)     for each ST($a_i$)$_j$', where the root has sufficient support, do {
(8)         scan ST($a_i$)$_j$' once and collect P_set = the set of paths under the root;
(9)         for each pair of paths PB and PC in P_set do {
(10)            P = CAP(PB, PC);
(11)            PB = PB excludes P;
(12)            PC = PC excludes P;
(13)            if (length(CI(PB, PC)) > 0)  $\wedge$  (SC(PB, PC) > 0) then {
(14)                switch SC(PB, PC) {
(15)                    case 1: shuffle the nodes corresponding to CI(PB, PC) to their
                            MAP in PB and PC;
(16)                    case 2: shuffle the nodes corresponding to CI(PB, PC) to their
                            MAP in PB;
(17)                    case 3: shuffle the nodes corresponding to CI(PB, PC) to their
                            MAP in PC;
(18)                    case 4: do nothing;
                    }
(19)                ST($a_i$)$_j$' = UF-Merge(ST($a_i$)$_j$', PB, PC, length(CI(PB, PC)));
(20)                update P_set;    // Updates will be used in loop at line (9)
                }
            }
        }
(21)    return ST($a_i$)';
```

Figure 21. UF-Reorganize

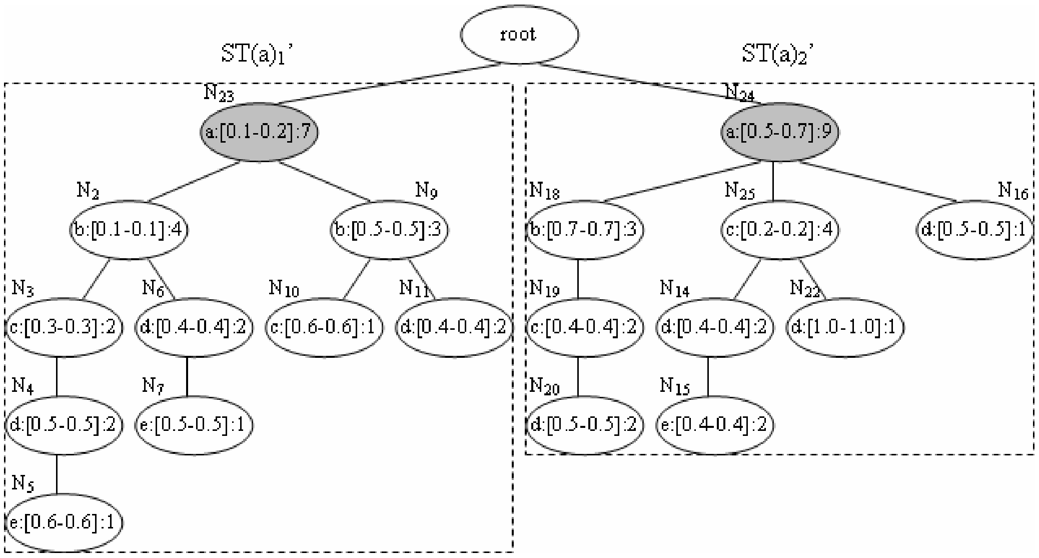


Figure 22. ST(a)'

Then, UF-Reorganize tries to shuffle and merge pairs of paths under N_{23} and N_{24} in Figure 22. UF-Reorganize shuffles the nodes in $ST(a)_1$', and $ST(a)_1$' evolves into Figure 23. The result is a pair of paths (i.e. <$N_4N_5N_3$> and <N_6N_7>) to be merged by using UF-Merge. UF-Merge merges the pair of paths, and $ST(a)_1$' evolves into Figure 24. Similarly, UF-Reorganize and UF-Merge shuffle and merge the nodes in $ST(a)_2$'.

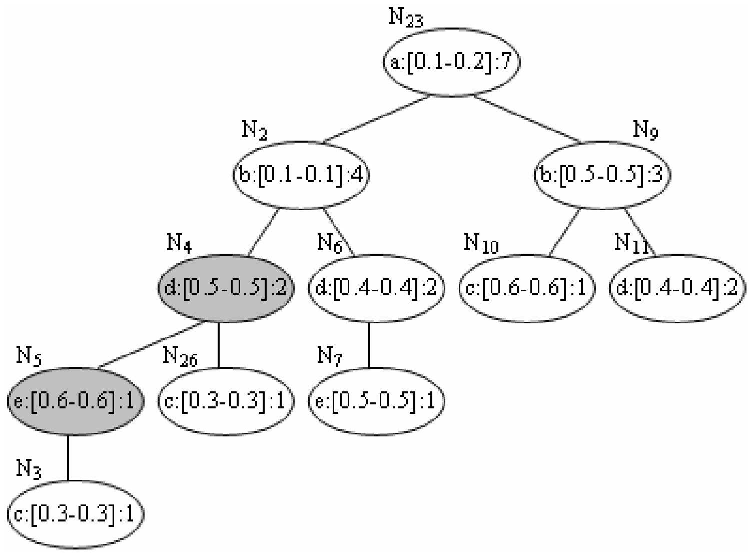


Figure 23. ST(a)$_1$' after shuffling N$_4$ and N$_5$ to their MAP

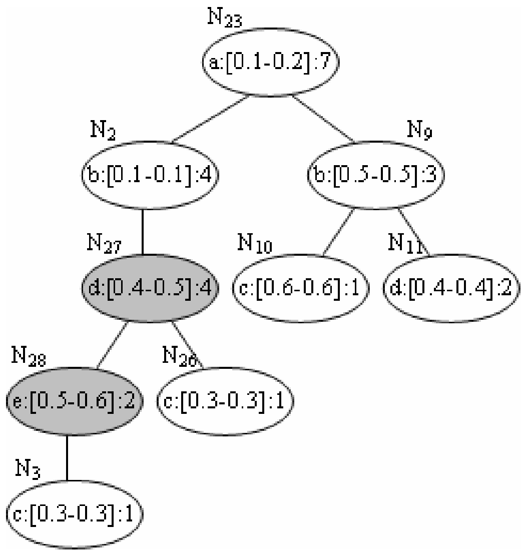


Figure 24. ST(a)$_1$' after merging N$_4$ with N$_6$ and N$_5$ with N$_7$

After that, ST(a)' evolves into Figure 25.

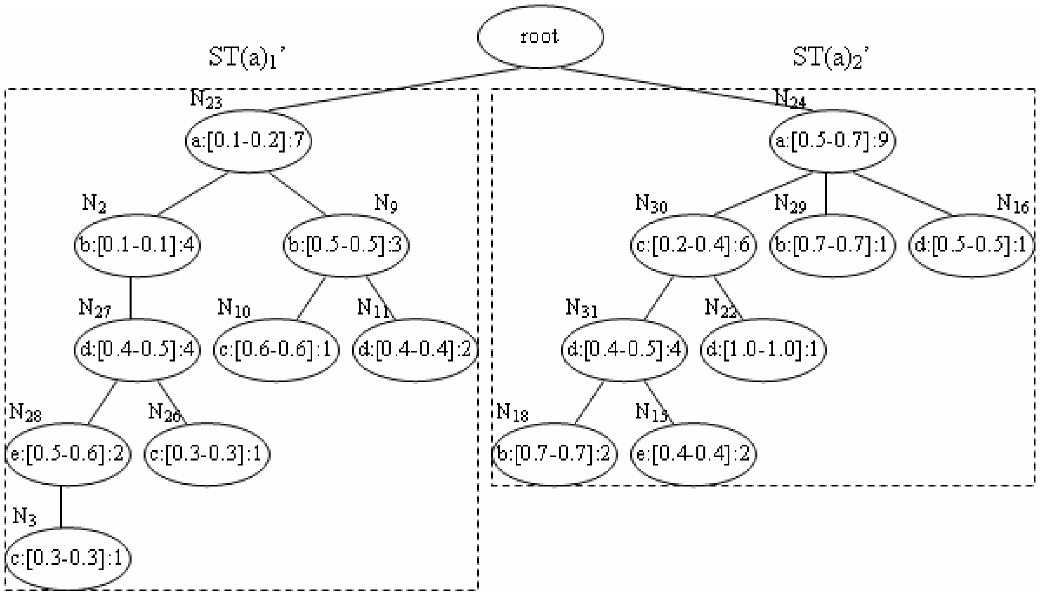


Figure 25. ST(a)' after shuffling and merging

UF-Prune calls UF-Mine to mine ST(a)$_1$' and ST(a)$_2$', and UF-Mine returns a new set of uncertain frequent patterns, which is {(a:[0.1-0.2]):7, (a:[0.1-0.2])(b:[0.1-0.1])(d:[0.4-0.5]):4, (a:[0.5-0.7]):9, (a:[0.5-0.7])(c:[0.2-0.4]):6, (a:[0.5-0.7])(c:[0.2-0.4])(d:[0.4-0.5]):4}. After processing ST(a), we have discovered uncertain frequent patterns for item a. In order not to repeat these patterns in the following mining process, UF-Prune prunes away the nodes corresponding to item a, and orders the remaining nodes in the mUF-trie-ordering. The mUF-trie evolves as shown in Figure 26.

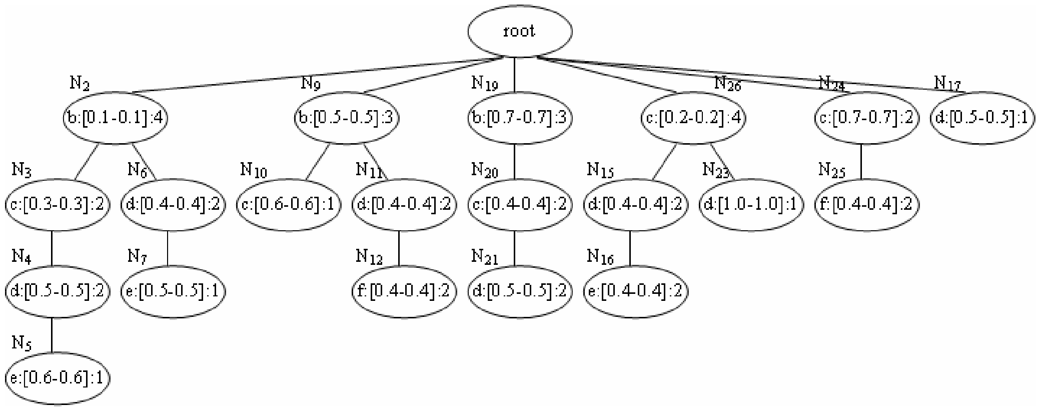


Figure 26. mUF-trie after pruning item a

UF-Prune generates ST(b), as shown in Figure 27.

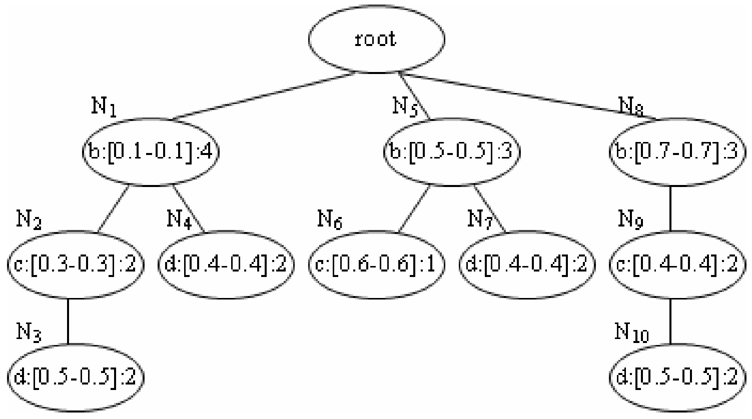


Figure 27. ST(b)

For each item, a sub-trie is generated, being mined, its nodes would then be shuffled and merged if possible, and being mined again. Finally, we get the set of uncertain frequent patterns as shown in Table 7. We have observed that UF-Prune can mine more specific uncertain frequent patterns for each item, while UF-Evolve can mine more general uncertain frequent patterns for all items, as shown in Table 5. Then, people can choose the different algorithms accordingly.

Item a_i	**Uncertain frequent patterns mined in $ST(a_i)$**	**New uncertain frequent patterns mined in $ST(a_i)'$**
a	(a:[0.1-0.1])(b:[0.1-0.1]):4 (a:[0.7-0.7]):6	(a:[0.1-0.2]):7 (a:[0.1-0.2])(b:[0.1-0.1])(d:[0.4-0.5]):4 (a:[0.5-0.7]):9 (a:[0.5-0.7])(c:[0.2-0.4]):6 (a:[0.5-0.7])(c:[0.2-0.4])(d:[0.4-0.5]):4
b	(b:[0.1-0.1]):4	(b:[0.1-0.1])(d:[0.4-0.5]):4 (b:[0.5-0.7]):6 (b:[0.5-0.7])(d:[0.4-0.5]):4
c	(c:[0.2-0.2]):4	(c:[0.2-0.4]):8 (c:[0.2-0.4])(d:[0.4-0.5]):6
d	(d:[0.4-0.4]):6 (d:[0.5-0.5]):5	(d:[0.4-0.5]):11 (d:[0.4-0.5])(e:[0.4-0.6]):4
e		(e:[0.4-0.6]):4
f	(f:[0.4-0.4]):4	

Table 7. Discovered uncertain frequent patterns with mUF-trie

Chapter 6 : Performance Study

In this chapter, we present some performance results of UF-Evolve, UF-Prune and FP-growth. Since the existing uncertain data mining methods do not have the same semantic meaning as our proposed methods, they do not merge the probabilities to obtain more frequent patterns. Some of them compute expected supports and mine frequent patterns in the form of an itemset with the expected support [14, 22, 23]. Therefore, we have not compared our methods with those. FP-growth is not for uncertain data mining, but we utilize it with some special arrangements in order to have a consistent and direct comparison. All experiments have been performed on a 2.83 GHz Xeon server with 3.00 GB of RAM, running Microsoft Windows Server 2003. The programs are written in Java. Runtime here means the total execution time, i.e. the period between input and output, instead of CPU time. Also, all runtime measurements of UF-Evolve/UF-Prune/FP-growth included the time of constructing mUF-trees/mUF-tries/FP-trees from the original databases. We have done many trials, while we have only selected those representative experiments to be included in this chapter.

6.1 Performance of UF-Evolve with FP-growth

6.1.1 Data Preparation

The experiments are done on a synthetic database (T10.I4.D3K), which is generated by using the techniques in [34]. In this database, the average record length is 10, the average length of a pattern is 4, and the number of records is 3K. Besides, we set the number of items as 1000. All the probabilities for the items are randomly generated, with a uniform distribution in the range [0, 1].

For UF-Evolve, each record in the database consists of multiple item:probability pairs; while for FP-growth, each record in the database consists of multiple items. These two representations have different semantic meanings. In order to carry out a consistent comparison of UF-Evolve with FP-growth, we make the following arrangements.

- First, a UDB is generated for UF-Evolve. Each record in UDB consists of multiple (a_i:p_i).
- Based on UDB, we generate a special database UDB' for FP-growth. Each record in UDB' consists of multiple (a_j:[l_j-u_j]), where $a_j = a_i$, and $l_j = u_j = p_i$. FP-growth treats each (a_j:[l_j-u_j]) as an item.
- While constructing an mUF-tree from UDB, we keep the Record IDs in the corresponding nodes. When UF-Evolve generates iterative versions of the mUF-tree, we record the changes of [lowerbound-upperbound] in the nodes together with their Record IDs.
- We use the Record IDs to trace back to the records in UDB' and change the corresponding l_j and u_j. Then, there will be iterative versions of UDB' for discovering new frequent patterns by FP-growth.

With these arrangements, we can have a consistent comparison for the runtime and number of mined frequent patterns of the two algorithms.

6.1.2 Experiments

In the first experiment, we measured the runtime, number of shuffles and number of mined frequent patterns with different numbers of records for UF-Evolve and FP-growth. The number of records varies from 0.5K to 3K, the minimum support threshold is 3%, and the maximum merging threshold is 0.3. The runtime results of UF-Evolve and FP-growth are shown in Figure 28. UF-Evolve is faster and more scalable than FP-growth. The number of shuffles of UF-Evolve is shown in Figure 29. Since when the number of records increased, UF-Evolve shuffled a bigger mUF-tree. The numbers of mined frequent patterns of UF-Evolve and FP-growth are shown in Figure 30. The two curves overlapped since the two algorithms mined the same numbers and the same sets of frequent patterns.

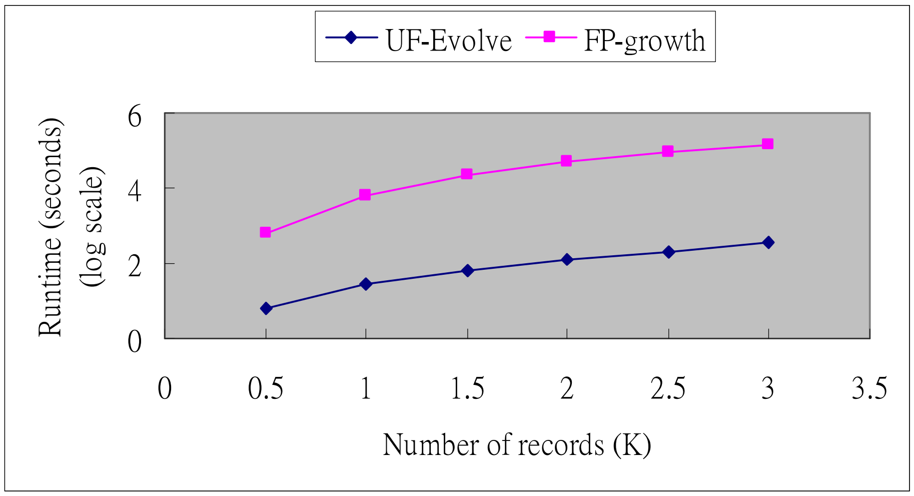


Figure 28. Runtime with number of records for UF-Evolve and FP-growth

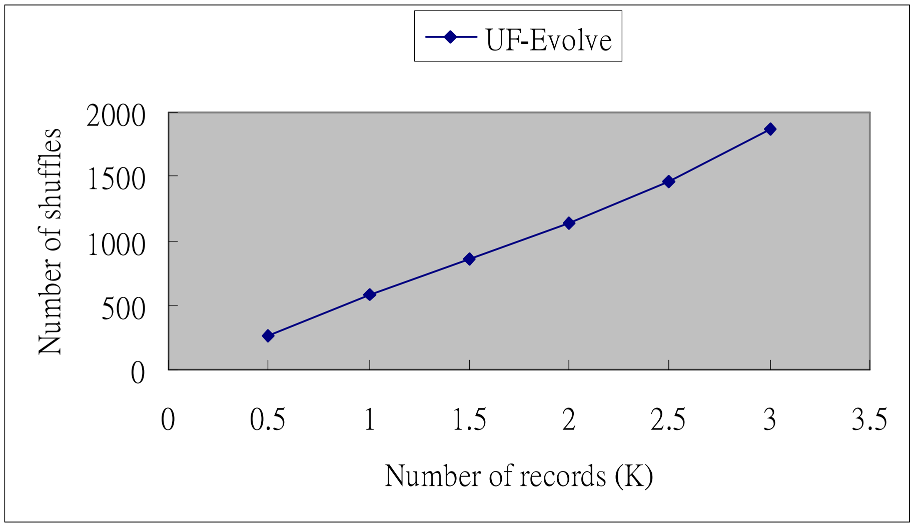


Figure 29. Number of shuffles with number of records for UF-Evolve

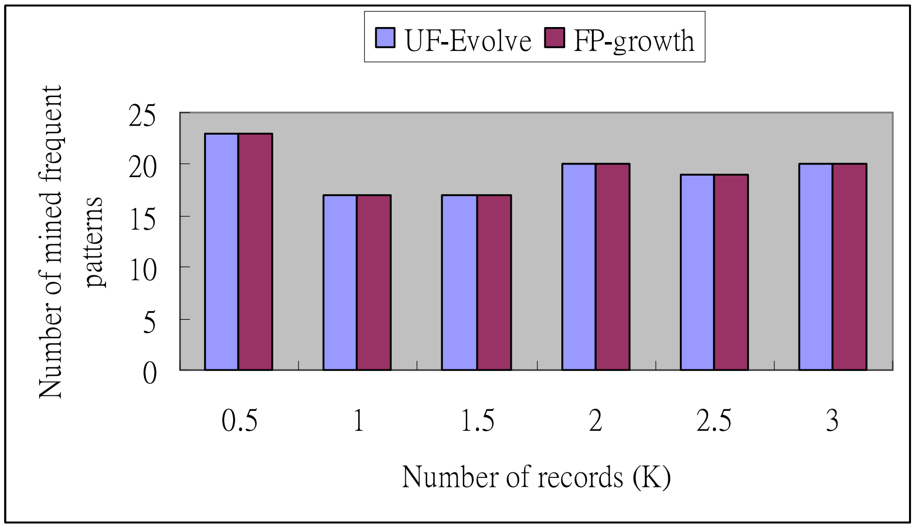


Figure 30. Number of mined frequent patterns with number of records for UF-Evolve and FP-growth

In the second experiment, we measured the runtime, number of shuffles and number of mined frequent patterns with different minimum support thresholds for UF-Evolve and FP-growth. The minimum support threshold varies from 0.1% to 8%, the number of records is 3K, and the maximum merging threshold is 0.3. The runtime results of UF-Evolve and FP-growth are shown in Figure 31. UF-Evolve is faster than FP-growth. When the minimum support threshold increased, the runtime of both UF-Evolve and FP-growth decreased. Since when the minimum support threshold is high, UF-Evolve processes fewer and smaller conditional mUF-trees. The number of shuffles of UF-Evolve is shown in Figure 32. When the minimum support threshold increased, the number of shuffles of UF-Evolve remained the same. This is because for the same database, UF-Evolve always generates and shuffles the same mUF-tree, regardless of the different minimum support thresholds. The numbers of mined frequent patterns of UF-Evolve and FP-growth are shown in Figure 33. The two curves overlapped since the two algorithms mined the same numbers and the same sets of frequent patterns. When the minimum support threshold increased, the numbers of mined frequent patterns of both UF-Evolve and FP-growth decreased. This is because when the minimum support threshold is high, the frequent patterns are short and the set of such patterns is not large.

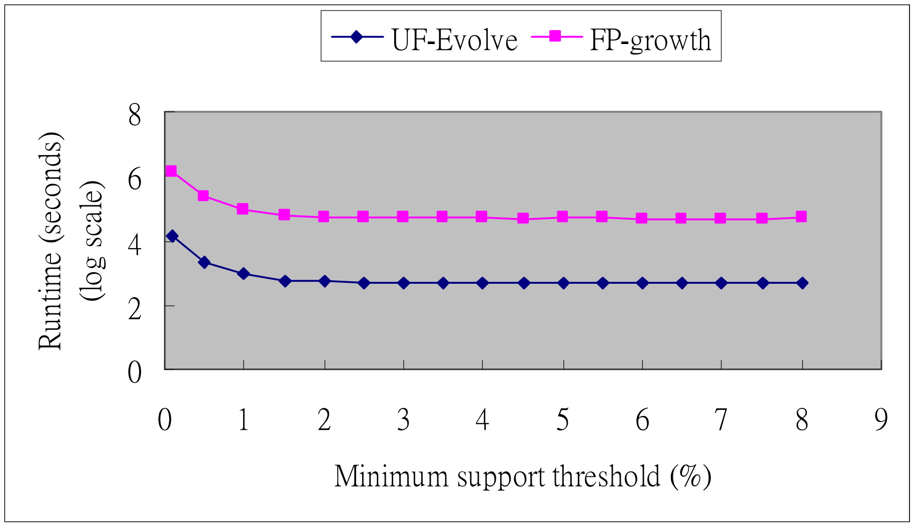


Figure 31. Runtime with minimum support threshold for UF-Evolve and FP-growth

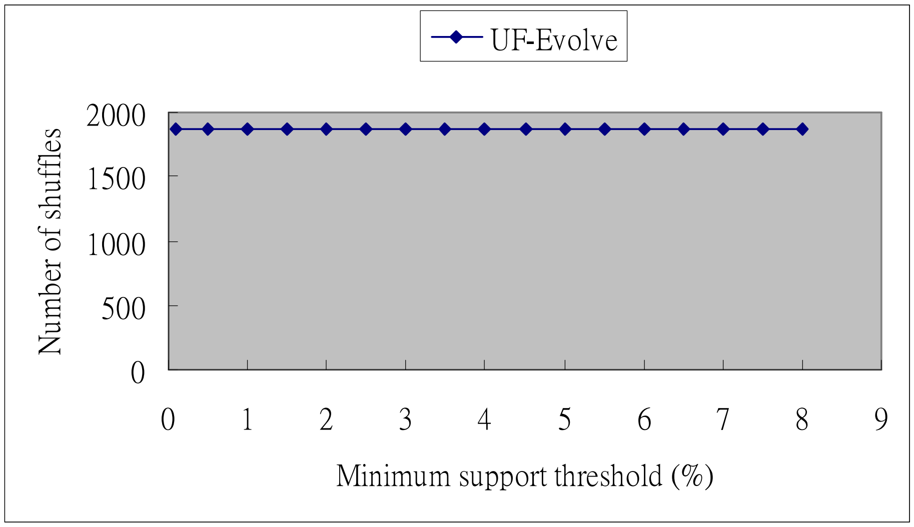


Figure 32. Number of shuffles with minimum support threshold for UF-Evolve

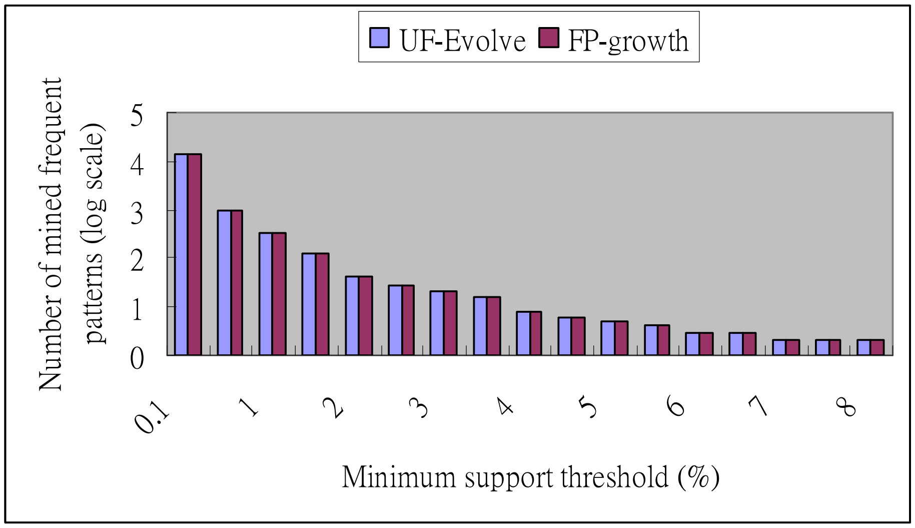


Figure 33. Number of mined frequent patterns with minimum support threshold for UF-Evolve and FP-growth

In the third experiment, we measured the number of mined frequent patterns in each iteration for UF-Evolve. The number of records is 3K, the minimum support threshold is 0.1%, and the maximum merging threshold is 0.3. As shown in Figure 34, the UF-Evolve algorithm discovered new frequent patterns from iterative versions of mUF-tree, and the number of mined frequent patterns kept increasing until stabilized.

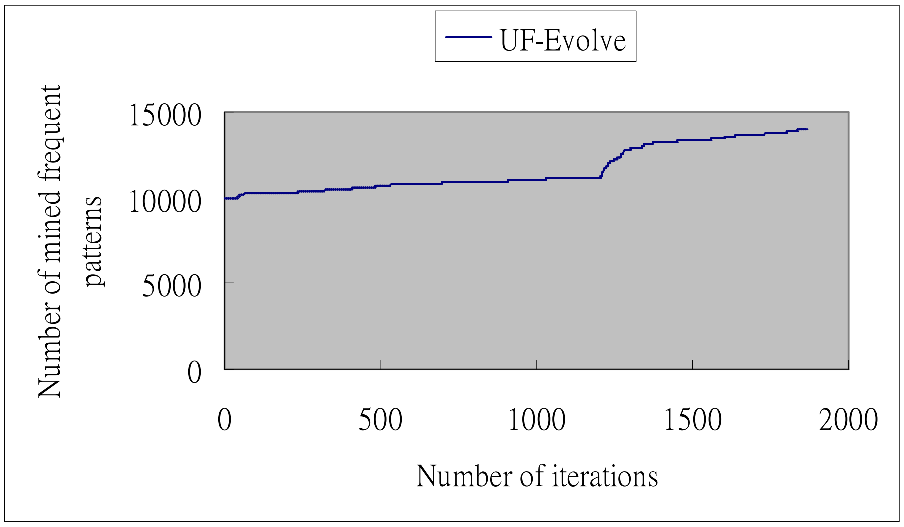


Figure 34. Number of mined frequent patterns in each iteration for UF-Evolve

For the above experiments, UF-Evolve and FP-growth mined the same numbers and the same sets of frequent patterns. Some of the discovered frequent patterns are shown in Table 8. Here, the supports for frequent patterns are shown as (support count)/(number of records in UDB) in percentage.

Frequent patterns mined in 1st iteration	**New frequent patterns mined in 2nd iteration**	**New frequent patterns mined in 3rd iteration**
(59757:[0.3-0.3]):4%	(29340:[0.1-0.3]):3%	(45973:[0.2-0.3]):4%
(45370:[0.7-0.7]):5%	(59757:[0.3-0.4])	(38212:[0.8-0.8])
…	(22360:[0.3-0.3])	(8885:[0.2-0.5])
	(18474:[0.5-0.7])	(45973:[0.2-0.3]):4%
	(29340:[0.1-0.3]):3%	…
	…	

Table 8. Discovered frequent patterns for UF-Evolve and FP-growth

6.2 Performance of UF-Prune with UF-Evolve

6.2.1 Data Preparation

The experiments are done on a synthetic database (T10.I4.D15K), which is generated by using the techniques in [34]. In this database, the average record length is 10, and the average length of a pattern is 4. We generated the number of records as 15K, which is different from that in Section 6.1.1. Since UF-Evolve and FP-growth have long execution time, we selected up to 3K records for these experiments. Also, we use smaller databases of up to 3K records for comparing UF-Prune with UF-Evolve, but we can use larger databases of up to 15K records for testing UF-Prune. The number of items is stayed the same as 1000. Each record in the database consists of multiple item:probability pairs. All the probabilities for the items are randomly generated, with a uniform distribution in the range [0, 1]. We will have a consistent comparison for the runtime, number of shuffles and number of mined frequent patterns for the two algorithms.

6.2.2 Experiments

In the first experiment, we measured the runtime, number of shuffles and number of mined frequent patterns with different numbers of records for UF-Prune and UF-Evolve. The number of records varies from 0.5K to 3K, the minimum support threshold is 3%, and the maximum merging threshold is 0.3. The runtime results of UF-Prune and UF-Evolve are shown in Figure 35. UF-Prune is faster and more scalable than UF-Evolve. The numbers of shuffles of UF-Prune and UF-Evolve are shown in Figure 36. UF-Prune is expected to shuffle more than UF-Evolve. Note that UF-Evolve shuffles the whole mUF-tree. The candidate paths are long and the shuffle cases 1, 2, 3 and 4 limit the possible number of shuffles. On the other hand, UF-Prune shuffles small sub-tries. The candidate paths would be short and the shuffle cases would have less limitation. The numbers of mined frequent patterns of UF-Prune and UF-Evolve are shown in Figure 37. UF-Prune discovered more frequent patterns than UF-Evolve.

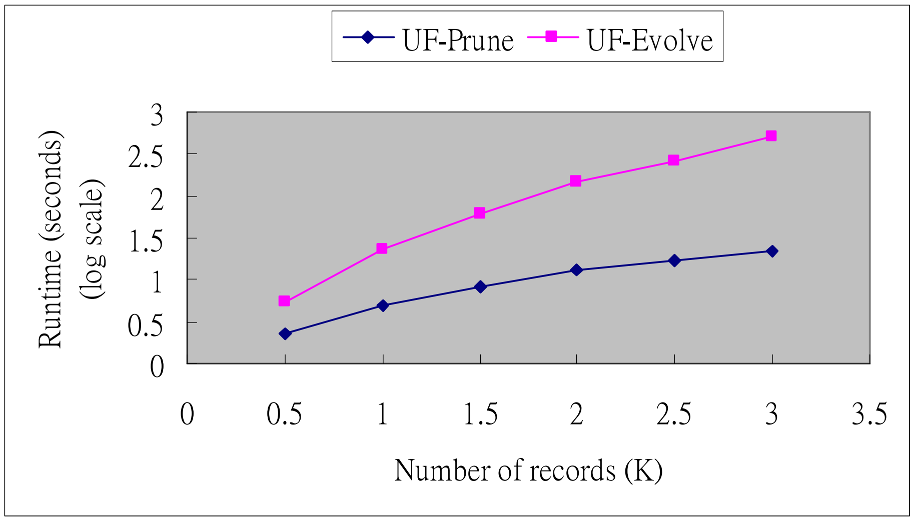


Figure 35. Runtime with number of records for UF-Prune and UF-Evolve

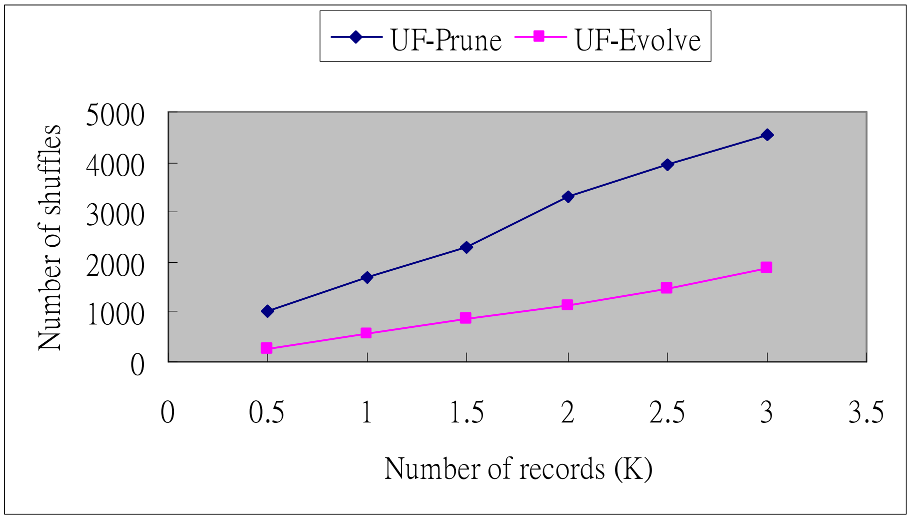


Figure 36. Number of shuffles with number of records for UF-Prune and UF-Evolve

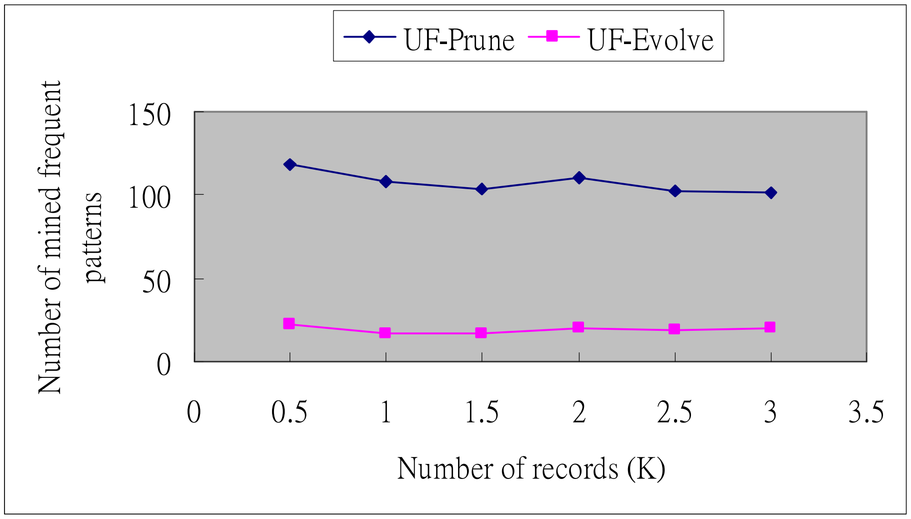


Figure 37. Number of mined frequent patterns with number of records for UF-Prune and UF-Evolve

Since the runtime of UF-Evolve is increasing exponentially, when the number of records becomes large, UF-Evolve is extremely slow. However, UF-Prune is much faster. We tested it with up to 15K records, the minimum support threshold is 3%, and the maximum merging threshold is 0.3. As shown in Figure 38, the number of shuffles of UF-Prune has been increasing linearly. Since when the database size increased, there would be more different probability values for the same item, the mUF-trie became larger, and the number of shuffles also increased.

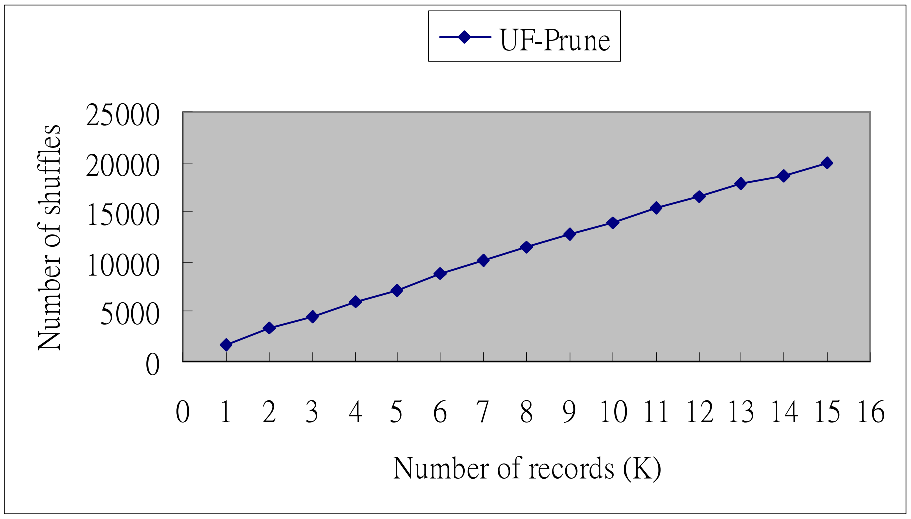


Figure 38. Number of shuffles with number of records for UF-Prune

In the second experiment, we measured the runtime, number of shuffles and number of mined frequent patterns with different minimum support thresholds for UF-Prune and UF-Evolve. The minimum support threshold varies from 0.1% to 8%, the number of records is 3K, and the maximum merging threshold is 0.3. The runtime results of UF-Prune and UF-Evolve are shown in Figure 39. UF-Prune is faster than UF-Evolve. When the minimum support threshold increased, the runtime of both UF-Prune and UF-Evolve decreased. Since when the minimum support threshold is high, UF-Prune constructs and processes a small mUF-trie. The numbers of shuffles of UF-Prune and UF-Evolve are shown in Figure 40. When the minimum support threshold is low, UF-Prune shuffles more than UF-Evolve. When the minimum support threshold is high, UF-Prune shuffles fewer times than UF-Evolve. When the minimum support threshold increased, the number of shuffles of UF-Prune decreased. On the other hand, UF-Evolve remained the same under the same condition. This is because when the minimum support threshold is high, UF-Prune shuffles small sub-tries. However, for the same database, UF-Evolve always generates and shuffles the same mUF-tree, regardless of the different minimum support thresholds. The numbers of mined frequent patterns of UF-Prune and UF-Evolve are shown in Figure 41. Most of the time, UF-Prune discovered more frequent patterns than UF-Evolve. When the minimum support threshold increased, the numbers of mined frequent patterns of both UF-Prune and UF-Evolve decreased. This is because when the minimum support threshold is high, the frequent patterns are short and the set of such patterns is not large.

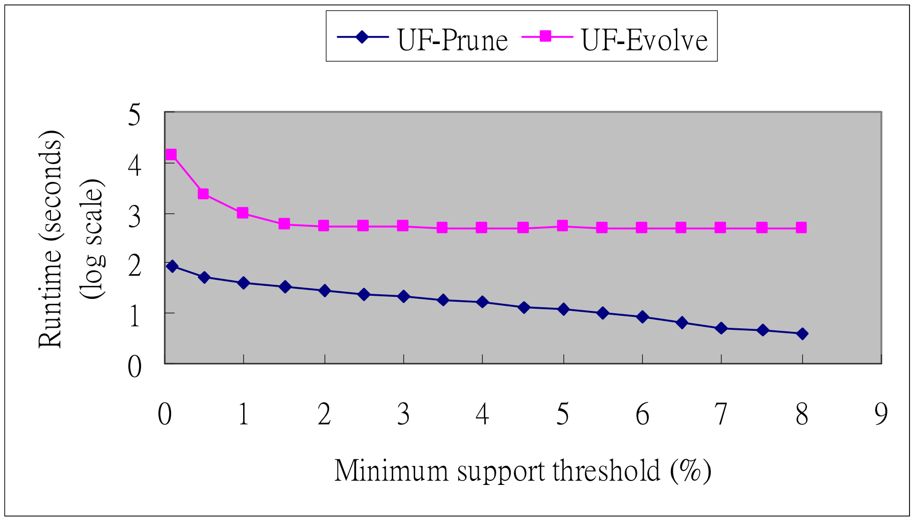


Figure 39. Runtime with minimum support threshold for UF-Prune and UF-Evolve

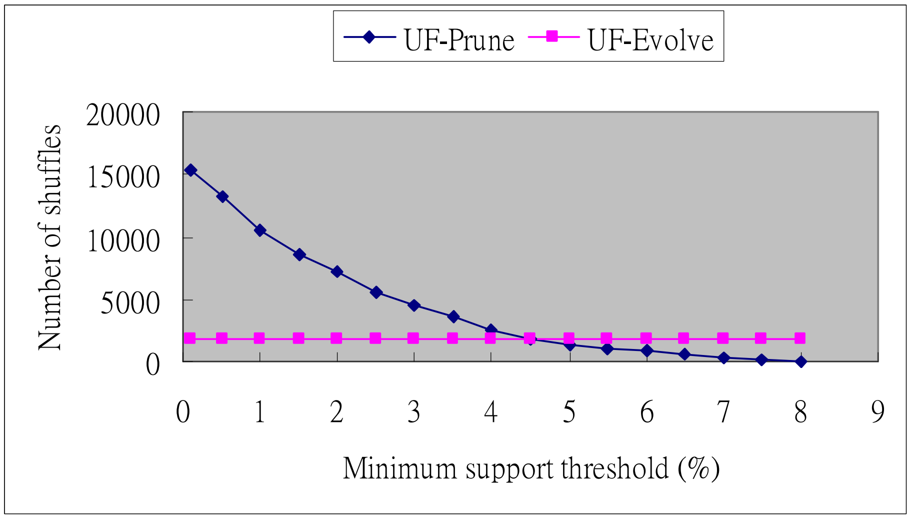


Figure 40. Number of shuffles with minimum support threshold for UF-Prune and UF-Evolve

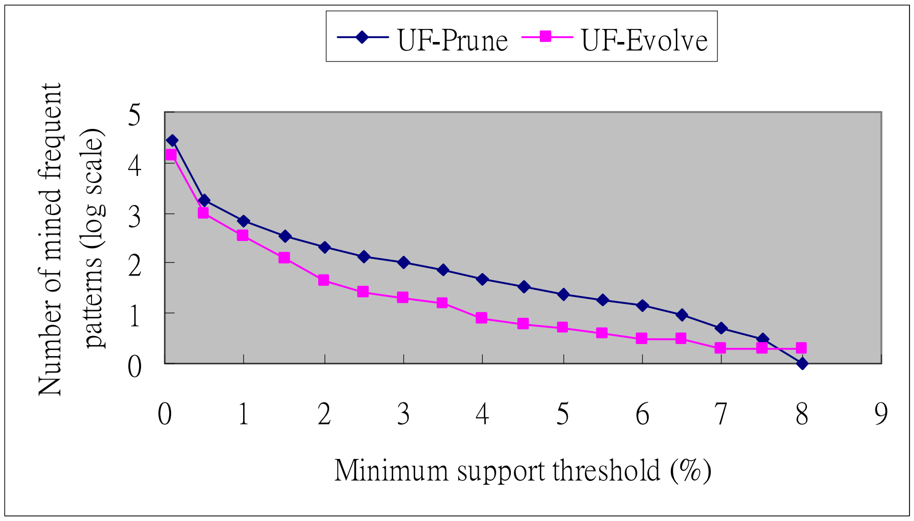


Figure 41. Number of mined frequent patterns with minimum support threshold for UF-Prune and UF-Evolve

In the third experiment, we measured the number of mined frequent patterns for UF-Evolve, UF-Prune and the complete set. The number of records is 100, the minimum support threshold is 5%, and the maximum merging threshold is 0.3. As shown in Figure 42, UF-Prune mined more frequent patterns than UF-Evolve, but both of them could not mine the complete set of frequent patterns. Through this experiment, we can estimate that around 15% of the frequent patterns have been missed for UF-Evolve and UF-Prune.

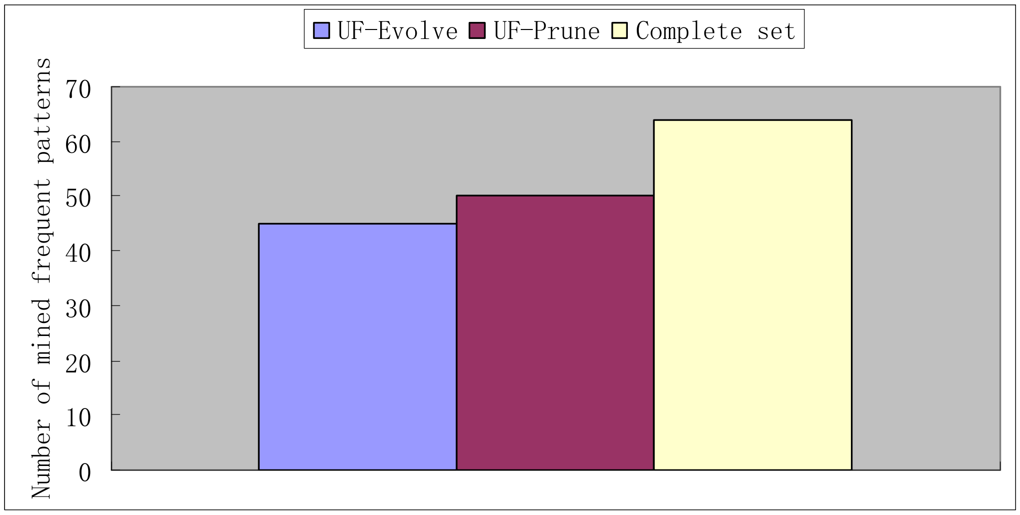


Figure 42. Number of mined frequent patterns for UF-Evolve, UF-Prune and complete set

For the above experiments, some of the discovered frequent patterns for UF-Prune are shown in Table 9. Here, the supports for frequent patterns are shown as (support count)/(number of records in UDB) in percentage.

Item a_i	**Frequent patterns mined in ST(a_i)**	**New frequent patterns mined in ST(a_i)'**
29340	(29340:[0.1-0.1]):3%	(18474:[0.5-0.7]) (22360:[0.3-0.3]) (29340:[0.1-0.3]) (59757:[0.3-0.4]):3%
45370	(45370:[0.7-0.7]):5%	
45973	(45973:[0.2-0.2]):4%	(38212:[0.8-0.8]) (45973:[0.2-0.3]) (8885:[0.2-0.5]):4%

59757	(59757:[0.3-0.3]):4%	
…	…	…

Table 9. Discovered frequent patterns for UF-Prune

After all the experiments, we conclude that UF-Evolve is faster and more scalable than FP-growth, while UF-Prune is even faster and more scalable than UF-Evolve. UF-Evolve/UF-Prune discovered new frequent patterns from iterative versions of mUF-tree/sub-trie, and the number of mined frequent patterns kept increasing until stabilized. Most of the time, UF-Prune discovered more frequent patterns than UF-Evolve.

Chapter 7 : Application and Extended Work

7.1 Uncertain Frequent Pattern Mining for Rural Data

The mUF-tree and mUF-trie can be applied to different applications. Here, we describe its use on uncertain rural data. Nowadays, many farmers grow crops which can gain better profits and are more concerned about the productivity and sustainability. It implies that different environmental and control factors, such as temperature, humidity, rainfall, pests, defects and soil salinity should be better controlled or monitored. If they can find out special patterns relating to productivity and sustainability, especially for valuable crops such as grapes in a vineyard, profitability or environmental gain would be improved. Also, it is getting common that a farm is divided into different blocks and sensors are installed (or people are employed) for monitoring. However, due to cost and sensor reliability, the readings may not be accurate and the collected data would be uncertain overtime. This type of data uncertainty is value uncertainty as compared with existential uncertainty in our previous chapters.

Adinarayana et al. [1] proposed an integration of Geo-ICT (location based services, spatial decision making and geo-computations) and Sensor Network (distributed sensing units pertaining to weather, crop and soil parameters under micro-climatic conditions) in agricultural systems. This helps farmers to combat inclement climate conditions and the global climate changes.

In value uncertainty, an item can be modeled as a closed value range [x_1-x_2] which bounds its possible values. A Normalized Value Range (NVR) can be used to represent the uncertain data during a collection period. Suppose each factor has a normal range [lb-ub], then a value range [x_1-x_2] of the sensor device will be casted to an NVR $[\frac{x_1 - lb}{ub - lb} - \frac{x_2 - lb}{ub - lb}]$. For example, assume that the normal range of a temperature reading is [–10°C-40°C], then a value range [10°C-20°C] of the temperature readings will be casted to an NVR [0.4-0.6] (i.e. $[\frac{10-(-10)}{40-(-10)} - \frac{20-(-10)}{40-(-10)}]$). If R_i represents a daily record of some sensors, then each item within R_i represents a factor and is associated with an NVR expressing the range of the readings of the corresponding sensor after normalization. For instance, in R_i, it can have an NVR [0.4-0.6] of the temperature, and an NVR [0.1-0.2] of the pests.

In an uncertain database on sensor readings, each record consists of multiple (a_i:[p_{i1}-p_{i2}]), where a_i is an item with [p_{i1}-p_{i2}] as the NVR. An example uncertain database on sensor readings is shown in Table 10, where a = temperature, b = humidity, c = rainfall, d = pests, e = defects and f = soil salinity.

Record ID	**Item:NVR pairs**
1	(d:[0.4-0.5]), (a:[0.1-0.4]), (e:[0.5-0.8])
2	(a:[0.1-0.4]), (d:[0.5-0.7]), (c:[0.3-0.5]), (e:[0.6-0.8])
3	(c:[0.3-0.5]), (a:[0.1-0.4]), (d:[0.5-0.7])
4	(a:[0.1-0.4])
5	(b:[0.2-0.4]), (d:[0.4-0.5]), (c:[0.3-0.5])
6	(f:[0.6-0.9]), (b:[0.2-0.4])
7	(b:[0.2-0.4])
8	(b:[0.2-0.4])

Table 10. An uncertain database on sensor readings

We can easily map the NVRs and directly store in a UDB with the lowerbound and upperbound information. The UDB would be the same structure as we discussed in Chapter 3. Then, mUF-tree and mUF-trie together with their respective algorithms can be applied for the pattern discovery.

Suppose we take the uncertain data in Table 10 as an example. Let the minimum support threshold be 3, and the maximum merging threshold be 0.3. With the mUF-tree and UF-Evolve, the set of uncertain frequent patterns is {(c:[0.3-0.5]):3, (b:[0.2-0.4]):4, (a:[0.1-0.4]):4, (d:[0.4-0.7]):3, (a:[0.1-0.4])(d:[0.4-0.7]):3}. For interpretation, the discovered pattern (a:[0.1-0.4])(d:[0.4-0.7]):3 means that when the temperature is between –5^{o}C to 10^{o}C, 40% to 70% of the plants have pests on them.

7.2 Incremental Uncertain Frequent Pattern Mining

In real-world applications, uncertain databases usually grow over time. When an incremental database udb is added to an original database UDB, we get the updated database {UDB $\cup$ udb}. The initial mUF-tree/mUF-trie constructed from UDB need to be updated. Some algorithms solve this problem by reprocessing {UDB $\cup$ udb} to reconstruct an mUF-tree/mUF-trie, which is time-consuming and wastes the initial mUF-tree/mUF-trie. Inspired by [9, 17], we propose to keep the initial mUF-tree/mUF-trie before the first iteration and update it with the incremental database. Due to the different characteristics between the mUF-tree and mUF-trie, there are two different ways to update the mUF-tree and mUF-trie.

The initial mUF-tree already contains the information about all items in UDB. After udb is added to UDB, some new items may arise and be included in the mUF-tree. Therefore, udb will be processed.

The initial mUF-trie only contains the information about frequent items in UDB.

After udb is added to UDB, some new frequent items may arise and be included in the mUF-trie, and some old frequent items may become infrequent and be removed from the mUF-trie. Therefore, udb will be processed, and if necessary, UDB will be reprocessed.

After the initial mUF-tree/mUF-trie has been updated, some existing uncertain frequent patterns may no longer exist, while some new patterns may be discovered. When considering incremental mining, it would be better to use the minimum support threshold in percentage instead of support counts. In the next two sections, we will consider the minimum support threshold in percentage.

7.2.1 Incrementally Updating of mUF-tree

The UF-Update algorithm is used to incrementally update the initial mUF-tree with udb. UF-Update scans udb without specifying a minimum support threshold. The updated mUF-tree contains the information about all items in udb. Since the initial mUF-tree already contains the information about all items in UDB, there is no need to reprocess UDB. Figure 43 shows the steps of the UF-Update algorithm.

```
UF-Update algorithm.
Input: An mUF-tree T, an incremental uncertain database udb, and a_set, the set of
    item:probability pairs in T with their respective supports in descending order.
Output: An updated mUF-tree T.

Procedure UF-Update(T, udb, a_set)
(1)    scan udb once and collect a_set' = the set of item:proabaility pairs with their
           respective supports;
(2)    remove the items appearing in a_set from a_set';
(3)    sort item:probability pairs in a_set' with their respective supports in
           descending order;
(4)    append a_set' to the end of a_set;
(5)    for each record R in udb do {
           // Same as lines (4) - (14) of Figure 3
       }
(6)    return T;
```

Figure 43. UF-Update

For lines (1) - (4) of Figure 43, in order not to distort the order of item:proabaility pairs in the initial mUF-tree, UF-Update keeps the order of item:probability pairs in a_set and appends new item:probability pairs arising from udb to the end of a_set.

7.2.2 Incrementally Updating of mUF-trie

With the minimum support threshold in percentage ξ, there are four cases for an item a_i in UDB and udb. In Case 1, a_i is frequent in UDB and also frequent in udb. Therefore, in {UDB $\cup$ udb}, a_i.support $\geq$ $|\{UDB \cup udb\}| \times \xi$, and a_i is always frequent. Since a_i is already included in the initial mUF-trie, its support will be updated by processing udb. In Case 2, a_i is frequent in UDB but infrequent in udb. It may be frequent in {UDB $\cup$ udb}. Since a_i is already included in the initial mUF-trie, this is to be determined by processing udb. In Case 3, a_i is infrequent in UDB but frequent in udb. It may be frequent in {UDB $\cup$ udb}. Since a_i is not included in the initial mUF-trie, this is to be determined by processing udb and reprocessing UDB. In Case 4, a_i is infrequent in UDB and also infrequent in udb. Therefore, in {UDB $\cup$ udb}, a_i.support < $|\{UDB \cup udb\}| \times \xi$, and a_i in always infrequent. The four cases are shown in Table 11.

udb / UDB	a_i.support $\geq$ $\|udb\| \times \xi$	a_i.support < $\|udb\| \times \xi$
a_i.support $\geq$ $\|UDB\| \times \xi$	Case 1: Frequent	Case 2: To be determined
a_i.support < $\|UDB\| \times \xi$	Case 3: To be determined	Case 4: Infrequent

Table 11. Four cases for a_i in UDB and udb with minimum support threshold in percentage

Given a minimum support threshold in percentage, the UF-Increment algorithm is used to incrementally update the initial mUF-trie with udb. UF-Increment scans udb once and collects only frequent items in udb. Then, UF-Increment processes udb as in Case 1, Case 2 and Case 3, and reprocesses UDB as in Case 3 if necessary. After possible pruning, the updated mUF-trie contains the information about frequent items in {UDB $\cup$ udb}, and the items are ordered in mUF-trie-ordering. Figure 44 shows the steps of the UF-Increment algorithm.

UF-Increment algorithm.

Input: An original uncertain database UDB represented by an mUF-trie T, an incremental uncertain database udb, a minimum support threshold in percentage ξ, and a_set, the set of items in T with their respective supports $\geq$ |UDB| $\times \xi$.

Output: An updated mUF-trie T.

```
Procedure UF-Increment(UDB, T, udb, $\xi$, a_set)
(1)  scan udb once and collect a_set' = the set of items with their respective
       supports $\geq$ |udb| $\times \xi$;
(2)  merge a_set' into a_set;
     // Process udb as in Case 1, Case 2 and Case 3 of Table 11
(3)  for each record R in udb do {
       // Same as lines (4) - (14) of Figure 17
     }
(4)  if a_set has been modified then {
       // Reprocess UDB as in Case 3 of Table 11
(5)    for each record R in UDB do {
         // Same as lines (4) - (8) of Figure 17
(6)      if $\exists$ $N_c$ then do nothing;
         // Same as lines (10) - (14) of Figure 17
       }
     }
(7)  T = T prunes away the nodes corresponding to the items with their
       respective supports < |(UDB $\cup$ udb)| $\times \xi$, and ordered in
       mUF-trie-ordering;
(8)  return T;
```

Figure 44. UF-Increment

Chapter 8 : Conclusions and Suggestions for Future Research

8.1 Conclusions

In the thesis, we have developed two solutions for iterative uncertain frequent pattern mining with mUF-tree and mUF-trie. The first contribution is the proposal of the mUF-tree structure, which is a novel uncertain-frequent-pattern discover structure. The core idea is to allow the merging data of the same items but with small differences in their existence probabilities. Its related UF-Evolve algorithm can utilize the shuffling and merging techniques to iteratively discover new uncertain frequent patterns. With the mUF-tree, patterns not discovered before because of its insufficient support counts would now appear. This enhances users in finding interesting patterns amongst uncertain data.

The mUF-tree requires much computational effort. Hence, the next contribution of our work is the enhancement of the mUF-tree to mUF-trie structure. Uncertain information is arranged in the lexicographic order to reduce the shuffling and merging effort. Together with the UF-Prune algorithm, it can continuously generate a sub-trie for each item. After performing the shuffling and merging steps on the sub-trie, the processed item would be pruned away.

For the shuffling and merging techniques in the UF-Evolve and UF-Prune algorithms, we find that there are two cases for shuffling a node with its parent node and five cases for shuffling a pair of paths. We also showed the detailed steps of merging two nodes to form a new one. The maximum merging threshold can be defined by users. A different user can define different maximum merging thresholds for different items, and for different probability intervals of the same item to deal with different applications in uncertain frequent pattern mining.

A set of the preliminary experiments have been conducted to compare the performance of UF-Evolve, UF-Prune and FP-growth. It shows that the UF-Evolve and UF-Prune algorithms are efficient and scalable for mining additional uncertain frequent patterns with different sizes of uncertain databases and with different minimum support thresholds, but UF-Prune with mUF-trie is faster and more scalable than UF-Evolve with mUF-tree. The number of shuffles of UF-Prune depends on the size of the uncertain database and the minimum support threshold; while that of UF-Evolve depends on the size of the uncertain database only. Also, UF-Prune can discover more frequent patterns than UF-Evolve most of the time.

Our idea does not limit to the aforementioned applications. It can be transformed for another possible representation of uncertain information and be applied to discover uncertain patterns of environmental and control factors for improving crop growing in rural systems. Towards the end of our work, we have attempted to design approaches to solve the incremental uncertain frequent pattern mining problem with the two new structures, mUF-tree and mUF-trie.

8.2 Suggestions for Future Research

The work presented in the thesis is only a starting point of other potential related research work. There are certainly different possible extension and improvement of our work. First, the incremental uncertain frequent pattern mining with mUF-tree and mUF-trie should be implemented and reviewed. The UF-Update/UF-Increment algorithm will be implemented to update the mUF-tree/mUF-trie with an incremental uncertain database. After updating, we want to maintain the structure which facilitates the shuffling, merging and mining. It would help to verify the ideas and performance of the algorithms.

Second, we are interested to extend our work to a distributed environment for discovering uncertain frequent patterns between different sites or nodes. One needs to consider how to shuffle and merge trees from different nodes in the distributed sites. If messages are to be sent for shuffling and merging instructions, the question is a good design of the mUF-tree/mUF-trie information. The next question is the coordination of the messages as well as the local or global pattern discovery step afterwards. The number of messages to be passed is to be considered and the relationships between locally frequent and globally frequent patterns are to be studied.

Another extension of our work is to adopt the mUF-tree and mUF-trie to represent other types of uncertain data, such as heterogeneous uncertain streaming XML data in publish/subscribe systems. For example, given an Uncertain XML Document (XMLD), we can use a tree structure (TreeD) to store the information in XMLD. Upon receiving an XML Query (XMLQ), we can use another tree structure (TreeQ) to store the information in XMLQ. The tree structure facilitates transforming TreeD to match against TreeQ, traversing the transformed TreeD, getting the nodes in the transformed TreeD in a specific order, and generating a result for the query. An interesting issue is when the query is for the discovery of some uncertain patterns with a set of given constraints, how to represent the XML data and the query with good querying performance would be a challenging task.

Also, we can consider our work for quantitative association rule mining. The mined frequent patterns in our work are similar to quantitative association rules. Note that the databases would be different. We will consider revising our algorithms for mining quantitative association rules.

Last but not least, it is important to further improve the performance of the UF-Evolve and UF-Prune algorithms, study their performance in different running conditions, and decide a way to ensure obtaining “good” uncertain frequent patterns. For the runtime perspective, we should improve the shuffling procedures in the UF-Shuffle and UF-Reorganize algorithms to make them execute faster. For the memory space consideration, we should improve the various data structures used in the algorithms to allow bigger databases and efficient use of storage space. In the preliminary experiments, we have studied the performance of the algorithms in various running conditions, i.e. different numbers of records, different minimum support thresholds, different numbers of iterations, and randomly generated probabilities for the items. We also plan to study their performance in different running conditions, such as different maximum merging thresholds, and different probability generation methods for the items. So far, we have not decided the way to measure “good” uncertain frequent patterns for uncertain databases. We suggest deciding a quality measure, which includes the requirements, such as pattern length, common item length and shuffle case. Then we can mine a possible set of “good” uncertain frequent patterns from an uncertain database.

References

[1] Adinarayana, J., Sudharsan, D., Tripathy, A. K., Arun, J., Desai, U. B., Merchant, S. N., Naveen, C., Ashwani, R., Das, I., Ninomiya, S., Hirafuji, M., Kiura, T., Tanaka, K., and Fukatsu, T. *Towards Integration of Geo-ICT and Sensor Network in Agri-Systems.* INSAIT-II National Conference on Agro-Informatics and Precision Farming, 02-03.12.2009, Raichur, India, http://www.acr.edu.in/info/infofile/147.pdf.

[2] Adnan, M., Alhajj, R., and Barker, K. *Constructing Complete FP-Tree for Incremental Mining of Frequent Patterns in Dynamic Databases.* M. Ali and R. Dapoigny (Eds.): IEA/AIE 2006, LNAI 4031, pp. 363-372, 2006.

[3] Aggarwal, C. C., and Li, Y. *Frequent Pattern Mining with Uncertain Data.* KDD’09, June 28-July 1, 2009, Paris, France, pp. 29-37.

[4] Agrawal, R., and Srikant, R. *Fast Algorithms for Mining Association Rules.* Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994, pp. 487-499.

[5] Antova, L., Jansen, T., Koch, C., and Olteanu, D. *Fast and Simple Relational Processing of Uncertain Data.* ICDE 2008, pp. 983-992.

[6] Antova, L., Koch, C., and Olteanu, D. *From Complete to Incomplete Information and Back.* SIGMOD’07, June 12-14, 2007, Beijing, China, pp. 713-724.

[7] Bodon, F. *A Trie-based APRIORI Implementation for Mining Frequent Item Sequences.* OSDM’05, August 21, 2005, Chicago, Illinois, USA, pp. 56-65.

[8] Borgelt, C. *An Implementation of the FP-growth Algorithm.* OSDM’05, August 21, 2005, Chicago, Illinois, USA, pp. 1-5.

[9] Chang, C. C., Li, Y. C., and Lee, J. S. *An Efficient Algorithm for Incremental Mining of Association Rules.* Proceedings of the 15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications (RIDE-SDMA'05), pp. 3-10, 2005.

[10] Chau, M., Cheng, R., and Kao, B. *Uncertain Data Mining: A New Research Direction.* Proceedings of the Workshop on the Sciences of the Artificial, Hualien, Taiwan, December 7-8, 2005, pp. 1-8.

[11] Cheung, D. W., Han, J., Ng, V. T., Fu, A. W., and Fu, Y. *A Fast Distributed Algorithm for Mining Association Rules.* 1996 IEEE, pp. 31-42.

[12] Cheung, D. W., Lee, S. D., and Kao, B. *A General Incremental Technique for Maintaining Discovered Association Rules.* Proceedings of the Fifth International Conference on Database Systems for Advanced Applications, Melbourne, Australia, April 1-4, 1997, pp. 185-194.

[13] Cheung, W., and Zaiane, O. R. *Incremental Mining of Frequent Patterns Without Candidate Generation or Support Constraint.* Proceedings of the Seventh International Database Engineering and Applications Symposium (IDEAS'03), pp. 111-116, 2003.

[14] Chui, C. K., Kao, B., and Hung, E. *Mining Frequent Itemsets from Uncertain Data.* Z.-H. Zhou, H. Li, and Q. Yang (Eds.): PAKDD 2007, LNAI 4426, pp. 47-58, 2007.

[15] Ezeife, C. I., and Su, Y. *Mining Incremental Association Rules with Generalized FP-Tree.* R. Cohen and B. Spencer (Eds.): AI 2002, LNAI 2338, pp. 147-160, 2002.

[16] Han, J., Pei, J., Yin, Y., and Mao, R. *Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach.* Data Mining and Knowledge Discovery, 8, 53-87, 2004.

[17] Hong, T. P., Lin, C. W., and Wu, Y. L. *Incrementally Fast Updated Frequent Pattern Trees.* Expert Systems with Applications 34 (2008), 2424-2435.

[18] Hunter, A., and Liu, W. *Merging Uncertain Information with Semantic Heterogeneity in XML.* http://www.cs.qub.ac.uk/~W.Liu/sac.pdf.

[19] Keijzer, A. D. *Probabilistic XML in Information Integration.* Proceedings of the VLDB2006 Ph.D. Workshop, Seoul, Rep of Korea, 2006, pp. 1-5.

[20] Keijzer, A. D., and Keulen, M. V. *Quality Measures in Uncertain Data Management.* H. Prade and V.S. Subrahmanian (Eds.): SUM 2007, LNAI 4772, pp. 104-115, 2007.

[21] Krakovetskiy, A. U. *Search Method of Associative Rules Using Strong Itemsets and FP-tree.* Automatics and Information Measuring Facilities, 2008, pp. 1-4.

[22] Leung, C. K. S., Carmichael, C., and Hao, B. *Efficient Mining of Frequent Patterns from Uncertain Data.* ICDM-DUNE 2007, pp. 1-19.

[23] Leung, C. K. S., Mateo, M. A. F., and Brajczuk, D. A. *A Tree-Based Approach for Frequent Pattern Mining from Uncertain Data.* T. Washio et al. (Eds.): PAKDD 2008, LNAI 5012, pp. 653-661, 2008.

[24] Li, H. F., Lee, S. Y., and Shan, M. K. *An Efficient Algorithm for Mining Frequent Itemsets over the Entire History of Data Streams.* http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.81.9955.

[25] Li, H. F., Lee, S. Y., and Shan, M. K. *Online Mining (Recently) Maximal Frequent Itemsets over Data Streams.* Proceedings of the 15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications (RIDE-SDMA'05), pp. 11-18, 2005.

[26] Li, H. F., Shan, M. K., and Lee, S. Y. *Online Mining of Frequent Query Trees over XML Data Streams.* WWW 2006, May 23-26, 2006, Edinburgh, Scotland, pp. 959-960.

[27] Malik, H. H., and Kender, J. R. *Optimizing Frequency Queries for Data Mining Applications.* http://www.cs.columbia.edu/~hhm2104/papers/Malik_ICDM07.pdf.

[28] Ni, Y., and Chan, C. Y. *Dissemination of Heterogeneous XML Data in Publish/Subscribe Systems.* CIKM'09, November 2-6, 2009, Hong Kong, China, pp. 127-136.

[29] Sanderson, R. *ARM: A Priori and Data Structures.* http://www.csc.liv.ac.uk/~azaroth/courses/current/comp527/lectures/comp527-19.pdf.

[30] Sun, L., Cheng, R., Cheung, D. W., and Cheng, J. *Mining Uncertain Data with Probabilistic Guarantees.* KDD'10, July 25-28, 2010, Washington, DC, USA, pp. 273-282.

[31] Terrovitis, M., Passas, S., Vassiliadis, P., and Sellis, T. *A Combination of Trie-trees and Inverted Files for the Indexing of Set-valued Attributes.* CIKM'06, November 5-11, 2006, Arlington, Virginia, USA, pp. 728-737.

[32] Zhang, W., Lin, X., Pei, J., and Zhang, Y. *Managing Uncertain Data: Probabilistic Approaches.* Proceedings of the Ninth International Conference on Web-Age Information Management, 2008, pp. 405-412.

[33] 楊，婉祺，康，聖祥，羅，貴魁，and 楊，達立 *Fuzzy Data Mining with Multi-Level Using FP-tree Like Structure.* http://www.mis.stut.edu.tw/result/2008/03-2.pdf.

[34] *IBM Quest Market-Basket Synthetic Data Generator.* http://www.cs.loyola.edu/~cgiannel/assoc_gen.html.