View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Institutional Knowledge at Singapore Management University

Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems | School of Information Systems

10-2001

Mining multi-level rules with recurrent items using FP'-Tree

Kok-Leong ONG

Wee-Keong NG

Ee Peng LIM
Singapore Management University, eplim@smu.edu.sg

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Databases and Information Systems Commons, and the Numerical Analysis and Scientific Computing Commons

Citation

ONG, Kok-Leong; NG, Wee-Keong; and LIM, Ee Peng. Mining multi-level rules with recurrent items using FP'-Tree. (2001). *Third International Conference on Information Communications and Signal Processing (ICICS 2001)*. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/904

This Conference Paper is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.

Mining Multi-Level Rules with Recurrent Items Using FP′-Tree

Kok-Leong Ong Wee-Keong Ng Ee-Peng Lim

Centre for Advanced Information Systems, Nanyang Technological University
Nanyang Avenue, Singapore 639798, SINGAPORE
awkng@ntu.edu.sg

Abstract

Association rule mining has received broad research in the academic and wide application in the real world. As a result, many variations exist and one such variant is the mining of multi-level rules. The mining of multi-level rules has proved to be useful in discovering important knowledge that conventional algorithms such as Apriori, SETM, DIC *etc., miss. However, existing techniques for mining multi-level rules have failed to take into account the recurrence relationship that can occur in a transaction during the translation of an atomic item to a higher level representation. As a result, rules containing recurrent items go unnoticed. In this paper, we consider the notion of 'quantity' to an item, and present an algorithm based on an extension of the* FP-Tree *to find association rules with recurrent items at multiple concept levels.*

1 Introduction

Association rule mining was introduced in 1993 [1] by Agrawal *et al.* and a year later, Agrawal and Srikant proposed the Apriori [2, 6] algorithm for fast association rule mining. Essentially, an association rule is a rule of the form $X \Rightarrow Y$ where $X, Y \subseteq \mathcal{I}$ and $\mathcal{I} = \{x_1, x_2, \ldots, x_n\}$, the set of unique items in the database $\mathcal{D}$ containing a set of transactions. A transaction $T \in \mathcal{D}$ is a tuple of the form $\langle tid, \{x_i, x_j, \ldots, x_k\}\rangle$, where *tid* is the transaction ID and $\{x_i, x_j, \ldots, x_k\} \subseteq \mathcal{I}$. Hence, a rule of the form {`sunshine-bread, marigold-butter`} $\Rightarrow$ {`daisy-milk`} can produce the interpretation *"A customer who buys Sunshine bread and Marigold butter also buys Daisy milk"*. Such a rule is valuable to the domain expert in many ways. For example, the manager may decide to shelf bread, butter and milk in close vicinity so as to enhance the shopping experience of the customer. Alternatively, the store manager may use the rule discovered to bundle goods such that the combination appeals to the consumers and that in turn, helped to increase sales and profit.

However, mining rules at the atomic (i.e., primitive) level proved to be less rewarding then mining rules at multiple concept levels. Han and Fu observe two problems with mining rules at the atomic level [4]. First, large support is likely to exist at a higher level concept rather than a low one. This means that it is difficult to find strong rules at the atomic level or one has to reduce the support threshold substantially. Reducing support creates two problems — the generation of too many rules, and the reduced efficiency due to a larger set of candidates to generate and test. Second, it is unlikely to find many strong rules at the atomic level due to the small average support for each atomic item within a large item set. As a result, mining of association rules should occur at different conceptual levels appropriate for the domain expert. With a multi-level view to the rules, a domain expert is likely to find knowledge that is useful within a shorter time frame. This is because the domain expert can proceed from any level, moving into details on concepts that are of interest to him or her.

While rules at different concept levels provide new insights to the basket data, we observed that the translation of an atomic item to its conceptual form results in recurrent items within a transaction. Current multi-level algorithms ignore this quantitative value. For example, if a transaction contains *strawberry* and *chocolate milk*, the translation results in a single conceptual item — *milk*. In the more precise case, the transaction should indicate two units of *milk*, instead of just *milk*. As observed by Zaiane *et al.* [11], important knowledge can be obtained by considering the recurrence behavior of objects within an image. In the case of the supermarket scenario, finding a pattern like three *beverages* appearing together with four *snacks* certainly gives more insight to the domain expert over just *beverages* and *snacks* occurring together.

To discover rules as described above, existing algorithms proposed in [2, 4] are inadequate. In this paper, we address the above in two steps. First, we introduce the notion of 'quantity' as an attribute to an item. Hence, every transaction now holds the item and its recurrence value. We then propose an algorithm based on the FP-Tree structure [5, 7] to mine multi-level rules containing recurrent items. Unlike the multi-level algorithms in [4, 9], which is fundamentally a generate and test process, the FP-Tree allows rules to be discovered without candidate generation. This reduces the mining of all levels to two database scans and a fraction of the memory required by the Apriori. We extend the algorithm for FP-Tree to mine multi-level rules with recurrent items. We called this extension the FP′-Tree.

The rest of the paper is organized as follows. In the next section, we discuss the concepts and notations of multi-level rules with recurrent items. Section 3 looks at the FP′-Tree, an extension of the efficient FP-Tree structure, to find multi-level rules with recurrent items. Finally, we conclude our work in Section 4.

2 Notations

Most algorithms on mining association rules use the *a-priori* property to traverse a part of the search space while ensuring that the rules discovered are complete. This efficiency is achieved by using two measures — *support* and *confidence*.

Definition 1 The **support** (φ) of an itemset X is the frequency that X appears in the database $\mathcal{D}$. A transaction T in $\mathcal{D}$ supports X (denoted by $X \preccurlyeq T$) if all elements of X appear in T. Unlike the Apriori, where each transaction contributes the same **support count** ($\widehat{\varphi}$, see **Definition 5**) to X (i.e., 1), each transaction contributes a different support count when recurrent items are considered. As a result, the support of an itemset X is given by the following expression:

$$\varphi(X) = \sum_{i=0}^{|\mathcal{D}|} \frac{\widehat{\varphi}(X, T_i)}{|\mathcal{D}|}$$

Definition 2 The **confidence** (σ) of a rule $X \Rightarrow Y$ is the strength of the statement "if X appears in a transaction, then Y can also be found in that transaction". Formally, $\sigma(X \Rightarrow Y)$ is the conditional probability of Y given that X has occurred, and is given by

$$\sigma(X \Rightarrow Y) = \frac{P(X \cup Y)}{P(X)}$$

Although a single support and confidence threshold can be specified for all levels, a better alternative is to define individual support and confidence measures for each level because the average support for each item increases as we ascend the concept tree. Formally, we observe the following about the support of an item at different concept levels.

Definition 3 Let $\alpha(p, q)$ be a function that returns `true` if p is a higher level concept of q. Given two items x_i and x_j at level ℓ_k that translates to the same conceptual item y_p at level ℓ_{k+1}, then $\varphi(y_p) = \varphi(x_i) + \varphi(x_j) - \varphi(\{x_i, x_j\})$. A generalization of the above is as follows:

$$\varphi(p \in \mathcal{I}_{\ell' > \ell}) \geqslant \sum \varphi(q \in \mathcal{I}_\ell \mid \alpha(p,q)) - |\beta(p)|$$

where $\beta(p)$ is defined as

$$\beta(p) = \{r \subseteq T_i \mid \forall r' \in r \rightarrow \alpha(p, r') \wedge |r| \geqslant 2\}_{i=0}^{|\mathcal{D}|}$$

which leads to the observation that

$$\forall \ell, \ell' \in L, \varphi(x_i \in \mathcal{I}_\ell) \leqslant \varphi(x_j \in \mathcal{I}_{\ell' > \ell})$$

As a result of the above, it makes sense to conclude the decision to have different support and confidence thresholds. In addition to the notion of support and confidence, the consideration of recurrent items changes the way two items are perceived as similar. While current algorithms recognize the similarity of two items by their string literal, we go beyond this scope by including the comparison of its quantity. This is necessary for the purpose of constructing unique nodes for storing the correct support count in the FP$'$-Tree. Therefore, in the context of the FP$'$-Tree, we have the following definitions.

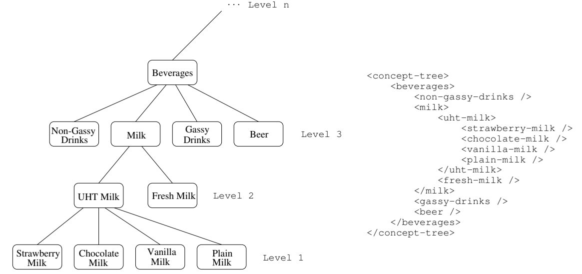

Figure 1: A concept tree and its XML representation.

Definition 4 Given an item x, the **recurrence value** (ψ) of x with respect to the transaction T and the itemset X is $\psi(x, T)$ and $\psi(x, X)$ respectively.

Definition 5 Given two items x_i and x_j in the FP$'$-Tree, x_i and x_j are **similar** (μ) if both x_i and x_j has the same literal, and are **equal** ($\widehat{\varrho}$) if and only if

$$\mu(x_i, x_j) \wedge \psi(x_i) = \psi(x_j).$$

Once items are differentiated, it is then possible to compute the support count. Most algorithm simply increments the support of an itemset if $X \preccurlyeq T$ holds. It does not consider how many times an itemset can be counted within the same transaction. Zaiane *et al.* noted this when the concept of recurrent objects in images were studied [11]. As mentioned in the *beverage/snack* example, we believe the notion of quantity is equally applicable in the transaction scenario. We observed that the support count of an itemset with recurrent items can be computed in the following manner.

Definition 6 Given a transaction T that supports an itemset X, the **support count** ($\widehat{\varphi}$) is given by

$$\widehat{\varphi}(X, T) = \min \left(\left\lfloor \frac{\psi(y \mid \mu(y \in T, x_i \in X))}{\psi(x_i \in X)} \right\rfloor \Bigg|_{i=0}^{|X|} \right)$$

We notice that the introduction of quantity (ψ) as an attribute of an item results in some implications. First, we can compute the support of an itemset X in $\mathcal{O}(|X|)$ time. This speedup, with respect to the technique in [11], is respectable when large number of itemsets needs to be checked against a huge database. Second, it reduces the number of physical nodes needed to represent all the patterns in the FP$'$-Tree. Consider a simple situation $X = \{x_i, x_i, x_i, x_j\}$. A naive modification of the FP-Tree will require four nodes to represent the pattern X. However, the FP$'$-Tree requires only two (i.e., $x_i(3), x_j(1)$) where the number within the bracket indicates the recurrence value of the item. Another important data structure that determines the abstraction of an item at some concept level. A number of ways to represent the concept tree exists [9, 10]. In this paper, we propose the use of XML where facilities to query and manipulate the XML structure already exists. Adding to the fact that XML documents are text files that can be easily edited, it becomes an attractive option for our purpose. Figure 1 shows the concept tree and the equivalent XML document. We define a function $\lambda()$ that uses the concept tree in Figure 1 as follows.

Definition 7 The **mapping function** (λ) translates a given transaction T, an itemset X or an item x to the concept level ℓ such that $\alpha(\lambda(T, \ell), t \in T)$, $\alpha(\lambda(X, \ell), x \in X)$ and $\alpha(x' = \lambda(x, \ell), x)$ holds respectively (See **Definition 3** for α).

Notice that it is possible to hold the FP$'$-Tree in memory due to its compact structure. Hence, it follows intuitively that each FP$'$-Tree at the higher concept level is smaller in size based on the number of nodes due to the merging of items into higher level concepts. If every FP$'$-Tree can be stored in main memory as assumed above, then the amount of database scan reduces to the initial requirement for constructing the base tree. Subsequent FP$'$-Trees are then constructed from the base tree and hence, no further I/O is required.

3 Mining multi-level rules with recurrent items

In the previous section, we have introduced the various notations and concepts for the purpose of mining multi-level rules with recurrent items using an extension of the FP-Tree. So far, the introduction in the last section focuses on defining the domain concepts. In this section, we present an algorithm to integrate the definitions. Due to space constraints, we shall briefly describe the fundamentals of the FP-Tree. The reader can refer to [5, 7] for a more complete discussion of the data structure and algorithm.

The frequent pattern tree (FP-Tree) is a tree like data structure that is a compact representation of the database. Since we are only interested in the frequent items, we can create a subset of the database by scanning it once and collecting items that meet or exceed the specified support threshold. Suppose that this structure is small enough, we will be able to store the representation in the main memory and avoid the need to pay for I/O overheads that the Apriori requires. Furthermore, with frequent items repeating in the transactions, it is actually possible to store the frequent item only once and the occurrences in the database registered. The FP-Tree takes advantage of this property and allows transactions to share a common prefix among the common items. For example, if $T_1 = \{a, b, d, e, h\}$ and $T_2 = \{a, b, d, g, h\}$ then the common prefix of the two transactions is $\{a, b, d\}$ and we only need to stored it once. The constructed FP-Tree is similar to the one shown in Figure 3 except that the FP-Tree structure does not have a "Transaction count" table.

Instead of candidate generation and then paying for the I/O cost to determine if a pattern is frequent, the frequent pattern growth algorithm traversed the FP-Tree to determine the set of frequent patterns. To generate a rule containing an item x, the algorithm begins from the header table and finds all paths containing x. The prefix nodes of x are then extracted to form the conditional pattern base. Thinking for a moment that each conditional pattern base is a transaction, we can construct a conditional pattern tree like how we construct the initial FP-Tree. The paths in the conditional pattern tree are then enumerated to obtain all combinations of patterns containing the item x. We continue this principal idea and applied the procedure to mine patterns containing recurrent items. An example will be discussed in Section 3.3.

Input: Transaction database with recurrent items $\mathcal{D}$, support threshold (φ_1) at the atomic level.

Output: The base tree, FP'-Tree$_{\varphi_1}$

Method:

1. Scan $\mathcal{D}$ and collect the support count of each 1-itemset X using $\widehat{\varphi}(X, T \in \mathcal{D})$. Add the frequent items that satisfies φ_1 to F_ℓ and sort F_ℓ in descending order.

2. Create the root of the FP'-Tree $\mathcal{T}$, and label it as `null`. For each transaction $T \in \mathcal{D}$, do the following.

   (a) Let $T' = \{x \mid \mu(x \in T, y \in F_\ell)\}$ where x is sorted according to the order of F_ℓ.

   (b) Let the sorted frequent items in T' be $[H'|h|H]$, where H' is the set of ancestor nodes of h, and H the remaining elements in the list. Call $insertTree([\texttt{null}|h|H], \mathcal{T})$.

   (c) The function $insertTree()$ performs the following. If $\mathcal{T}$ has a child N such that $\widehat{\varrho}(N, h)$ holds, then increment N's count by 1; else create a new node N and set its count to 1, it's parent link be linked to $\mathcal{T}$, the parent node's `next_ptr` set to N, and its node-link to the nodes that meet **Definition 4** via the node-link structure. Then, for each path $p \in \mathcal{T} - \{[H'|h|H]\}$, $\mathcal{H} = H' \cup \{h\}$, $\mathcal{P} = p \cap_\mu \mathcal{H}$, and $\forall e_h \in \mathcal{H}, \exists e_p \in p \rightarrow \mu(e_h, e_p)$:

      i. If $\mathcal{H} \subset_\varphi \mathcal{P}$ and N is newly created then $h.\texttt{count} \mathrel{+}= \widehat{\varphi}(\mathcal{H}, \mathcal{P})$

      ii. If $\mathcal{H} \supset_\varphi \mathcal{P}$ then $q.\texttt{count} \mathrel{+}= \widehat{\varphi}(\mathcal{P}, \mathcal{H})$ if and only if $\exists q \in p \rightarrow \mu(q, h)$ and $\exists p' \in \mathcal{P} \rightarrow parent(p') = \texttt{null}$ and $\mathcal{P}$ is a consecutive pattern of p.

      iii. If $\mathcal{H} =_\varphi \mathcal{P}$ then

         If $|\mathcal{H}| < |p|$ and N is newly created then $h.\texttt{count} \mathrel{+}= \widehat{\varphi}(\mathcal{H}, \mathcal{P})$

         If $|\mathcal{H}| > |p|$ then $q.\texttt{count} \mathrel{+}= \widehat{\varphi}(\mathcal{P}, \mathcal{H})$ if and only if $\exists q \in p \rightarrow \mu(q, h)$ and $\exists p' \in \mathcal{P} \rightarrow parent(p') = \texttt{null}$ and $\mathcal{P}$ is a consecutive pattern of p.

   (d) If h is the last node in the path p and $h.\texttt{next_ptr}$ is `null` (i.e., the node for h has been just created), then set $h.\texttt{next}$ to the address of the constant `1` in the path count table; else set $h.\texttt{next}$ to the address of the constant that is 1 higher than the current value pointed by $h.\texttt{next}$.

   (e) Call $insertTree([H' \cup \{h\}|h'|H - \{h'\}], N)$ recursively if H is non-empty.

Figure 2: Algorithm for base tree construction.

3.1 Generating the Base Tree

Assume that the set of support and confidence thresholds are already defined by the user, and that the concept tree represented in XML has already been created. The process begins with the construction of the base tree. We begin by running the algorithm in Figure 2 using the support threshold defined for the atomic level (φ_1). In this case, we have to consider the recurrence value of an item within the transaction resulting in the difference between our algorithm versus the original.

First, each transaction T now holds the item and its recurrence value. When computing the frequent 1-itemsets, the respective support count are obtained using $\widehat{\varphi}(X, T)$ where X is the frequent 1-itemset. In the second pass through the database, items in the transaction are ordered based on the frequency of their 1-itemset. The algorithm has been modified to use the definition of **equal** ($\widehat{\varrho}$) and the support computation ($\widehat{\varphi}$) defined earlier. For illustration purposes, the FP$'$-Tree$_{\varphi_1}$ constructed in Figure 3 is the result of running the base tree construction algorithm over the database in Table 1 with a support count of 3. Each transaction in

TID	Transaction Details
100	b(3), a(2), e(1)
200	b(1), e(1)
300	b(3), e(1)
400	b(9), e(5)
500	b(6), a(4)
600	b(9), f(3)
700	b(6), a(4)

Table 1: The sample database.

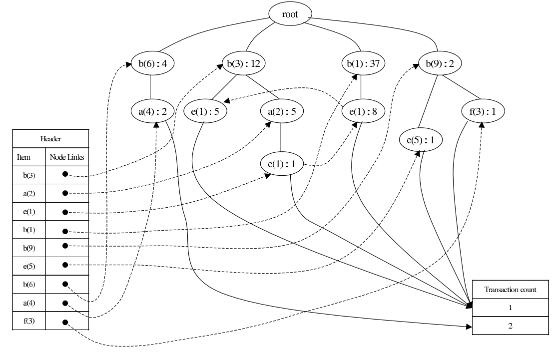

Figure 3: The FP'-Tree.

the database now contains the item and its recurrence value within the brackets.

In the first pass, the support count of each item is obtained and placed into F_ℓ. For our example, $F_\ell = \{$b:37, a:10, e:8, f:3$\}$. Since the support count is 3, all the items at the current level qualify for the FP$'$-Tree construction (the number after the colon indicates the support count). In the second pass, we read each transaction, prune items not in F_ℓ, and ordered them according to the sequence in F_ℓ (step 2(a)). Starting with the item in T' having the highest support count for its 1-itemset, we insert the item into the FP$'$-Tree according to the definition of step 2(b) and 2(c). Beginning with transaction **100**, we first prune items in the transaction not in F_ℓ. Since all items are in F_ℓ, we next arrange them in the order of F_ℓ. Starting with the first item 'b(3)', and that the FP$'$-Tree is now empty, we create a new node $N = \langle$b(3):1$\rangle$, set its support count to 1, set the `next_ptr` to `null`, set the parent pointer to the root of the tree and update the node link structure. Since there are no other paths other than itself, steps (i), (ii) and (iii) are not executed. This continues for the other two nodes in T'. When the last node 'e(1)' is inserted, it's `next_ptr` is set to the address where the constant 1 is located in the "Transaction count" table.

Proceeding on with the second transaction, we create a node representing 'b(1)' since it does not exist at the root of the tree. Now that a path other than the one that the current node exists, we need to perform steps (i) to (iii). The symbol $\cap_\mu$ represents the intersection of two sets based on their similarity while $X \subset_\varphi Y$ holds if $\forall x \in X, \exists y \in Y \rightarrow \varphi(x) \leqslant \varphi(y) \wedge \mu(x, y)$. Likewise for $X \supset_\varphi Y$. Once $\langle$b(1):1$\rangle$ is created, we have $\mathcal{H} = \{\texttt{null}\} \cup \{\langle\text{b(1):1}\rangle\}$, $\mathcal{P} = \{$b(3):1$\}$. Hence, step (i) is executed and the support count of 'b(1)' is incremented by $\widehat{\varphi}(\mathcal{H}, \mathcal{P})$ updating the node to $\langle$b(1):4$\rangle$. The reason for updating the count lies in the way recurrent items are counted. In this case, since *one* b occurs *three* times in 'b(3)', its support count due to transaction **100** is 3. Add to the support count due to transaction **200**, we have a total support count seen by 'b(1)' to be 4. When 'e(1)' is added, $\mathcal{H} = \{$b(1):4, e(1):1$\}$ and $\mathcal{P} = \{$b(3):1, e(1):1$\}$. Again, step (i) is executed and *only* the support count of 'e(1)' is incremented. This continues for all transactions until every transaction is processed and placed into the tree.

Note that step (c) is where most of the extension takes place. Further details of step (c) can be analyzed from the algorithm. However, it would be sufficient for the reader to understand that the principle of step 2(c) is to correct the support count due to the difference in computation of support against transactions that occur before and after its insertion. Once the base tree is constructed, we can proceed to generate the atomic level rules from the base tree using the enumeration technique discussed in [5]. Before we go into this, we look at the construction of higher level FP$'$-Tree from the base tree.

3.2 FP$'$-Tree Construction

While the FP-Tree discovers rules at the atomic level, it does not perform mining at multiple concept levels. The trivial approach to provide this facility with FP-Tree is to work on an interface that rescans the database every time a different concept level is to be mined. This translates to a total of $2+\ell$ passes where ℓ is the number of levels to mine. With typical memory at orders of magnitude faster than the disk, the extra ℓ passes through the database is clearly undesirable. Hence, we seek an alternative where we construct a higher concept FP$'$-Tree from its lower level concept FP$'$-Tree. Although the algorithm presented in this paper includes the notion of recurrent items, deriving a version for the FP-Tree is relatively straightforward.

Figure 4 shows the algorithm to construct a FP$'$-Tree given concept level ℓ' and a support threshold $\varphi_{\ell'}$. Let ℓ be a lower concept level of ℓ' (i.e., $\ell < \ell'$). In step 1, we generate $F_{\ell'}$ from F_ℓ by first mapping each item in F_ℓ to its high level representation at ℓ'. We then sum the support count of repeating concepts, pruning away items in $F_{\ell'}$ that do not meet $\varphi_{\ell'}$. Finally, we sort $F_{\ell'}$ in descending order. In step 2, we use the FP$'$-Tree at level ℓ to create transactions holding the conceptual items at level ℓ'. This is achieved by abstracting each node in the FP$'$-Tree to level ℓ, deleting those that do not appear in $F_{\ell'}$, and removing duplicates by combining the similar ones and their recurrence value. This is done in steps 2(a), 2(b) and 2(c). When all nodes in the path $p \in \mathcal{T}_\ell$ have been translated, we call *insertTree*() by the number of times indicated by the "Transaction Count" table. This repeats until all paths in the FP$'$-Tree at level ℓ are traversed.

3.3 Finding Frequent Itemsets

The enumeration process to finding frequent pattern is similar to that of the FP-Tree. For the sake of completeness,

Input: Concept tree at level ℓ $\mathcal{T}_\ell$, support threshold for level ℓ' ($\varphi_{\ell'}$).

Output: The FP'-Tree$_{\varphi_{\ell'}}$ for level ℓ'

Method:

1. Let F_ℓ be the set of frequent 1-itemsets for the concept level ℓ. Let $F_{\ell'} = \{\varnothing\}$. For each $x \in F_\ell, x' = \lambda(x, \ell')$, if $F_{\ell'}$ does not contain x' then insert x' into $F_{\ell'}$ and set its support to that of x; else increment the support of x' in $F_{\ell'}$ by the support of x.

2. For each $x' \in F_{\ell'}$, if $\varphi(x') < \varphi_{\ell'}$ then remove x' from $F_{\ell'}$. Sort $F_{\ell'}$ in descending order.

3. For each path $p \in \mathcal{T}_B$, create a new path p'. Then, for each node $n \in p$, do the following:

   (a) Create a new node n' such that $n' = \lambda(n, \ell)$.

   (b) Set $\psi(n') = \sum \psi(m_i \mid \mu(n, m_i))|_{i=0}^{|p|}$.

   (c) If $n' \notin p'$ then $p' = p' + \{n'\}$; else $\psi(m' \in p') = \psi(m') + \psi(n')$ where $\mu(m', n')$ holds. Let $p = p - \{m \mid \mu(n', m)\}$ and repeat step (a) and (b) until $p = \{\varnothing\}$.

   (d) Let the sorted items in p' be $[H'|h|H]$ where H', h and H are defined as in step (b) of the base tree construction algorithm. Repeat the call $insertTree([\texttt{null}|h|H])$ to do step (c) and (d) of the base tree construction algorithm by the number of times indicated by the `next_ptr` of the last node of p.

Figure 4: Algorithm for concept tree construction.

we illustrate with an example. Suppose we are interested in finding frequent patterns containing 'e(1)' (i.e., itemsets with a single item e) and that the support threshold is 3. Following the "Header" table, we derive ⟨e(1):1⟩, ⟨e(1):8⟩, ⟨e(1):5⟩ and three paths in the FP′-Tree: ⟨b(3):10, a(2):5, e(1):1⟩, ⟨b(1):37, e(1):8⟩ and ⟨b(3):10, e(1):5⟩. The first path appears once in the database by definition of $\widehat{\varphi}$ while the other paths appears 8 and 5 times respectively.

We first derive the conditional pattern base. In this case, the conditional pattern base for 'e(1):1', 'e(1):8' and 'e(1):5' is ⟨b(3):1, a(2):1⟩, ⟨b(1):8⟩ and ⟨b(3):5⟩ respectively. Constructing the conditional FP′-Tree, leads to two branches: ⟨b(1):8⟩ and ⟨b(3):5⟩. Hence, the frequent patterns containing 'e(1)' and has a support count exceeding the threshold are ⟨b(1), e(1)⟩:8 and ⟨b(3), e(1)⟩:5. Using the same principle, all rules in the database can be discovered on the fly by going through the header table once.

4 Conclusions

The defining work [4, 10] on multi-level concept mining has explored various algorithms based on an extension of the *Apriori*. Around the same time, the notion of recurrent items was proposed for mining objects in an image where objects occurring more than once has an implicit meaning. We believe the notion of recurrent items in multimedia data is equally applicable to the quantity of an item in a supermarket scenario, and the abstraction process should maintain this information. While the solution to mining multi-level rules and recurrent items exists, they are developed along separate paths. An algorithm that combines the two variants appear lacking. Moreover, both techniques use a variation of the Apriori and thus suffers from the bottleneck of candidate generation when the support threshold is low.

Recently, the FP-Tree and frequent pattern mining technique was proposed as an alternative to the Apriori. In particular, frequent pattern mining eliminates the need for candidate generation and test. Hence, empirical tests conducted showed dramatic improvements in performance over the Apriori in terms of speed, I/O and memory consumption. However, the FP-Tree only supports mining of primitive patterns and facilities for mining recurrent items and multi-level rules are missing. We noted the potential of extending the algorithm, much like how Apriori was extended for the multi-level and the recurrence case. This paper reports the results of our work done.

References

[1] R. Agrawal, T. Imielinski and A. Swami. "Mining Association Rules Between Sets of Items in Large Databases", in *Proc. of the ACM Int. Conf. on Management of Data*, Washington, USA, May 1993.

[2] R. Agrawal and R. Srikant. "Fast Algorithm for Mining Association Rules", in *Proc. of the 20th VLDB Conference*, Santiago, Chile, 1994.

[3] Y. Fu and J. Han. "Meta-Rule-Guided Mining of Association Rules in Relational Databases", in *Int. Workshop on Knowledge Discovery and Deductive and Object-Oriented Databases*, Singapore, Dec. 1995.

[4] J. Han and Y. Fu. "Discovery of Multiple-Level Association Rules from Large Databases", in *Proc. of the 21st VLDB Conference*, Zurich, Switzerland, 1995.

[5] J. Han and J. Pei. "Mining Frequent Patterns by Pattern-Growth: Methodology and Implications", in *SIGKDD Explorations* **2**(2), Dec. 2000.

[6] J. Hipp, U. Guntzer, G. Nakhaeizadeh. "Algorithms for Association Rule Mining — A General Survey and Comparison", in *SIGKDD Explorations* **2**(1), Jun. 2000.

[7] J. Han, J. Pei and Y. Yin. "Mining Frequent Patterns without Candidate Generation", in *Proc. of the ACM Int. Conf. on Management of Data*, Dallas, TX, May 2000.

[8] G. Psaila and P. L. Lanzi. "Hierarchy-based Mining of Association Rules in Data Warehouses", in *Proc. of the ACM Symp. on Applied Computing*, Como, Italy, Mar. 2000.

[9] R. Srikant and R. Agrawal. "Mining Quantitative Association Rules in Large Relational Tables", in *Proc. of the ACM Int. Conf. on Management of Data*, Montreal, Canada, Jun. 1996.

[10] S. Thomas and S. Sarawagi. "Mining Generalized Association Rules and Sequential Patterns Using SQL Queries", in *Proc. of the 4th Int. Conf. on Knowledge Discovery and Data Mining*, New York, 1998.

[11] O. R. Zaiane, J. Han and H. Zhu. "Mining Recurrent Items in Multimedia with Progressive Resolution Refinement", in *Proc. of Int. Conf. on Data Engineering*, San Diego, Mar. 2000.