On Mining Max Frequent Generalized Itemsets

Gene Cooperman  Donghui Zhang  Daniel Kunkle
College of Computer & Information Science
Northeastern University, Boston, MA 02115
{gene, donghui, kunkle}@ccs.neu.edu

Abstract. A fundamental task of data mining is to mine frequent itemsets. Since the number of frequent itemsets may be large, a compact representation, namely the max frequent itemsets, has been introduced. On the other hand, the concept of generalized itemsets was proposed. Here, the items form a taxonomy. Although the transactional database only contains items in the leaf level of the taxonomy, a generalized itemset may contain some non-leaf generalized items. Naturally, an interesting question arises: can we combine the two concepts and develop algorithms to mine max frequent generalized itemsets? This is a compact representation of the set of all frequent generalized itemsets. To the best of our knowledge, this paper is the first work that efficiently solves this new problem. Our solution has the following components: a conceptual classification tree, the algorithm MFGI\texttt{class} that dynamically generates the needed part of the conceptual classification tree by applying three pruning techniques, an online method for checking false positives, and an optimization technique called PHDB that batch-compute frequencies. Besides market-basket analysis, our technique has a vast range of other applications. For instance, identifying associations among (categories of) diseases, identifying associations among (groups of) occupations, and so on. Experimental results demonstrate the efficiency of our approach.

1 Introduction

A fundamental task in data mining is to mine frequent itemsets (e.g. bread, egg and milk appear together in over 1% of all transactions). Since the number of frequent itemsets may be large, Bayardo [5] proposed the concept of max frequent itemsets: a frequent itemset without frequent proper superset. It is a compact representation of the set of all frequent itemsets. Each max frequent itemset represents an exponential number of frequent itemsets (all of its subsets). And from the set of max frequent itemsets, one can derive the complete set of frequent itemsets without accessing the transactional database.

Informally, one is given a transactional database. Each transaction is a set of items. An itemset is a set of items with which one queries the database. One wishes to extract from the database the family of all frequent itemsets, or the more compact family of max frequent itemsets. The support (also called

* Submitted EDBT’06, paper #556.
frequency) of an itemset is the number of transactions containing that itemset. The itemset is frequent if its support or frequency is at least minsupport.

Another extension to the problem of mining frequent itemsets is to consider generalized items, or g-items in short. Here, the set of all g-items form a taxonomy $T$, which is a tree structure. For instance, the items ‘apple’ and ‘strawberry’ may have a common parent in $T$ called fruit. A transactional database continues to involve only leaf-level g-items (ordinary items) from $T$. However, a g-item at higher levels of $T$ may also appear in itemsets. As an example, \{fruit, bread\} may be a frequent itemset, even though neither \{apple, bread\} nor \{strawberry, bread\} is frequent. The extended concept is called a generalized itemset (g-itemset), and there are existing solutions to mine frequent generalized itemsets [11, 15, 16, 18].

Similarly, it is desirable to extend the concept of max frequent itemsets to the generalized case when there is a taxonomy. Just as for frequent itemsets, the set of all frequent g-itemsets benefit from being more compactly represented as the set of max frequent g-itemsets. Of course, this benefit depends on whether we can find an efficient algorithm that directly finds the set of max frequent g-itemsets. In other words, an algorithm is meaningless if it first finds the set of g-itemsets and then derives max frequent g-itemsets from it. Unfortunately, the introduction of g-items brings some special requirements, and thus the solutions to the ordinary cases do not apply.

For this new problem, one can think of a straightforward solution based on browsing the lattice of g-itemsets. The algorithm is discussed in Section 4. However, we demonstrate, both analytically and experimentally, that this is an inefficient approach.

In the ordinary case of mining max frequent itemsets, a relatively simple classification tree exists [5]. In particular, all proper subsets of an itemset $ABCD$ are classified into four classes: (i) those containing $A$; (ii) those containing $B$ but not $A$; (iii) those containing $C$ but not $A$ or $B$; and (iv) those containing only $D$. But such a classification is not clear in the generalized case.

The novelty of this paper is that we efficiently solve, for the first time, the problems of mining max frequent g-itemsets. Our solution is a classification-tree-based algorithm. Due to the complexity of the generalized environment, this tree is more complex than the MaxMiner tree [5]. The classification tree (Section 5.1) is conceptual, in the sense that no pruning of the tree is employed. By employing three pruning techniques, our proposed algorithm MFGI_class (Section 5.2) generates only a small part of the classification tree, in order to finds all max frequent g-itemsets. As part of the solution, a method to remove false positives in an online fashion is provided (Section 5.3). Finally, we address the problem of frequency computation. It is clearly not efficient to scan through the transactional database for computing each individual frequency. We provide an optimization technique called PHDB (Section 5.4) which aims to reduce the number of database scans by batch-compute frequencies.

Our methods apply to a broad range of applications. As long as there is a taxonomy of items and a list of transactions (each containing some leaf items
of the taxonomy), the application can benefit from our methods. Here are some example taxonomies:

- Diseases have a natural hierarchy, e.g. as defined by the Merck Manual of Medical Information (http://www.merck.com/mmhe/index.html).
- Occupations have a hierarchy, e.g. as classified by the U.S. Department of Labor (http://www.bls.gov/oes/current/oes_stru.htm).
- Companies have a hierarchy, e.g. as assigned by the North American Industry Classification System (NAICS) (http://www.census.gov/epcd/www/naics.html).
- URLs have a taxonomy, e.g. as provided by DMOZ (http://dmoz.org).

Our methods enables the efficient computation of associations among (categories of) diseases, associations among (groups of) occupations, association among (types of) companies, associations among (classes of) URLs, and so on.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 provides the problem definition. Section 4 briefly discusses the straightforward lattice-based solution. The core of the paper is Section 5 which provides the classification-based solution. Section 6 shows experimental results. Finally, Section 7 concludes the paper.

2 Related Work

Table 1 shows a structured view of selected work on mining frequent itemsets. The table entry (frequent itemsets, ordinary) shows earlier work on mining frequent itemsets (Section 2.1). The table entry (max frequent itemsets, ordinary) lists work on mining max frequent itemsets, as described in Section 2.2. Also in Section 2.2, we review another compact representation of frequent itemsets, namely the closed frequent itemsets. In the generalized case, there exists work on mining frequent g-itemsets and closed frequent g-itemsets, as discussed in Section 2.3. Table 1 highlights that there is a blank table entry with no existing work: to mine max frequent g-itemsets. This paper fills the blank.

<table>
<thead>
<tr>
<th></th>
<th>Ordinary</th>
<th>Generalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequent itemsets</td>
<td>[3, 4, 10], etc.</td>
<td>[11, 15, 16, 18]</td>
</tr>
<tr>
<td>max frequent itemsets</td>
<td>[2, 5, 6, 8, 12]</td>
<td>this paper</td>
</tr>
<tr>
<td>closed frequent itemsets</td>
<td>[13, 14]</td>
<td>[17, 18]</td>
</tr>
</tbody>
</table>

Table 1. Structured view of selected results on mining frequent itemsets.
2.1 Earlier Work on Mining Frequent Itemsets

The concept of mining frequent itemsets was first introduced by Agrawal et al. [4]. Earlier works on mining frequent itemsets focused on the Apriori algorithm and its improved versions. Recently, Han et al. [10] proposed the FP-tree technique, which finds frequent itemsets without generating candidates. Agarwal et al. [3] proposed the depth-first generation of frequent itemsets. Note that we only pick a few representatives here, as there are hundreds of papers on this topic.

2.2 Two Compact Representations of Frequent Itemsets

Mining max frequent itemsets was introduced by Bayardo [5], where the MaxMiner algorithm was presented. The Pincer algorithm was proposed by Lin and Kedem [12]. The MAFIA algorithm was proposed by Burdick et al. [6], which finds supersets of max frequent itemsets. Gouda and Zaki [8] proposed GenMax and evaluated it against MaxMiner and MAFIA. They concluded that GenMax is better for finding the exact set of max itemsets, and that, depending on the dataset, MaxMiner may be the optimal. A depth-first search with pruning was proposed by Agarwal et al. [2]. The reason why there is extensive work on this topic is that max frequent itemset is a good compact representation for the set of all frequent itemsets. This is one motivation for us to design good methods to find max frequent g-itemsets while considering general items. Unfortunately, these existing works cannot be easily applied to the generalized environment. Another compact representation of all frequent itemsets was that of closed frequent itemsets, introduced by Pasquier et al. [13]. Later, another solution to the closed itemset problem was proposed by Pei et al. [14].

Both compact representations, the max frequent itemsets and the closed frequent itemsets, have value. It is unfair to say one representation is always better than the other. The representation of closed frequent itemsets retains more information in the sense that, from the closed compact representation, one can not only derive the set of all frequent itemsets, but also determine their support. On the other hand, the representation of max frequent itemsets is much more compact.

We can safely state that choosing max frequent itemsets is better if one only wants to find the set of frequent itemsets without caring about their support. Furthermore, if the support is needed for specific frequent itemsets, one still retains the option of a scan of the database. In this paper we look at taxonomies of items for max frequent itemsets.

2.3 Mining Generalized Itemsets

The problem of mining generalized itemsets was first introduced by Srikant and Agrawal [15]. They proposed three algorithms for frequent generalized itemsets: Basic, Cummulate and EstMerge. Later, Hipp et al. [11] proposed the Prutax algorithm. The idea was to utilize the vertical database format (for every item, store
the IDs of transactions that involve the item). Sriphaew and Theeramunkong [16, 18] proposed the SET algorithm to find generalized itemsets.

As a generalization of the mining of closed frequent itemsets to the case when considering g-items, the cSET algorithm was proposed to mine closed frequent g-itemsets [17, 18].

Han and Fu [9] proposed the multiple-level itemsets. Similarly to g-itemsets, a taxonomy $T$ is involved. But a multi-level itemset only contain items from the same level of $T$.

3 Problem Definition

To make the paper self-contained, we describe the concept of mining frequent generalized itemsets which was introduced in [15]. The problem of mining max frequent generalized itemsets is defined towards the end of the section.

3.1 Generalized Itemsets

The set of all items form a taxonomy $T$, which is a tree structure. An example is shown in Fig. 1(a). Here items $D$, $E$ and $F$ may represent apple, peach and strawberry, respectively; $Z$ represents fruit, while $X$ represents food. A transactional database $D$ (e.g. Fig. 1b) is a list of transactions, each of which contains some items from the leaf level of $T$.

\[
\begin{array}{|c|}
\hline
\text{Transactions} \\
1 \text{BCD} \\
2 \text{ABC} \\
3 \text{ABCDEF} \\
4 \text{E} \\
5 \text{CE} \\
\hline
\end{array}
\]

(a) taxonomy $T$  \hspace{1cm} (b) transactional database $D$

Fig. 1. Taxonomy and database.

**Definition 1** Given a taxonomy $T$, a generalized itemset, or g-itemset in short, is a non-empty set of g-items from $T$, where no two of the g-items have an ancestor-descendent relationship in $T$.

Hence, $ACZ$ in Fig. 1(a) is a g-itemset, but $ACDZ$ is not. By convention, an itemset is not empty. Intuitively, anybody who bought an apple (or peach or strawberry) is considered to have also bought fruit. So the set $\{\text{apple, fruit}\}$ is not compact, and so is not considered a g-itemset. (The equivalent compact itemset is $\{\text{apple}\}$. ) We need any itemset to be in its most compact representation.
3.2 Item-Itemset and Itemset-Itemset Relationships and Operators with regards to $T$

Given an item $i \in T$ and a g-itemset $S$, we say $i$ belongs to $S$ with respect to $T$, denoted as $i \in_T S$, if $\exists j \in S$ such that $i = j$ or $i$ is an ancestor of $j$ in $T$. For example, in Fig. 1(a), $Z \in_T \{WCD\}$, as $Z$ is an ancestor of $D$ in $T$. Intuitively, we might assign $W$, $C$, $D$ and $Z$ to beef, chicken, apple and fruit. Hence, any transaction that contains beef, chicken and apple is considered to contain fruit (due to the apple).

To motivate the definition of $i \in_T S$, note that for ordinary itemsets, the notation $A \in AB$ has the interpretation that $A \rightarrow B$ (any itemset containing $A$ and $B$ must in fact contain $A$) and that the support of $AB$ is less than or equal to the support of $A$. In a similar vein, if $A$ is a child of $W$ in the taxonomy $T$, we write $W \in_T T A$. This is motivated, since the logical implication $A \rightarrow W$ (any itemset containing $A$ must contain $W$) holds and so the support of $A$ is less than or equal to the support of $W$.

Given two g-itemsets $S_1$ and $S_2$, we say $S_1$ is a subset of $S_2$ with respect to $T$, denoted as $S_1 \subseteq_T S_2$, if $\forall i \in S_1$, $i \in_T S_2$. For example, in Fig. 1(a), $\{WC\} \subseteq_T \{ACD\}$. Among all g-itemsets, the largest one is $\{ABCDEF\}$, and the smallest one is $\{X\}$. That is, for any g-itemset $S$, we have: $\{X\} \subseteq_T S \subseteq_T \{ABCDEF\}$. We also have the proper subset notation ($\subset_T$) with its obvious meaning.

The union and intersection operators with respect to $T$ are also defined for two g-itemsets $S_1$ and $S_2$. $S_1 \cup_T S_2$ is the smallest g-itemset that is a superset of both $S_1$ and $S_2$ with respect to $T$. $S_1 \cap_T S_2$ is the largest g-itemset that is a subset of both $S_1$ and $S_2$ with respect to $T$. For instance, in Fig. 1, $\{WC\} \cup_T \{BD\} = \{BCD\}$, $\{WC\} \cap_T \{BD\} = \{W\}$. (If $W$, $B$, $C$, and $D$ are beef, veal (a type of beef), chicken and apple, then the union of “beef and chicken” with “veal and apple” is “veal, chicken and apple”. Beef is implicitly part of that union, since the union contains veal, a kind of beef. Also, the intersection of “beef and chicken” with “veal and apple” is “beef”, since both types of transactions contain beef.)

3.3 Max Frequent Generalized Itemsets

The support of a g-itemset $S$ is the percentage of transactions in $D$ that are supersets of $S$ with respect to $T$. For instance, in Fig. 1, the support of $\{WC\}$ is 60%. The reason is that among the five transactions, three of them contain either $\{AC\}$ or $\{BC\}$, and thus are supersets of $\{WC\}$ with respect to $T$. A g-itemset is frequent if its support is above a given threshold $\text{minsupport}$.

**Definition 2** Given a taxonomy $T$, a transactional database $D$, and a threshold $\text{minsupport}$, a **max frequent g-itemset** is a frequent g-itemset without frequent proper superset with respect to $T$.

We are interested in efficiently mining the set of all max frequent g-itemsets.
4 The Lattice-based Solution and Its Drawback

A straightforward solution to mine max frequent g-itemsets is to dynamically browse a lattice of g-itemsets.

A lattice can be defined by a set and a partial-order operator between elements in the set. In our case, all g-itemsets form a lattice. The partial-order operator is $\supset_T$. There is an edge from element $S_1$ to $S_2$ if $S_1 \supset_T S_2$ and $\not\exists S_3$ s.t. $S_1 \supset_T S_3 \supset_T S_2$. In this case, $S_1$ is said to be a parent of $S_2$ in the lattice, and $S_2$ is called a child of $S_1$. As an example, given the taxonomy of Fig. 2(a), the corresponding lattice of g-itemsets is shown in Fig. 2(b).

The lattice-based algorithm that mines max frequent g-itemsets, which we call MFGI\textsubscript{lattice}, can be summarized below. Starting with the root of the lattice, we dynamically browse the lattice in a top-down fashion. Whenever we meet a frequent g-itemset, it is a candidate of max frequent g-itemset and we do not browse its children (or descendants).

It remains to discuss how to dynamically generate the children of each g-itemset. Given a g-itemset $S_1$, the method to generate its children is: replace a g-item in $S_1$ by its parent in the taxonomy $T$. This replacement should be followed by an attempt to compact the g-itemset. That is, if the new g-item $i$ is an ancestor in $T$ of some other g-item in the set, the item $i$ should be removed.

For example, in Fig. 2(b), $BC$ is a child of $ABC$, by generalizing $A$ to $W$ (its parent in $T$). The generalization results in $WBC$, which is not compact. The reason is that $W$ and $B$ have ancestor-descendant relationship in $T$. In this case, we remove $W$, and the resulted child is $BC$. If we further generalize $B$, we get a grandchild $WC$.

The problem with MFGI\textsubscript{lattice} is that it is not efficient. As an example, if all max frequent g-itemsets appear near the bottom of the lattice, the algorithm needs to check the frequency of almost all g-itemsets! In the rest of this paper,
we focus on developing a much more efficient algorithm based on our novel classification tree.

5 The Classification-based Solution

The classification-based solution has four components. Section 5.1 defines a conceptual classification tree. Section 5.2 describes the algorithm \texttt{MFGI\_class} which dynamically generates the needed part of the tree, while pruning entire branches using three pruning techniques. Since this algorithm produces a superset of the max frequent g-itemsets, which includes false positives, Section 5.3 describes how to efficiently eliminate those false positives in an online fashion. Finally, Section 5.4 presents the \texttt{PHDB} technique such that multiple frequencies will be computed for each database scan.

5.1 The Conceptual Classification Tree

This section provides a conceptual classification tree. Every g-itemset corresponds to exactly one leaf node in the tree. An index node also corresponds to a g-itemset, which is a superset of all g-itemsets in the sub-tree.

We emphasize that the classification tree is only conceptual, in the sense that to mine max frequent g-itemsets, only (a small) part of the tree needs to be generated, with appropriate pruning. The tree will be dynamically generated in a top-down fashion. Suppose we are examining an index node and we find that its corresponding g-itemset, \( I \), is frequent. Since all g-itemsets in the sub-tree are subsets of \( I \), no g-itemset in the sub-tree except \( I \) can be a candidate for max frequent g-itemsets. Thus there is no need to generate the sub-tree.

Before describing the mining algorithm, in this section we focus on defining the complete classification tree, without considering pruning. An example of a classification tree is shown in Fig. 3.

The Form of A Tree Node A node \( N \) in the classification tree has the following form: \((S_1)(S_2)(S_3)\). Here, \( S_1, S_2 \) and \( S_3 \) are g-itemsets. The meanings are:

- **MUST-LITERALLY-HAVE-ALL-OF** \((S_1)\): Every g-itemset in \( \text{subtree}(N) \) must contain every g-item in \( S_1 \). For instance, in Fig. 3, \((WC)(())\) is a leaf node in the classification tree, which corresponds to a g-itemset \( WC \).

- **MUST-HAVE-PART-OR-ALL-OF** \((S_2)\): In our method, the \( S_2 \) part consists of zero or one g-items. If it contains a g-item \( i \), some g-item in \( \text{tax}(i) \) must appear in each g-itemset in \( \text{subtree}(N) \). Here, \( \text{tax}(i) \) denotes the sub-taxonomy rooted by \( i \). For instance, the root node of the classification tree is \((())(Y)()\), where \( Y \) is the root of the taxonomy \( T \). The node means that every g-itemset in the subtree contains at least one of the g-items \( Y, W, C, A \) or \( B \), which are the g-items from the taxonomy rooted at \( Y \).
Fig. 3. The classification tree corresponding to the taxonomy of Fig. 2(a).

- **MAY-HAVE-PART-OR-ALL-OF** \((S_3)\): A g-itemsets in the sub-tree may, but are not required to, contain g-items in the sub-taxonomies rooted by g-items in \(S_3\). For instance, consider the node \("(W)()(C)\". Every g-itemset in the sub-tree must literally contain \(W\), and may or may not contain \(C\). Hence, the sub-tree has two g-itemsets: \(WC\) and \(W\).

Notice that if at some node \(N\), the \(S_2\) part contains some leaf item \(i\) in \(T\), moving \(i\) to \(S_1\) will create an equivalent node \(N'\). For instance, \((())(B)(C)\) is equivalent to \((B)()(C)\). For this reason, we require that only a non-leaf g-item can appear in the \(S_2\) part of some node in the classification tree.

We have seen the form of a tree node and the content of the root node (the \(S_2\) part is the root of \(T\), while \(S_1\) and \(S_3\) are empty). To define the classification tree, it remains to define the child nodes for an arbitrary node. We know the \(S_2\) part of any tree node is either a single non-leaf g-item, or the empty set. In the discussion below we differentiate these two cases.

**Child Nodes of** \((S_1)(X)(S_3)\) Here \(X\) is a non-leaf g-item in \(T\). Let the children of \(X\) in \(T\) be \(X_1, \ldots, X_k\). The node \(N = (S_1)(X)(S_3)\) has the following \(k + 1\) children in the classification tree ordered as follows:

1. \((S_1)(X_1)(X_2 \ldots X_k S_3)\)
2. \((S_1)(X_2)(X_3 \ldots X_k S_3)\)
3. \((S_1)(X_3)(X_4 \ldots X_k S_3)\)
   :
   k. \((S_1)(X_k)(S_3)\)
   k+1. \((S_1 X)() (S_3)\)

The g-itemsets in \(\text{subtree}(N)\) are classified into \(k+1\) categories. The \((k+1)^{th}\) category consists of the g-itemsets that contain the g-item \(X\) literally. Recall that \(\text{tax}(i)\) denotes the sub-taxonomy rooted by \(i\). For the remaining g-itemsets,
since they must contain some g-item in some \( tax(X_i) \), they can be classified into \( k \) categories. Category 1 contains the g-itemsets that contain some g-item in \( tax(X_1) \). Category 2 consists of the g-itemsets that contain some g-item in \( tax(X_2) \) but does not contain any g-item in \( tax(X_1) \). Category 3 consists of the g-itemsets that contain some g-item in \( tax(X_3) \) but does not contain any g-item in \( tax(X_1) \) or \( tax(X_2) \), and so on.

A special case is when any \( X_i \) is a leaf g-item in \( T \). In this case, the child node \((S_1)(X_i)(X_{i+1} \ldots X_kS_3)\) should be replaced by the equivalent node \((S_1X_i)(X_{i+1} \ldots X_kS_3)\).

**Child Nodes of \((S_1)()S_3\)** Here we differentiate three cases. First, if \( S_3 \) is also empty, this is a leaf node. And no child node is needed.

The second case is when all g-items in \( S_3 \) are leaf g-items in \( T \). The child nodes can be generated in the following way: take each subset of \( S_3 \) and add it to \( S_1 \). For instance, the children of node \((A)(BC)\) are: \((ABC)()\), \((AB)()\), \((AC)()\), and \((A)()\).

The third case is when \( S_3 \) contains some non-leaf g-item in \( T \). Let \( S_3 = \{X\} \cup S'_3 \) where \( X \) is a non-leaf g-item in \( T \). The node \( N = (S_1)()(XS'_3) \) has two children ordered as follows:

1. \((S_1)(X)(S'_3)\): whose sub-tree corresponds to the g-itemsets that contain some g-item in \( tax(X) \); and
2. \((S_1)()S'_3\): whose sub-tree corresponds to the g-itemsets that do not contain any g-item in \( tax(X) \).

### 5.2 The Classification-Tree-based Mining Algorithm MFGI_class

This section outlines the classification-tree-based algorithm to mine max frequent g-itemsets, given a taxonomy \( T \), a transactional database, and \textit{minsupport}. The algorithm dynamically generates the classification tree as defined in Section 5.1, with pruning techniques to be discussed in this section. The order in which nodes are generated will be important for the false-positive elimination discussed in Section 5.3.

We mentioned before that an index node in the classification tree corresponds to a g-itemset, which is a superset of all g-itemsets in the sub-tree. Let’s formally define the concept of \textit{corresponding g-itemset} for an arbitrary tree node.

**Definition 3** The \textbf{corresponding g-itemset} of a node \((S_1)(S_2)S_3\) is a g-itemset that contains every g-item in \( S_1 \) literally, and all leaf g-items in all sub-taxonomies rooted by g-items in \( S_2 \) and \( S_3 \).

For example, the corresponding g-itemset for \((W)()(C)\) is \( WC \), and the corresponding g-itemset for \((())()\) is \( ABC \).

**Theorem 1** The corresponding g-itemset of an index node in the classification tree is the smallest superset of all corresponding g-itemsets in the sub-tree.
Due to space limitations the proofs of all theorems are omitted but can be found in the full version of the paper [7].

Again to mine max frequent g-itemsets, we should perform pruning of (branches) of sub-trees whenever possible.

- **Pruning Technique 1**: If the corresponding g-itemset of a node $N$ is frequent, prune $\text{subtree}(N)$.

As an example, at the root node $(Y)(\emptyset)$ we check the frequency of $ABC$. If $ABC$ is frequent, it is reported as a max frequent g-itemset and the generation of the sub-tree is omitted.

- **Pruning Technique 2**: When generating the child nodes of some index node $(S_1)(X)(S_3)$, we check the frequency of $S_1 \cup \{X_i\}$ for every child g-item $X_i$ of $X$ in $T$. If $S_1 \cup \{X_i\}$ is not frequent, prune $X_i$ before generating the child nodes.

As an example, at node $(Y)(\emptyset)$, we check the frequency of $W$ and $C$. Suppose $W$ is not frequent, we know no g-itemset that contains $W$ or descendants of $W$ in $T$ can be frequent. So to generate the child nodes, we should imagine $W$ does not exist, and $Y$ has a single child $C$ in $T$. Thus only two child nodes should be generated: $(C)(\emptyset)$ and $(Y)(\emptyset)$.

- **Pruning Technique 3**: When generating the child nodes of some index node $(S_1)(\emptyset)(S_3)$, where $S_3$ only contains leaf g-items in $T$, instead of enumerating all subsets of $S_3$, we should use MaxMiner [5] (or other efficient algorithms for mining max frequent itemsets). The reason is that the problem can be transformed into the traditional problem of mining max frequent itemsets without a taxonomy. We basically want to find, among the transactions that support $S_1$, max frequent itemsets when considering the items in $S_3$.

Our algorithm for finding max frequent g-itemsets can be summarized as follows:

**Algorithm MFGI_class**

1. Starting from the root node $(Y)(\emptyset)$, where $Y$ is the root of $T$, dynamically and recursively generate the classification tree, in a depth-first manner.
2. At each node $(S_1)(S_2)(S_3)$, according to Pruning Technique One, we check the frequency of the corresponding g-itemset. If it is frequent, it is identified as a candidate for max frequent g-itemset and there is no need to expand the subtree.
3. If $S_2$ is not empty (but is a single g-item $X$), we apply Pruning Technique Two and then generate the child nodes as defined in Section 5.1.
   (a) Let $\{X_{s_i}\} (i \in [1..k])$ be the set of child g-items of $X$ in $T$, such that $\forall i \in [1..k], S_1 \cup \{X_{s_i}\}$ is frequent.
(b) Generate $k+1$ child nodes in the following order:

$$(S_1)(X_{s_1})(X_{s_2} \ldots X_{s_k} S_3),$$
$$(S_1)(X_{s_2})(X_{s_3} \ldots X_{s_k} S_3),$$
$$\vdots$$
$$(S_1)(X_{s_k})(S_3),$$
$$(S_1 X)()(S_3).$$

Again if some $X_{s_i}$ is a leaf g-item in $T$, add it to the $S_1$ part instead.

4. If $S_2$ is empty, we generate the child nodes as defined in Section 5.1.

(a) If $S_3$ is empty, this is a leaf node. Since the corresponding g-itemset is not frequent (otherwise step 2 of the algorithm would have identified it), nothing needs to be done.

(b) If all g-items in $S_3$ are leaf g-items in $T$, according to Pruning Technique Three, we should plug-in MaxMiner (or similar tools) to process the subtree.

(c) Otherwise, let $S_3$ be $\{X\} \cup S'_3$ where $X$ is a non-leaf g-item in $T$. Generate the two child nodes $(S_1)(X)(S'_3)$ and $(S_1)()(S'_3)$ in the given order.

5.3 Online Elimination for False Positives

The algorithm of Section 5.2 may produce false positives in the sense that it produces a superset of the max frequent g-itemsets. Thus, the g-itemsets produced by the algorithm should be viewed as candidate g-itemsets. Luckily, as we will show, the candidates produced by our algorithm satisfy a superset-before-subset property. That is, if $S_1$ is a superset of $S'_1$ in the taxonomy $T$ ($S_1 \supseteq_T S'_1$), and if both are produced by our algorithm, then the algorithm will generate $S_1$ before $S'_1$.

**Theorem 2** Algorithm MFGI_class can find all max frequent g-itemsets. The generated candidates satisfy the superset-before-subset property.

This has two important benefits. First, the false positives can be identified and eliminated online. Second, in testing a candidate max frequent g-itemset $S'_1$, we need only compare it with the known max frequent g-itemsets that have been generated so far. Therefore, as each candidate max frequent g-itemset is produced, it can be immediately checked and eliminated if it is a false positive.

This method performs $O(nm)$ comparisons of g-itemsets, where $n$ is the number of candidate max g-itemsets and $m$ is the number of true max g-itemsets. The offline method, which would be required without the superset-before-subset property, requires $O(n^2)$ comparisons of g-itemsets. In cases where there are many more false positives than true ones (when $n \gg m$), the online method is much faster.

5.4 PHDB: An Optimization to Batch-Compute Frequencies

So far we have ignored the discussion on how to compute frequencies. The discussion of MFGI_class sort of implied a naive way: go through the transactional
database each time the frequency of some g-itemset is needed. This is obviously inefficient. The reason is that it is typically very expensive to scan through the transactional database. This section introduces an optimization technique called PHDB, which aims to minimize the number of database scans by computing multiple frequencies per scan.

There are two issues that arise. A simple issue is: for each database scan, how many frequencies we should compute? Well, to minimize the number of database scans, we should compute as many as possible. This number is limited by the available memory. Therefore we assume this number is provided by the user who knows the application settings.

The second issue is: given a number num of frequencies to compute for each database scan, which num g-itemsets we should pick? This is a challenging issue. If we are computing one frequency at a time, we are sure that all frequency computation is necessary. But if we compute multiple frequencies at a time, some of them may be “wasteful”. That is, we may compute the frequency for some classification-tree node, which could be pruned if we had computed one frequency at a time. It is challenging to predict which g-itemsets are “useful”.

We address this issue by using the Parameterized Hybrid Depth-first Breadth-first expansion (PHDB). It is a hybrid approach between depth-first and breadth-first expansions, with a parameter controlling the tendency. The depth-first approach uses the following method to choose num g-itemsets to compute frequency. It maintains the current tree: the currently expanded part of the classification tree. At each loop, it temporarily expands the current tree in a depth-first manner, until num g-itemsets are met. It then computes their frequencies and updates the current tree accordingly. Similarly, the breadth-first approach chooses num g-itemsets by temporarily expanding the current tree in a breadth-first manner.

In PHDB, we use a parameter $\sigma \in [-1, 1]$ to control the balance between depth-first and breadth-first behavior. When $\sigma < 0$ the expansion is skewed toward depth-first and when $\sigma > 0$ the expansion is skewed toward breadth-first. Further, when $\sigma = -1$ the expansion is exactly depth-first and when $\sigma = 1$ the expansion is exactly breadth-first. Consider the nodes in the current tree. Let a live node be one that has not had all of its children expanded yet. Let a live level be the set of live nodes with the same depth. PHDB picks num g-itemsets by temporarily expanding the current tree in the following way:

**Algorithm PHDB**
1. Let $L = (l_0, l_2, \ldots, l_m)$ be a list of live levels, sorted in ascending order of depth.
2. Probabilistically choose a level, where the probability to choose level $i$ is:
   \[
   p_i = \begin{cases} 
   (1 - \sigma)^i / \sum_{j=0}^{m} (1 - \sigma)^j, & \text{if } \sigma \geq 0; \\
   (1 + \sigma)^{m-i} / \sum_{j=0}^{m} (1 + \sigma)^j, & \text{if } \sigma < 0. 
   \end{cases} 
   \]
   (1)
3. Expand one child of the left-most live node from the chosen level.
4. If we have met num g-itemsets, stop. Otherwise, goto step 1.
To understand PHDB, let’s study an example of probability distribution of choosing a level among four levels, with different values of $\sigma$. Here $S = 1 + 0.2 + 0.2^2 + 0.2^3$.

<table>
<thead>
<tr>
<th>Level</th>
<th>$\sigma = -1$</th>
<th>$\sigma = -0.8$</th>
<th>$\sigma = 0$</th>
<th>$\sigma = 0.8$</th>
<th>$\sigma = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0</td>
<td>0</td>
<td>$0.2^3/S$</td>
<td>$1/4$</td>
<td>$1/S$</td>
<td>1</td>
</tr>
<tr>
<td>Level 1</td>
<td>0</td>
<td>$0.2^2/S$</td>
<td>$1/4$</td>
<td>$0.2/S$</td>
<td>0</td>
</tr>
<tr>
<td>Level 2</td>
<td>0</td>
<td>$0.2/S$</td>
<td>$1/4$</td>
<td>$0.2^2/S$</td>
<td>0</td>
</tr>
<tr>
<td>Level 3</td>
<td>1</td>
<td>$1/S$</td>
<td>$1/4$</td>
<td>$0.2^3/S$</td>
<td>0</td>
</tr>
</tbody>
</table>

When $\sigma = -1$, PHDB is equivalent to the depth-first approach, since it always picks the deepest level to expand. When $\sigma < 0$, PHDB is skewed towards the depth-first approach, since the probability of picking a deeper level is larger. When $\sigma = 0$, the probability of picking any level is the same. When $\sigma > 0$, PHDB is skewed towards the breadth-first approach. Finally, when $\sigma = 1$, PHDB is equivalent to the breadth-first approach.

6 Experimental Analysis

We cannot find a very appropriate algorithm to compare with our new algorithm MFGI\textsubscript{class}, simply because this is the first work that mines max frequent g-itemsets. Nevertheless, we manage to compare with two not-so-appropriate algorithms, so that people get an idea of the efficiency of MFGI\textsubscript{class}.

As far as we know, the naive lattice-based algorithm MFGI\textsubscript{lattice} is the only other algorithm that can mine the set of max frequent g-itemsets. So the first set of experiments we conduct (Section 6.1) is to compare our algorithm MFGI\textsubscript{class} with MFGI\textsubscript{lattice}.

In the second set of experiments, we compare MFGI\textsubscript{class} with BASIC [15]. Note that BASIC was proposed to find all frequent g-itemsets. So we give MFGI\textsubscript{class} the additional handicap of producing all frequent g-itemsets from the set of identified max frequent g-itemsets. Note that we choose BASIC (and not later algorithms such as SET) for two reasons. First, since our approach is intrinsically much faster, the relative performance difference among the existing algorithms (for mining frequent g-itemset) is very small. Second, BASIC is simple both in concept and in implementation.

Furthermore, to favor the other algorithms, in the first two sets of experiments we do not apply the PHDB optimization in the comparison graphs.

In the third set of experiments, we measure the effect of applying the PHDB optimization and experiment with different choices of the parameter $\sigma$.

The algorithms were implemented in Sun Java 1.4.2, and executed on a Sun Blade 1500 with 1 GB of memory running SunOS 5.9. In the first two sets of experiments, we put the generated databases in memory and measure the total execution time. In the third set of experiments, we measure the number of database scans.

The experimental data were generated with the widely used Quest Synthetic Generator [1]. The specific properties of each of the datasets will be described
in detail in the following sections. Let us explain why we do not use real data. Even though the work is strongly motivated by the existence of real taxonomies, we tried hard but could not locate any good-size real transactional data with real taxonomies. In [18], although the transactional data was real, the taxonomy was synthetically generated.

6.1 MFGI_class versus Naive Lattice-Based Enumeration

We compare MFGI_class with the lattice-based approach MFGI_lattice (Section 4). Since MFGI_lattice takes extremely long to execute, here we only use a very small dataset. The database has 1000 transactions, each of which contains between 2 and 5 randomly chosen items from the leaf-level items in the taxonomy. The taxonomy has \( N \) g-items with constant fanout 2. The specified minimum support is 0.3. Table 2 demonstrates the practical infeasibility of lattice-based methods for mining max frequent g-itemsets.

<table>
<thead>
<tr>
<th>( N )</th>
<th>Lattice</th>
<th>MFGI_class</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>2645</td>
<td>33</td>
</tr>
<tr>
<td>31</td>
<td>( 3.6 \times 10^6 )</td>
<td>33</td>
</tr>
</tbody>
</table>

The two methods are compared over three different metrics: the number of nodes generated in either the lattice or classification tree; the number of g-itemsets for which support must be calculated; and running time.

The infeasibility of the lattice based method is evident in these results. For example, the lattice method required approximately 8 hours of running time for a taxonomy with 31 items and a database with 1000 transactions, whereas MFGI_class required only 1.5 seconds.

6.2 MFGI_class versus BASIC

Although there are many algorithms for mining frequent g-itemsets, we chose to compare with BASIC for its simplicity. The huge speedup over BASIC that we achieve especially for taxonomy levels of 5 and above are far beyond what is achieved by other algorithms for mining frequent g-itemsets.

Since Srikant and Agrawal also presented Cumulate and EstMerge [15] and reported that they are 2 to 5 times faster than BASIC, in the performance graphs we include a band of a factor of 5 in the speed of Basic.

Table 3 presents the default parameters used for experimental data generation.

Fig. 4(a) compares MFGI_class with BASIC as the number of levels of the taxonomy increases, while holding constant the total number of items in the taxonomy. Here \( \text{minsupport} = 0.05 \). In the graph, the “previous best” line was manually generated by taking 1/5 of the execution time of BASIC.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Taxonomy</strong></td>
<td></td>
</tr>
<tr>
<td>Number of items</td>
<td>1000</td>
</tr>
<tr>
<td>Number of levels</td>
<td>5</td>
</tr>
<tr>
<td><strong>Transactional Database</strong></td>
<td></td>
</tr>
<tr>
<td>Number of transactions</td>
<td>10000</td>
</tr>
<tr>
<td>Average size of transaction</td>
<td>5</td>
</tr>
<tr>
<td>Number of patterns</td>
<td>300</td>
</tr>
<tr>
<td>Average length of patterns</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3. Parameters for data set generation.

![Graphs](image)

(a) varying levels  
(b) varying min_support

Fig. 4. Comparison against BASIC.

Clearly, MFGI_class is exponentially faster than BASIC as the number of levels of the taxonomy increases. With a 5-level taxonomy, BASIC took approximately 8 hours to complete. In an extrapolation of the timing, it appears that MFGI_class should be more than 1,000,000 times faster than BASIC for 7 levels.

Fig. 4(b) shows the performance comparison by varying min_support while holding the number of taxonomy levels constant at 5. MFGI_class shows exponential improvement over BASIC with decreasing minimum support. Experiments for BASIC with minimum support lower than 0.05 were infeasible.

### 6.3 The Effect of the PHDB Optimization

We use the same data as described in Table 3. In particular, the taxonomy has depth=5, and minsupport = 0.05.

We first measure the effect of applying the PHDB optimization, as shown in Table 4. Here num is the number of frequencies to compute for each database scan. And Speedup is the ratio between the number of database scans needed when num = 1 and the number of database scans needed for each given num. As expected, typically the speedup is not as big as num because some frequency computations are wasteful. Nevertheless, we do observe an increasing speedup.
as $num$ increases. As $num$ becomes larger, the increase of speedup tends to be slower, since a larger portion of the frequency computations are wasteful.

<table>
<thead>
<tr>
<th>$num$</th>
<th>1 5 10 20 50 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>2.5 4.4 6.2 11.7 12.3</td>
</tr>
</tbody>
</table>

**Table 4.** The speedup on the number of database scans due to PHDB.

Finally, we experiment with the choice of parameter $\sigma$. In Fig. 5(a), when the number of frequencies to check per scan is $num = 100$, the best choice of $\sigma$ is -1. In fact, our extensive experiments revealed that for most practical $num$, the depth-first approach is the best. Only when $num$ is very small can the depth-first approach stop being the best choice. For example, in Fig. 5(b) where $num$ is as small as 7, the best choice of $\sigma$ is -0.8. We recommend to use $\sigma = -1$ (i.e. the depth-first approach) for all practical scenarios.

![Fig. 5. The impact of the parameter $\sigma$ in PHDB.](image)

7 Conclusions & Future Work

This paper solved, for the first time, the problem of mining max frequent $g$-itemsets. Our solution is based on a proposed conceptual classification tree of $g$-itemsets. The algorithm **MFGI_class** was provided to dynamically generate the classification tree. By utilizing three pruning techniques, the algorithm can efficiently generate max frequent $g$-itemsets by pruning the sub-trees that are not needed. The algorithm was carefully designed such that the generated candidates satisfy the superset-before-subset property. This enables on-line elimination of false positives. To reduce the number of database scans, the paper proposed the **PHDB** optimization to batch-compute frequencies. Experimental results showed that to mine max frequent $g$-itemsets, **MFGI_class** is definitely superior than its only opponent, the lattice-based approach. Also, **MFGI_class** can be used to much more efficiently find all frequent $g$-itemsets than existing algorithms. We are currently working on the problem of efficiently mining a compact set of generalized association rules.
References

10. J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In SIGMOD, pages 1–12, 2000.
14. J. Pei, J. Han, and R. Mao. CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. In ACM/SIGMOD Int. Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD), pages 21–30, 2000.