Frequent Pattern Mining with Uncertain Data

Charu C. Aggarwal*
IBM T. J. Watson Research Ctr
Hawthorne, NY, USA
charu@us.ibm.com

Yan Li
Tsinghua University
Beijing, China
lniyan@163.com

Jianyong Wang†
Tsinghua University
Beijing, China
jianyong@tsinghua.edu.cn

Jing Wang
New York University
New York, NY, USA
jwang5@stern.nyu.edu

ABSTRACT

This paper studies the problem of frequent pattern mining with uncertain data. We will show how broad classes of algorithms can be extended to the uncertain data setting. In particular, we will study candidate generate-and-test algorithms, hyper-structure algorithms and pattern growth based algorithms. One of our insightful observations is that the experimental behavior of different classes of algorithms is very different in the uncertain case as compared to the deterministic case. In particular, the hyper-structure and the candidate generate-and-test algorithms perform much better than tree-based algorithms. This counter-intuitive behavior is an important observation from the perspective of algorithm design of the uncertain variation of the problem. We will test the approach on a number of real and synthetic data sets, and show the effectiveness of two of our approaches over competitive techniques.

Executable and Data Sets: Available at:
http://dbgroup.cs.tsinghua.edu.cn/liyan/u_mining.tar.gz

Categories and Subject Descriptors

H.2.8 [**Information Systems**]: Database Applications

General Terms

Algorithms

*This research is continuing through participation in the International Technology Alliance sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defense.

†Jianyong Wang was supported in part by Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology (abbr. TNList), and the Program of State Education Ministry of China for New Century Excellent Talents in University under Grant No. NCET-07-0491, and a research award from Google, Inc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
KDD'09, June 28–July 1, 2009, Paris, France.
Copyright 2009 ACM 978-1-60558-495-9/09/06 ...$5.00.

1. INTRODUCTION

Data mining of uncertain data has become an active area of research recently. A detailed survey of uncertain data mining techniques may be found in [2]. In this paper, we will study the problem of frequent pattern mining with uncertain data. The problem of frequent pattern mining with uncertain data has been studied in a limited way in [7, 8, 11, 14], and a variety of pruning strategies are proposed in order to speed up the algorithm for the uncertain case. This paper will study the problem of frequent pattern mining by examining the relative behavior of the extensions of well known classes of deterministic algorithms. Since many of the techniques used across different frequent pattern mining algorithms are similar, the methodologies for extending the different algorithmic classes have will have applicability beyond the specific algorithms discussed in this paper.

One observation from our extensions to the uncertain case is that the respective algorithms do not show similar trends to the deterministic case. For example, in the deterministic case, the FP-growth algorithm is well known to be an extremely efficient approach. However, in our tests, we found that the extensions of the candidate generate-and-test as well as the hyper-structure based algorithms are much more effective. Furthermore, many pruning methods, which work well for the case of low uncertainty probabilities do not work very well for the case of high uncertainty probabilities. This is because the extensions of some of the algorithms to the uncertain case are significantly more complex, and require different kinds of trade-offs in the underlying computations. Thus, in addition to the new efficient methods proposed by this paper, an important contribution of this paper is the insight that natural extensions of deterministic algorithms may show counter-intuitive behavior.

This paper is organized as follows. The next section defines the uncertain version of the problem. We will also discuss the extension of candidate generate-and-test algorithms to the uncertain version of the problem. The remainder of the paper discusses the extension of other classes of algorithms, and provides comparative experimental results.

1.1 Contributions of this Paper

This paper extends and compares many of the conventional techniques for frequent pattern mining to the uncertain case. These include candidate generate-and-test algorithms, hyper-structure based algorithms, and pattern-

growth algorithms. While the problem of uncertain frequent-pattern mining has recently been studied in a limited way in the literature [8, 14], this is the first comprehensive study which proposes extensions of many classes of algorithms to the uncertain case. In addition to the efficient algorithms proposed, our results suggest that the behavior of the natural uncertain extensions of frequent pattern algorithms is quite different from the deterministic case. The results of this paper suggest for the first time that the efficiency and memory trade-offs are very different for the uncertain case, and it is important to pick the algorithms carefully for extension to uncertain data sets. Since the observations of this paper are fairly general, they may also help in designing better extensions of other deterministic algorithms to the uncertain case.

2. FREQUENT PATTERN MINING OF UNCERTAIN DATA SETS

In this section, we will discuss frequent pattern mining for uncertain data sets. We first introduce some additional notations and definitions. We assume that we have a database $\mathcal{D}$ containing N transactions. We assume that the total number of unique items is d, and each item is denoted by a unique index in the range of $\{1 \ldots d\}$. In sparse databases, only a small number of items have a nonzero probability of appearing in a given transaction. Let us assume that the ith transaction in database $\mathcal{D}$ contains n_i items with non-zero probability. Let us assume that the items in the ith transaction are denoted by $j_1^i \ldots j_{n_i}^i$. Without loss of generality, we can assume that these items are in sorted order. We assume that the probability of the ith item being present in transaction T is given by $p(i, T)$. Thus, in the uncertain version of the problem, the item may be present in the transaction T with the above-mentioned probability.

First, we will define the frequent pattern mining problem. Since the transactions are probabilistic in nature, it is impossible to count the frequency of itemsets deterministically. Therefore, we count the frequent itemsets only in expected value. In order to do so, we need to count the probability of presence of an itemset in a given transaction. Let $s(I)$ be the support of the itemset I. This support can only be counted in probabilistic value. The expected support of an itemset I is defined as follows:

Definition 2.1. *The expected support of itemset I is denoted by $E[s(I)]$, and is defined as the sum of the expected probabilities of presence of I in each of the transactions in the database.*

The problem of frequent itemset mining is defined in the context of uncertain databases as follows:

Definition 2.2. *An itemset I is said to be frequent when the expected support of the itemset is larger than the user-defined threshold minsup.*

Note that the expected number of occurrences of the itemset I can be counted by summing the probability of presence of the itemsets in the different transactions in the database. The probability of the presence of itemset I in a given transaction can be computed using the relationship below.

Observation 2.1. *The probability of the itemset I occurring in a given transaction T is denoted by $p(I, T)$ and is the product of the corresponding probabilities. Therefore, we have the following relationship:*

$$p(I, T) = \prod_{i \in I} p(i, T) \quad (1)$$

The above observation assumes *statistical independence* between the different items in terms of their *uncertain probability behavior*, which is the same as the assumption in other related work [7, 8]. This is also a widely used simplifying assumption across other data mining applications [2]. This does not mean that the underlying *instantiations* of such a database will result in uncorrelated items, since the values of $p(i, T)$ may be non-zero in a transaction only across a small fraction of correlated items.

2.1 Candidate Generate-and-Test Algorithms

We first study candidate generate-and-test algorithms for frequent pattern mining. These can be *join-based* [3] or *set-enumerations based* [1]. The conventional *Apriori* algorithm [3] belongs to this category. The *Apriori* algorithm uses a candidate generate-and-test approach which uses repeated joins on frequent itemsets in order to construct candidates with one more item. A key property for the correctness of *Apriori*-like algorithms is the downward closure property. We will see that the downward-closure property is true in the uncertain version of the problem as well.

Lemma 2.1. *If a pattern I is frequent in expected support, then all subsets of the pattern are also frequent in expected support.*

Proof. Let J be a subset of I. We will first show that for any transaction T, $p(J, T) \geq p(I, T)$. Since J is a subset of I, we have $\frac{p(I,T)}{p(J,T)} = \prod_{i \in I-J} p(i, T) \leq 1$. This implies that $p(J, T) \geq p(I, T)$. Summing this over the entire database $\mathcal{D}$, we get $\sum_{T \in \mathcal{D}} p(J, T) \geq \sum_{T \in \mathcal{D}} p(I, T)$. Therefore, we have:

$$E[s(J)] \geq E[s(I)]$$

The result follows. □

The maintenance of the downward closure property means that we can continue to use the candidate-generate-and-test algorithms without the risk of losing true frequent patterns during the counting process. In addition, pruning tricks (such as those discussed in *Apriori*) which use the downward closure property can be used directly. Therefore the major steps in generalizing candidate generate-and-test algorithms are as follows: (1) All steps for candidate generation using joins and in pruning with the downward closure property remain the same. (2) The counting procedure needs to be modified using Observation 2.1.

We note that a number of recent techniques [7, 8] use further pruning tricks in order to improve efficiency. We will show that the effectiveness of such techniques is data-dependent, since such pruning tricks do not work very well in the case of dense data sets in which uncertainty probabilities are high.

Similar techniques can be used in order to extend set-enumeration based methods [1, 4, 6]. These methods typically use top-down tree-extension in conjunction with branch validation and pruning using the downward closure property. Different algorithms use different strategies for generation

of the tree in order to obtain the optimum results. Since the set-enumeration based algorithms are also based on the downward closure property, they can be easily extended to the uncertain version of the problem. The key modifications to set-enumeration based candidate generate-and-test algorithms are as follows:
(1) The tree-extension phase uses the ordering of the different items in order to construct it in top-down fashion. The tree extension phase is exactly the same as in candidate generate-and-test algorithms. **(2)** The counting of frequent patterns uses Observation 2.1. **(3)** In the event that transactions are projected on specific branches of the tree (as in [1]), we can perform the projection process, except that we need to retain the probabilities of presence of specific items along with the transactions. Also note that the probabilities for the items across different transactions need to be maintained respectively, even if the transactions are identical after projection. This is because the probabilities of the individual items will not be identical after projection. Therefore each projected transaction needs to be maintained separately. **(4)** The pruning of the branches of the tree remains identical because of the downward closure property.

3. PATTERN GROWTH ALGORITHMS

There are also some popular frequent pattern mining algorithms, which are based on the pattern growth paradigm. Among these methods, the H-mine [12] and FP-growth [9] algorithms are two representative ones. Their main difference lies in the data representation structures. FP-growth adopts a prefix tree structure while H-mine uses a hyper-linked array based structure. We will see that the use of such different structures have a substantially different impact in the uncertain case as compared to the deterministic case. Next, we will discuss the extension of each of these algorithms to the uncertain case in some detail.

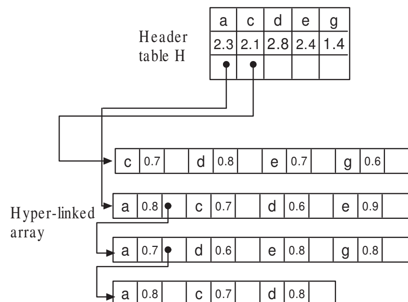


Figure 1: H-Struct

3.1 Extending the H-mine algorithm

The H-mine algorithm proposed in [12] adopts a hyper-linked data structure called H-struct. Similar to FP-growth, it is a partition-based divide-and-conquer method. Initially, H-mine scans the input database once to find the frequent items. The infrequent items are removed from the database. The frequent items left in each input transaction are sorted according to a certain global ordering scheme. The transformed database is stored in an array structure, where each row corresponds to one transaction. During the mining process, there always exists a prefix itemset (denoted by P, which is initially empty). H-mine needs to construct a header table which records the starting places of the projected transactions. By following the links in the header table, H-mine can locate all the projected transactions and find the locally frequent items by scanning the projected transactions. The locally frequent items w.r.t the prefix P can be used to extend P to longer prefix itemsets.

As the hyper-linked array structure used in H-mine is not in a compressed form, it is relatively easy to extend the H-struct for mining frequent itemsets from uncertain data. As described in [12], each frequent item in a transaction is stored in an entry of the H-struct structure with two fields: an item_id and a hyper-link. In addition, the probability $p(i,T)$ of the presence of item i in transaction T is maintained. Figure 1 shows an example of an extended H-struct structure[1]. With the extended H-struct structure there are two ways to mine frequent itemsets with the current prefix P. The first approach is to maintain the probability $p(P,T)$ of prefix P occurring in each projected transaction T in memory. As the probability of the presence of locally frequent item i in transaction T is recorded in the extended H-struct, it is straightforward to compute the expected support of the new itemset of $P \cup \{i\}$, $E[s(P \cup \{i\})]$, according to Observation 2.1 and Definition 2.1. However, the expected support of prefix P with respect to each conditional transaction needs to be maintained until all the locally frequent items w.r.t. prefix P have been processed. This may cost significant memory and may also lead to deterioration in the internal caching behavior of the algorithm.

In order to avoid maintaining the probability of prefix P with respect to each projected transaction T, $p(P,T)$, we have another approach for computing it on the fly. As H-mine adopts the pseudo-projection method, each original input transaction is stored in the H-struct. By scanning the sub-transaction before the projected transaction of prefix P, we can find the probability of each item in P. Thus, $p(P,T)$ can be computed according to Observation 2.1. In a similar way, $E[s(P \cup \{i\})]$ can be computed according to Observation 2.1 and Definition 2.1. In this paper, we adopt the second approach for computing the expected support of the current prefix itemset P. This is because the use of on-the-fly computations reduces the space-requirements of the technique. The reduced space-requirements also indirectly improve the locality of the caching behavior of the underlying computations. This leads to improved efficiency of the overall algorithm.

3.2 Extending the FP-growth Algorithm

FP-growth [9] is a key frequent itemset mining algorithm, which is based on the pattern growth paradigm. It adopts a prefix tree structure, FP-tree, to represent the database (or conditional databases). As FP-tree is a compressed structure, it poses several challenges when we try to adapt the FP-growth algorithm for uncertain data sets. These challenges are as follows:
(1) In the original FP-tree structure, each node has a 'count' entry which records the number of transactions containing the prefix path from the root node to this node. For uncertain data, if we just store in a node the sum of item probabilities with respect to the transactions containing the prefix path, we will no longer be able to determine the prob-

[1]Note that the second row of the header table in the H-struct structure stores the sum of item probabilities for each locally frequent item.

ability of the presence of an item in each transaction. Thus, there is an irreversible loss of information in the uncertain case with the use of a compressed structure. Thus, we need to find a different and efficient way to store the item probabilities without losing too much information.
(2) The original FP-growth algorithm mines frequent itemsets by searching the tree structure in a bottom up manner. The computation of the support of a given prefix path is quite straightforward. Its support is simply the support of the lowest node of the path. However, for uncertain data, the expected support of a given prefix path should be computed according to Definition 2.1. As we no longer know the mapping among the item probabilities and the transactions, it is impossible to compute the exact expected support of a given prefix according to Definition 2.1.
(3) Since it is impossible to determine the exact expected support of each frequent itemset based on the FP-tree structure, we may need to first mine all the candidate itemsets, and then remove the infrequent itemsets by checking the original database. The process of determining such infrequent itemsets efficiently can be quite difficult in the uncertain case.

There are two extreme solutions to adapt the FP-tree structure for uncertain data mining. Let us denote the FP-tee built from uncertain data by UFP-tree. The first one is to store (in each node) the sum of item probabilities with respect to the transactions containing the prefix path from the root to it. The UFP-tree built in this way is as compact as the original FP-tree. However, it cannot even be used to compute the the upper bound or lower bound of the expected support of an itemset, because it loses information with respect to the distinct probability values for different transactions. Another extreme solution is to split a node into m nodes if the item in this node has m distinct probability values. In this case, we can compute the exact expected support. On the other hand, the UFP-tree built in this way consumes a lot of memory.

In this work, we adopt a compromise by storing a subset of the probabilities for the item in each node. These probabilities are selected using clustering, and are stored as floating point numbers. This method does not consume too much memory, and we will show that it allows us to compute an upper bound on the expected support of any itemset. We then compute a set of candidate frequent itemsets based on this upper bound. This set of candidates provides us with a superset of the complete set of real frequent itemsets. Any remaining false positives will then need to be removed by scanning the input database. Next, we will discuss the adaptation of different steps of the FP-growth algorithm to the uncertain case.

3.2.1 Construction of the UFP-tree

The process of constructing the UFP-tree for uncertain data is very similar to the construction of the FP-tree for deterministic data sets. The main difference lies in the information stored in each node. The UFP-tree is built using the following steps. First, the database is scanned to find the frequent items and to generate a support descending item list. Then, the transactions are read one by one, and the infrequent items are pruned. The remaining frequent items are sorted according to the frequent item list. The re-ordered transactions are inserted into the UFP-tree.

As discussed earlier in this paper, each node of the UFP-tree stores a summary of the probabilities of the non-zero probability items in those transactions which share the same prefix path in clusters. We partition the probabilities into a set of k clusters. The corresponding parameters created for the ith cluster by the partitioning are represented by c_i and m_i ($1 \leq i \leq k$), where c_i denotes the maximum probability value in the ith cluster and m_i is the number of item probabilities in the ith cluster. We assume that $c_1 > c_2 > \ldots > c_k$. The reason that we store the maximum probability value in each cluster instead of the center of the cluster (i.e., the average value of all the item probabilities in this cluster) is to make sure that the support computed from the summary is no less than the true support. In Section 3.2.2, we will introduce a method to compute an upper bound on the true support based on the cluster information stored in each node. Besides the global UFP-tree construction from the database, conditional UFP-trees are generated from the global UFP-tree. Therefore, there are two different situations which need the data summarization in the construction process. We will discuss the solutions separately under the two situations.

There are several clustering and data summarization methods available for our task. The choice of the proper method should consider two factors. The first is memory usage. This also indirectly affects the performance since lower memory consumption results in better internal caching behavior on most machines. Since there could be a large number of nodes in the UFP-tree, the summarization of the probabilities in each node should be as concise as possible in order to reduce memory consumption. The trade-off is that greater conciseness leads to lower precision. In the mining process, we compute the upper bound of the support of each itemset according to the summarization of probabilities stored in each node. We use this to compute the candidate frequent itemsets. If the precision of the summarization is too low, the difference between the upper bound and the true support will be large and a large number of false positives may be generated. This will increase the memory and space requirements for the elimination process of the false positives. Clearly, the tradeoff needs to be carefully exploited in order to optimize the performance of the underlying algorithm.

This problem is closely related to that of building V-optimal histograms [10] for time-series data. It is however not natural to apply the V-optimal technique to this situation. During the construction of the UFP-tree, the transactions are read sequentially, and the item probabilities corresponding to a certain node will typically arrive in neither ascending nor descending order. In order to apply the V-optimal algorithm to this set of probabilities (which are floating point numbers) in each node, we would need to sort these numbers in ascending or descending order, and this is time consuming. Furthermore, the time and space complexities of the complete V-optimal method are $O(n^2 \cdot k)$ and $O(n \cdot k)$ respectively. Because of the expensive behavior of the V-optimal method, we decided to use k-means clustering instead. However, if we store all the item probabilities associated with each node before applying k-means clustering during the UFP-tree construction process, it will consume too much memory and is too expensive for large data sets.

Therefore, we used a different approach by using a modified version of the k-means algorithm. First, we partition the range of the probabilities of items into ϕ parts in equal width, where ϕ is chosen to be significantly larger than k.

We store the maximum probability value and the number of distinct item probabilities in each part. After we have all the transactions inserted into the UFP-tree, we then cluster these stored information by k-means.

As mentioned earlier, there are two points in the pattern mining process in which we need to compute the data summarizations. The first relates to the construction of the global UFP-tree. We have discussed the first situation above. The second is the construction of conditional UFP-trees during the mining process. We will discuss this second situation at this point. Suppose we begin to mine the frequent itemsets with prefix item 'g'. By computing the expected support of size 2-itemsets containing 'g' with the method discussed in Section 3.2.2, we could find the locally frequent items. Then, we traverse each path in the global UFP-tree linking the node with a label 'g' to the root to extract the locally frequent items and the corresponding distribution information of the item probabilities stored in each node along the path. This forms a conditional transaction. Here we give such an example of a conditional transaction, which contains three items and corresponds to 30 input transactions: $\{(a,((0.6,5),(0.7,5),\ (0.8,20))),\ (b,((0.8,10),(0.9,\ 20))),\ (e,\ ((0.7,20),(0.88,10)))\}$. Next, we insert each conditional transaction into the conditional UFP-tree with respect to the item 'g'. Note that the number of probabilities of each item equals 30. This is the number of the probabilities in the node 'g' at the bottom of the corresponding path. This also means that there are 30 input transactions containing 'g' in this path. Notice that we need to merge the clusters after all the transactions are inserted in the conditional UFP-tree in order to keep a limited number of entries in each node. Thus is also done with the k-means clustering algorithm.

3.2.2 Computation of Support Upper Bounds

In order to mine the frequent itemsets, we first need to mine the potentially-frequent itemsets using the information stored in each node. As mentioned earlier, the precise support of an itemset cannot be computed directly from the UFP-tree because of the information loss during compression. However, it is possible to compute an upper bound on the support. It is clear that the number of item probabilities in each node along any tree path may vary considerably. Let the number of item probabilities in the last node of a path be denoted by n (namely, the number of transactions containing the path is n). We should take out n largest probabilities in each node along the prefix path from this last node up to the root, and this is an easy task since the item probabilities are summarized in the clusters. For example, suppose all the item probabilities in each tree node are grouped into three clusters (i.e., k=3), and the cluster information in the last node N of a given path P is $\{(c_1{=}0.95,\ m_1{=}2),\ (c_2{=}0.9,\ m_2{=}2),\ (c_3{=}0.8,\ m_3{=}1)\}$. The last node contains five item probabilities. Let N' be any node along the path P, and its cluster information be $\{(c'_1{=}0.98,\ m'_1{=}3),\ (c'_2{=}0.91,\ m'_2{=}1),\ (c'_3{=}0.85,\ m'_3{=}2)\}$. The five largest item probabilities in N' are 0.98, 0.98, 0.98, 0.91, and 0.85, respectively. The process of computing an upper bound on the expected support of an itemset I w.r.t. a prefix P is shown in Algorithm 1.

ALGORITHM 1: **Computation of an upper bound on the expected support of an itemset I w.r.t. prefix path P**

Input: The cluster information stored in each node along path P corresponding to I, $(c_{i1},\ m_{i1})$, $(c_{i2},\ m_{i2})$, ..., $(c_{ik},\ m_{ik})$, i=1, 2, ..., $|I|$, and $c_{i1}>c_{i2}>\ldots>c_{ik}$ holds.
Output: An upper bound of the expected support of itemset I w.r.t. path P, $E(s(I)|_P)$
Initialization:
$E(s(I)|_P)=0$;
$C_1\leftarrow c_{11},\ C_2\leftarrow c_{21},\ \ldots,\ C_{|I|}\leftarrow c_{|I|1}$;
$M_1\leftarrow m_{11},\ M_2\leftarrow m_{21},\ \ldots,\ M_{|I|}\leftarrow m_{|I|1}$;
Method: Repeat the following steps below until no item probability in the last node of the path corresponding to itemset I is left.
1) $E(s(I)|_P)=E(s(I)|_P)+C_1\times C_2\times\ldots\times C_{|I|}\times m$, where $m = min(M_1, \quad M_2, ..., M_{|I|})$;
2) $M_1\leftarrow M_1-m,\ M_2\leftarrow M_2-m,\ \ldots,\ M_{|I|}\leftarrow M_{|I|}-m$;
3) For $i\in[1,|I|]$ do
if M_i=0
Suppose $C_i=c_{ij}$ (where $1\leq j<k$), then
$C_i\leftarrow c_{i(j+1)}$ and $M_i\leftarrow m_{i(j+1)}$;

Note that an itemset I may be contained in multiple prefix paths, and we can compute an upper bound of the expected support of itemset I with respect to each of these prefix paths. The sum of the upper bounds with respect to these prefix paths must form an upper bound of itemset I. We will prove that the output of Algorithm 1 is an upper bound on the expected support of the itemset w.r.t path P.

LEMMA 3.1. *Given an itemset I, when $|I| = 2$, the support computed according to algorithm 1 is an upper bound of the expected support of I with respect to path P.*

PROOF. Suppose the number of item probabilities in the last node of path P is n, that is, $\sum_{i=1}^{k} m_{|I|i} = n$. Let us denote the two tree nodes corresponding to the two items in I w.r.t. path P by a and b, the top n largest item probabilities in node a by $a_1\geq a_2\geq\ldots\geq a_n$, and the top n largest item probabilities in node b by $b_1\geq b_2\geq\ldots\geq b_n$. We will prove the lemma using induction.

(1) Let n=2. Since $(a_1-a_2)(b_1-b_2)\geq 0$ holds, we have

$$a_1b_1 + a_2b_2 \geq a_1b_2 + a_2b_1$$

Therefore, the lemma holds when $n = 2$.

(2) Assume the induction base that when n=k, $\sum_{i=1}^{k} a_ib_i$ is an upper bound. Next, let n=k+1, we will find the maximum sum of products. Let b_{k+1} multiply a_t, $1\leq t\leq k$, under the assumption above we know that the maximum sum of products of the k numbers left is $(\sum_{i=1}^{t-1} a_ib_i+\sum_{i=t+1}^{k+1} a_ib_{i-1})$. Furthermore, we have:

$$\begin{aligned}
&\sum_{i=1}^{k+1} a_ib_i - (\sum_{i=1}^{t-1} a_ib_i + a_tb_{k+1} + \sum_{i=t+1}^{k+1} a_ib_{i-1})\\
&= (a_tb_t + a_{t+1}b_{t+1} + a_{t+2}b_{t+2} + \ldots a_{k+1}b_{k+1}) -\\
&(a_tb_{k+1} + a_{t+1}b_t + a_{t+2}b_{t+1} + \ldots + a_{k+1}b_k)\\
&= a_{t+1}(b_{t+1}-b_t) + a_{t+2}(b_{t+2}-b_{t+1}) + \ldots\\
&+ a_{k+1}(b_{k+1}-b_k) + a_t(b_t-b_{k+1})\\
&\geq a_t(b_{t+1}-b_t) + a_t(b_{t+2}-b_{t+1}) + \ldots\\
&+ a_t(b_{k+1}-b_k) + a_t(b_t-b_{k+1})\\
&= a_t(b_{k+1}-b_t) + a_t(b_t-b_{k+1}) = 0
\end{aligned}$$

then, we derive that when n=k+1, $\sum_{i=1}^{k+1} a_ib_i$ is an upper bound of the sum of the products.

(3) Since when $|I| = 2$ the output of Algorithm 1 is $\sum_{i=1}^{n} a_ib_i$, the expected support computed following the steps in Algorithm 1 must be an upper bound. □

COROLLARY 3.1. *Given two groups of n ($\forall n, n > 0$) non-negative floating-point numbers sorted in decreasing order,*

c_{ij} ($\forall$ i, p, q, $1 \leq i \leq 2$, if $1 \leq p < q \leq n$, then $c_{ip} \geq c_{iq}$ holds), $\sum_{j=1}^{x} \prod_{i=1}^{2} c_{ij}$ is the largest among all the sums of x products, where $1 \leq x \leq n$.

PROOF. This result can be easily derived from Lemma 3.1 when x=n. For any possible set of x products which are constructed from the two groups of n floating-point numbers, denoted by s_1, s_2,...s_x, we can always find another set of x products which are constructed from the two groups of the first x floating-point numbers, denoted by s_1', s_2', ..., s_x', such that $s_l \leq s_l'$ ($\forall l$, $1 \leq l \leq x$). That is, $(\sum_{j=1}^{x} s_j) \leq (\sum_{j=1}^{x} s_j')$. In addition, according to the proof of Lemma 3.1 we know that $(\sum_{j=1}^{x} \prod_{i=1}^{2} c_{ij}) \geq (\sum_{j=1}^{x} s_j')$ holds. Thus, we have $(\sum_{j=1}^{x} \prod_{i=1}^{2} c_{ij}) \geq (\sum_{j=1}^{x} s_j)$, which means $\sum_{j=1}^{x} \prod_{i=1}^{2} c_{ij}$ is the largest among all the sums of x products, where $1 \leq x \leq n$. □

THEOREM 3.1. *Given m groups of n ($\forall n, n > 0$) floating-point numbers sorted in decreasing order, c_{ij} ($\forall$ i, p, q, $1 \leq i \leq m$, if $1 \leq p < q \leq n$, then $c_{ip} \geq c_{iq}$ holds), $\sum_{j=1}^{x} \prod_{i=1}^{m} c_{ij}$ is the largest among all possible sums of x products, where $1 \leq x \leq n$.*

PROOF. We prove the theorem using induction.
1. According to Corollary 3.1, we know that it is true when $m = 2$.
2. We assume, when $m = k$, the theorem holds.
3. We will derive from the above assumption that when m=k+1, $\sum_{j=1}^{x} \prod_{i=1}^{m} c_{ij}$ is still the largest among all possible sums of x products, where $1 \leq x \leq n$. Let the (k+1)-th group of n floating-point numbers be $c_{(k+1)1} \geq c_{(k+1)2} \geq \ldots \geq c_{(k+1)n}$. As $c_{(k+1)1}$, $c_{(k+1)2}$, ..., and $c_{(k+1)x}$ are among the top x largest values in the (k+1)-th group of n floating-point numbers, one of the largest values of the sum of x products constructed from the k+1 groups of n floating numbers must be in the form of $c_{(k+1)1}s_1 + c_{(k+1)2}s_2 + \ldots + c_{(k+1)x}s_x$, where $s_i = \prod_{j=1}^{k} z_{i_j}$, $z_{i_j} \in \{c_{j1}, c_{j2}, \ldots, c_{jn}\}$.

If we use s_y' to denote $\prod_{i=1}^{k} c_{iy}$, we have:

$$\sum_{j=1}^{x} \prod_{i=1}^{m} c_{ij} = c_{(k+1)1}s_1' + c_{(k+1)2}s_2' + \ldots + c_{(k+1)x}s_x'$$

and $s_1' \geq s_2' \geq \ldots \geq s_x'$ must hold.

In addition, we also have:

$$\begin{aligned}
&c_{(k+1)1}s_1' + c_{(k+1)2}s_2' + \ldots + c_{(k+1)x}s_x' - \\
&(c_{(k+1)1}s_1 + c_{(k+1)2}s_2 + \ldots + c_{(k+1)x}s_x) \\
&= (s_1' - s_1)(c_{(k+1)1} - c_{(k+1)2}) + \\
&[(s_1' - s_1) + (s_2' - s_2)](c_{(k+1)2} - c_{(k+1)3}) + ... + \\
&[(s_1' - s_1) + (s_2' - s_2) + ... + (s_x' - s_x)]c_{(k+1)x} \\
&= (s_1' - s_1)(c_{(k+1)1} - c_{(k+1)2}) + \\
&[(s_1' + s_2') - (s_1 + s_2)](c_{(k+1)2} - c_{(k+1)3}) + ... + \\
&[(s_1' + s_2' + ... + s_x') - (s_1 + s_2 + ... + s_x)]c_{(k+1)x}
\end{aligned}$$

According to our assumption, $\forall l \leq x$, $(\sum_{i=1}^{l} s_i' - \sum_{i=1}^{l} s_i) \geq 0$ holds, and as $(c_{(k+1)l} - c_{(k+1)(l+1)}) \geq 0$ also holds, we get that $(\sum_{j=1}^{x} c_{(k+1)j}s_j' - \sum_{j=1}^{x} c_{(k+1)j}s_j) \geq 0$. Therefore, when m=k+1, $\sum_{j=1}^{x} \prod_{i=1}^{m} c_{ij}$ is still the largest among all possible sums of x products, where $1 \leq x \leq n$. □

COROLLARY 3.2. *The output of Algorithm 1 must be an upper bound of the expected support of itemset I ($|I| \geq 2$) w.r.t. prefix P.*

PROOF. There are $|I|$ nodes in the path P which correspond to the $|I|$ items in I, and each node maintains k clusters. The cluster information of the last node in path P is represented by $c_i(m_i)$, i=1...k, and we let $n = \sum_{j=1}^{k} m_i$. We can then sort the n item probabilities in the last node in descending order. For each of the other $|I|-1$ nodes, we can extract its top n largest item probabilities and sort them in descending order. In this way, we get $|I|$ groups of n item probabilities, denoted by z_{ij}, where $1 \leq i \leq |I|$, $1 \leq j \leq n$, and $\forall$ p,q, if $p<q$, $z_{ip} \geq z_{iq}$. According to Algorithm 1 we know the output of Algorithm 1 equals $\sum_{j=1}^{n} \prod_{i=1}^{|I|} z_{ij}$. According to Theorem 3.1, we have that it is an upper bound of the expected support of itemset I w.r.t. prefix P. □

3.2.3 Mining Frequent Patterns with UFP-tree

We used two different approaches for the mining process with UFP-tree. One is the recursive pattern growth approach introduced in [9]. The other is the one described in [13], which constructs a conditional UFP-tree for each frequent item, and then mines frequent itemsets in each conditional tree. In the following, we will explain the two mining methods in detail.

Assume that the frequent item list in support-descending order is $\{e, a, c, d, g\}$. The process of recursively constructing all-level conditional UFP-trees is as follows. First, the algorithm mines frequent itemsets containing g. Second, it mines frequent itemsets containing d but not g. Third, it mines frequent itemsets containing c but neither d nor g. This pattern is repeated until it mines frequent itemsets containing only e. When we are mining frequent itemsets containing item d, we first compute the upper bound of the expected support of each itemset (e.g., $(e, d), (a, d), and (c, d)$) with the method described in Section 3.2.2, and form the locally frequent item list in support descending order (e.g., $\{c, e\}$). Next, the algorithm traverses the UFP-tree by following the node-links of item d again to get the locally frequent itemset information in each path which forms a conditional transaction. We insert the conditional transaction into the conditional UFP-tree with respect to item d. After that, we will repeat the above steps to this conditional UFP-tree of item d, which is the same as the depth-first search in [9].

Observation 2.1 defines the probability of an itemset as the product of the underlying item probabilities. This suggests that the expected support of an itemset decreases quickly when its length increases. While the algorithm proposed in [13] is designed for the deterministic case, we observe that it avoids recursively constructing conditional FP-trees, and can therefore make good use of the *geometric decrease* in the calculations of the expected support of itemsets. This is our rationale for specifically picking the frequent itemset mining algorithm introduced in [13]. It constructs a one-level conditional FP-tree for each frequent item. After we have found the locally frequent items for each conditional FP-tree, we do not reorder the items, but we follow the global frequent item list. The reason for doing so is for the generation of the trie tree. The algorithm in [13] also adopts the popular divide-and-conquer and pattern growth paradigm. Suppose the locally frequent items with respect to the prefix item 'g' are $\{e, a, c, d\}$. The algorithm in [13] computes the itemsets containing 'd' first, then computes those containing 'c' but

no ‘d’, then those containing ‘a’ but no ‘c’ nor ‘d’, and those containing ‘e’ only at the end. Then, the algorithm proceeds to generate itemsets of increasing size sequentially, until the set of locally frequent items becomes empty.

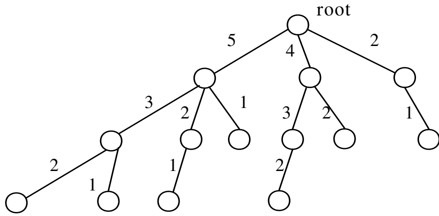


Figure 2: An example of a trie tree

3.2.4 Determining Support with a Trie Tree

As mentioned above, the itemsets mined so far are just candidate frequent itemsets and may not be really frequent. In order to determine the real support of each candidate itemset, we store the candidate frequent itemsets in a trie tree structure which is suitable to search and locate an itemset. Figure 2 shows an example of a trie tree, which contain a total of 14 nodes including the root node. Each path from the root to a certain node represents an itemset, thus the trie tree in Figure 2 stores 13 itemsets. We can also see that along each path from the root to a leaf, the indices of the items are sorted in decreasing order, and the child nodes of each node are also sorted in decreasing order. This arrangement of the items in the trie tree facilitates the search and locating of an itemset.

In order to obtain the exact support of these candidate itemsets stored in the trie tree, we need to read in the transactions one by one, find the candidate itemsets contained in each transaction and calculate the support according to Definition 2.1 and Observation 2.1. Similar to [5], when we deal with a transaction, we maintain two indices, one is for the transaction, the other points to the node of the trie tree. The entire process is an index moving process. The index for the trie tree is moved to find the item pointed by the index for the transaction, and then the index for the transaction is moved to the next item. Once an itemset is found to be contained in a transaction, its expected support is summed according to Definition 2.1 and Observation 2.1. This process continues until the transaction index reaches the last item in the transaction or the trie tree index reaches the last node of the tree.

4. PERFORMANCE STUDY

In this section, we present the performance study for the extended classical frequent pattern mining algorithms of Apriori, H-mine, and FP-growth. In the following we will denote these revised algorithms by UApriori, UH-mine, and UFP-growth, respectively. We will compare their performance with the state-of-the-art frequent itemset mining algorithm for uncertain data sets, which is the DP approach proposed in paper [8]. We implemented one of the DP methods proposed in [8] and denote it by UCP-Apriori. The UCP-Apriori integrates a pruning method called CP with the Apriori frequent itemset mining framework. The experiments were conducted on a machine with 2.66GHz CPU and 2G main memory installed. The operating system is GNU/Linux.

Four data sets were used in the experiments. The first two datasets, *Connect4* and *kosarak*, are real datasets which were downloaded from the FIMI repository.[2] The *Connect4* data is very dense while *kosarak* is very sparse. The other two data sets, T40I10D100K and T25I15D320k, were generated using the IBM synthetic data set generator [3]. We note that these are deterministic data sets. In order to obtain uncertain data sets, we introduced the uncertainty to each item in these data sets. We allocated a relatively high probability to each item in the data sets in order to allow the generation of longer itemsets. We assume that the uncertainty of those items follows the commonly used normal distribution $N(\mu, \sigma^2)$. The value of μ was independently and randomly generated in the range of [0.87, 0.99] for each item in each transaction, while the value of σ was generated in the same way but in the range of [1/21, 1/12]. We generated a number between 0 and 1 for every item according to its randomly given distribution. The high value of the uncertain probability allowed us to stress test the technique for a more challenging case than that discussed in other work such as [7, 8].

As discussed in Section 3.2.3, we implemented two variants of the UFP-growth algorithm for uncertain data mining. In the following we denote the variant of UFP-growth which recursively constructs all levels of conditional UFP-trees on uncertain data by UFP-tree, while we denote the other one which only constructs the first-level UFP-trees by UCFP-tree. In the experiments, we ran these algorithms under different support levels to compare their efficiency for data sets Connect4, kosarak, and T40I10D100K.

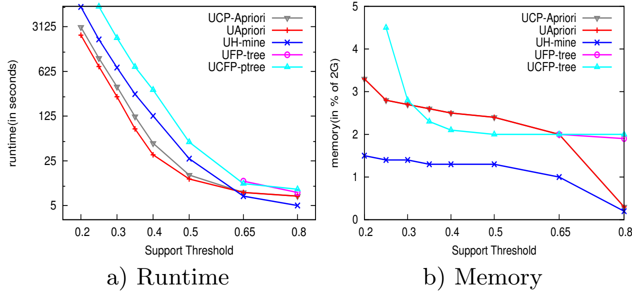
4.1 Performance Comparison

In the following, we illustrate the performance comparison of the five algorithms in terms of runtime and memory consumed on three data sets of Connect4, kosarak, and T40I10D100K, with varying support thresholds. In the uncertain case, memory consumption is an especially important resource because of the additional information about probabilistic behavior which needs to be stored. In resource-constrained hardware, memory-consumption may even decide the range in which a given algorithm may be used. In such cases, memory consumption may be an even more important measure than running time. Therefore, we will test the memory consumption in addition to efficiency. We will see that different algorithms provide the best performance with the use of different measures. Our broad observation is that UH-mine is the only algorithm which performs robustly for all measures over all data sets, whereas the variations of candidate generate-and-test also perform quite well, especially in terms of running time. This would suggest that UH-mine is the most practical algorithm to use in a wide variety of scenarios.

Figures 3(a) and 3(b) show the runtime and memory consumption comparison[3] result on the dense Connect4 data set. UApriori and UH-mine provide the fastest performance at different support thresholds, whereas UH-mine provides the best memory consumption across all thresholds. Thus,

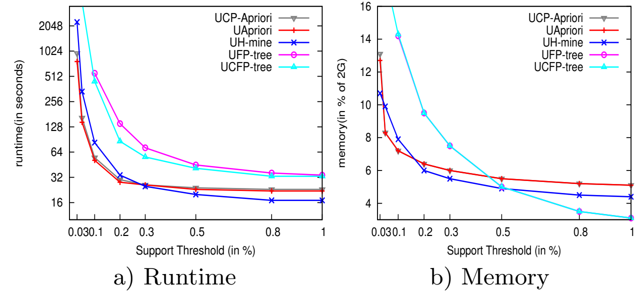
[2]**URL:** *http://fimi.cs.helsinki.fi/data/*

[3]In all the memory consumption comparisons in Figures 3(b), 4(b), 5(b) and 6(b), the curves for UApriori and UCP-Apriori closely overlap, and are therefore difficult to distinguish. Similarly, curves for UFP-Tree and UCFP-Tree closely overlap.



a) Runtime b) Memory

Figure 3: Connect4 dataset



a) Runtime b) Memory

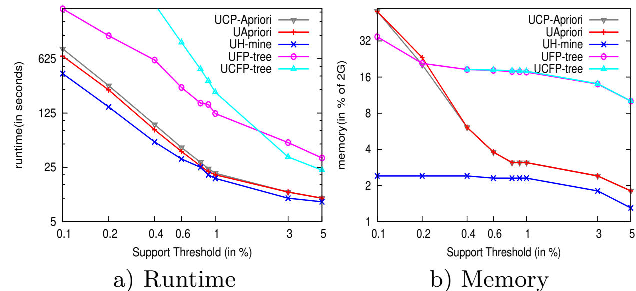
Figure 4: kosarak dataset

the UH-mine algorithm performs robustly on both measures. UFP-tree and UCFP-tree are the slowest. UCP-Apriori [8] is slower than our version of the Apriori algorithm, which is denoted by UApriori. This is because the method for candidate pruning in UCP-Apriori algorithm is not very efficient and only introduces additional overhead, unless the uncertainty probabilities are set too low as in [8]. However, low uncertainty probabilities are an uninteresting case, since the data will no longer contain long frequent patterns (because of the multiplicative behavior of probabilities and its impact on the expected support), and most algorithms will behave efficiently. It is particularly interesting that the uncertain extension of most deterministic algorithms can perform quite well, whereas the extensions to the well known FP-Tree algorithms do not work well at all. We traced the running process of UFP-tree and UCFP-tree and found that considerable time is spent on the last step of eliminating false positives. Furthermore, in most paths in the UFP-tree, the probabilistic information for thousands of transactions need to be stored, and the concise behavior of the deterministic case is lost. It is this concise behavior which provides the great effectiveness of this technique in the deterministic case, and the loss of this property in the probabilistic case is an important observation from the perspective of algorithmic design. In comparison to the UFP-tree, the UCFP-tree does not need to build all levels of conditional UFP-trees recursively, and it only needs to mine all frequent itemsets in one-level of conditional UFP-tree. Thus, it performs better than UFP-tree.

Figure 3(b) compares the memory usage on Connect4 data set. In this case, the behavior of UH-Mine is significantly superior to the other algorithms. As UApriori needs to store a large number of candidate itemsets, UApriori consumes more memory than UH-mine which outputs those mined frequent itemsets on the fly. Connect4 is a relatively coherent data set, and so it is more likely for transactions to share the same prefix path when inserting into the UFP-tree. Thus, it gets the highest compression ratio among all the data sets. However, because the UFP-tree stores uncertainty information, its memory consumption is greater than UApriori. Furthermore, as the support threshold goes down, it generates too many candidate itemsets. This leads to the sharp increase of memory usage.

Data set kosarak is sparse and therefore, the tree like enumeration of the underlying itemsets shows a bushy structure. As shown in figure 4(a), both UApriori and UH-mine perform very well on the kosarak data set. In figure 4(a), the Y-axis is in logarithmic scale. UCP-Apriori runs slightly slower than UApriori. UFP-tree and UCFP-tree do not scale well with the decrease of the support threshold. For data set kosarak, the UFP-tree is also bushy and large. When the support is 0.0003, UCFP-tree costs too much time on kosarak for the reason that the one-level conditional UFP-trees are still large. For UFP-trees, too many recursively constructed conditional UFP-trees and the large number of false positives consume too much memory.

Figure 4(b) shows the comparison of memory consumed by these algorithms on kosarak. As the support threshold decreases, the UFP-trees constructed become large and candidate itemsets generated for UFP-tree and UCFP-tree increase quickly, and thus the memory usage increases fast. For UApriori, the memory consumed for storing candidate frequent itemsets increases rapidly and surpasses UH-mine which only holds the H-struct when the support threshold becomes relatively low. The UH-mine maintains its robustness in terms of memory consumption across all datasets.



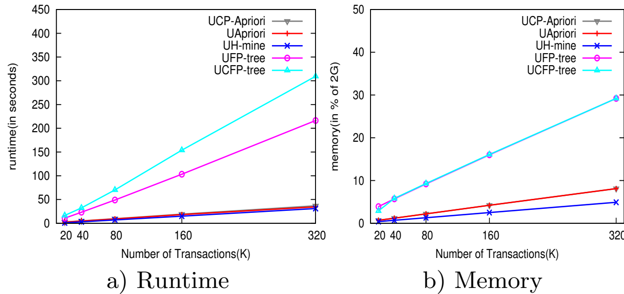
a) Runtime b) Memory

Figure 5: T40I10D100K dataset

Synthetic data set T40I10D100K contains abundant mixtures of short itemsets and long itemsets. Thus, it is a good data set for testing the behavior of the algorithms when the itemsets cannot be perfectly characterized to show any particular pattern. In this case, the UH-mine is significantly superior to all algorithms both in terms of running time and memory usage. We find that as the support threshold decreases, the gap between UH-mine and UApriori becomes quite large. We note that since the Y-axis in Figure 5(a) is in a logarithm scale of 5, the performance difference between the two algorithms is much greater than might seem visually. As shown in Figure 5(b), the memory cost for UApriori increases dramatically when the support threshold decrease below 0.6%. This is because the number of frequent itemsets increases rapidly with reduction in support. UCP-Apriori is a little slower than UApriori and they consumes similar volume of memory.

According to the above experimental results, UApriori and UH-mine are both efficient in mining frequent item-

sets. Both algorithms run much faster than UFP-tree and UCFP-tree, especially when the support threshold is pretty low. However, with the support level decreases, the number of frequent itemsets increases exponentially, which results in sharp increase of the memory cost. UH-mine is the only algorithm which shows robustness with respect to both efficiency and memory usage. The reason that the FP-growth algorithm is not suitable to be adapted to mine uncertain data sets lies in compressed structure which is not well suited for probabilistic data. UCP-Apriori [8] runs a little slower than UApriori on the three data sets. The memory cost for UCP-Apriori is almost the same as that for UApriori, and therefore UApriori is a more robust algorithm that UCP-Apriori on the whole.



a) Runtime b) Memory

Figure 6: Scalability Comparison

4.2 Scalability Comparison

To test the algorithm scalability w.r.t. the number of transactions, we used the synthetic data set T25I15D320k. It contains 320,000 transactions and a random subset of these is used in order to test scalability. The support threshold is set to 0.5%.

Figure 6 a) shows that all these algorithms have linear scalability in terms of running time against the number of transactions varying from 20k to 320k. Among them, UH-mine, UApriori, and UCP-Apriori have much better performance than UFP-tree and UCFP-tree, and among all the algorithms, H-mine has the best performance.

In Figure 6 b), all algorithms shows linear scalability in terms of memory usage. The curves denoted for UFP-tree and UCFP-tree almost coincide, and so do the curves denoted for UApriori and UCP-Apriori. Both UApriori and UH-mine scale much better than UFP-tree and UCFP-tree. UH-mine algorithm scales better than any of the algorithms. Thus, the UH-mine algorithms shows the best scalability both in terms of running time and memory usage.

5. DISCUSSION AND CONCLUSIONS

In this paper, we focus on the frequent itemset mining on uncertain data sets. We extended several existing classical frequent itemset mining algorithms for deterministic data sets, and compared their relative performance in terms of efficiency and memory usage. We note that the uncertain case has quite different trade-offs from the deterministic case because of the inclusion of probability information. As a result, the algorithms do not show similar relative behavior as their deterministic counterparts.

As mentioned in [9], the FP-growth method is efficient and scalable, especially for dense data sets. However, the natural extensions to uncertain data behaves quite differently. There are two challenges to the extension of the FP-tree based approach to the uncertain case. First, the compression properties of the FP-Tree are lost in the uncertain case. Second, a large number of false positives are generated, and the elimination of such candidates further affects the efficiency negatively.

UH-mine is an algorithm which divides the search space and employs the pattern-growth paradigm, which can avoid generating a large number of candidate itemsets, especially when most of them are infrequent. Both UCP-Apriori [8] and UApriori are extended from the well-known Apriori algorithm. The UCP-Apriori algorithm applies a candidate pruning method during the mining process. According to our performance study, the pruning method proposed for UCP-Apriori results in greater overhead than the efficiency it provides in the most challenging scenarios where uncertainty probabilities are high and long patterns are present. The UH-mine algorithm is especially useful, because it uses the pattern growth paradigm, but does so without using the FP-tree structure which does not extend well to the uncertain case. This also reduces the memory requirements drastically. The UH-mine algorithm proposed in this paper provides the best trade-offs both in terms of running time and memory usage.

6. REFERENCES

[1] R. Agarwal, C. Aggarwal, V. Prasad. A Tree Projection Algorithm for Generating Frequent Itemsets. *Journal of Parallel and Distributed Computing*, 61(3), 2001.

[2] C. C. Aggarwal. Managing and Mining Uncertain Data, *Springer*, 2009.

[3] R. Agrawal, R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. *VLDB* 1994.

[4] R. J. Bayardo. Efficiently mining long patterns from databases *SIGMOD* 1998.

[5] F. Bodon. A fast APRIORI implementation. **URL:** [http://fimi.cs.helsinki.fi/src/].

[6] D. Burdick, M. Calimlim, J. Gehrke. MAFIA: A Maximal Frequent Itemset Algorithm. *IEEE TKDE*, 17(11), pp. 1490–1504, 2005.

[7] C.-K. Chui, B. Kao, E. Hung. Mining Frequent Itemsets from Uncertain Data. *PAKDD* 2007.

[8] C.-K. Chui, B. Kao. Decremental Approach for Mining Frequent Itemsets from Uncertain Data. *PAKDD* 2008.

[9] J. Han, J. Pei, Y. Yin. Mining frequent patterns without candidate generation. *SIGMOD* 2000.

[10] S. Guha, N. Koudas, K. Shim. Approximation and streaming algorithms for histogram construction problems. *ACM TODS*, 31(1), 396-438, 2006.

[11] C. K.-S. Leung, M. A. F. Mateo, D. A. Brajczuk. A Tree-Based Approach for Frequent Pattern Mining from Uncertain Data, *PAKDD* 2008.

[12] J. Pei, J. Han, et al. H-Mine: Hyper-Struction Mining of Frequent Patterns in Large Databases. *ICDM* 2001.

[13] Y.G. Sucahyo, R.P. Gopalan. CT-PRO: A Bottom-up Non-Recursive Frequent Itemset Mining Algorithm Using Compressed FP-Tree Data Structure. **URL:** [http://fimi.cs.helsinki.fi/src/].

[14] Q. Zhang, F. Li, and K. Yi. Finding Frequent Items in Probabilistic Data, *SIGMOD* 2008.