Kunkle D, Zhang DH, Cooperman G. Mining frequent generalized itemsets and generalized association rules without redundancy. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 23(1): 77–102 Jan. 2008

# Mining Frequent Generalized Itemsets and Generalized Association Rules Without Redundancy*

Daniel Kunkle, Donghui Zhang (张冬晖), and Gene Cooperman

*College of Computer and Information Science, Northeastern University, Boston, MA 02115, U.S.A.*

E-mail: {kunkle, donghui, gene}@ccs.neu.edu

Received January 17, 2007; revised December 13, 2007.

**Abstract** This paper presents some new algorithms to efficiently mine max frequent generalized itemsets (g-itemsets) and essential generalized association rules (g-rules). These are compact and general representations for all frequent patterns and all strong association rules in the generalized environment. Our results fill an important gap among algorithms for frequent patterns and association rules by combining two concepts. First, generalized itemsets employ a taxonomy of items, rather than a flat list of items. This produces more natural frequent itemsets and associations such as (*meat*, milk) instead of (beef, milk), (chicken, milk), etc. Second, compact representations of frequent itemsets and strong rules, whose result size is exponentially smaller, can solve a standard dilemma in mining patterns: with small threshold values for support and confidence, the user is overwhelmed by the extraordinary number of identified patterns and associations; but with large threshold values, some interesting patterns and associations fail to be identified.

Our algorithms can also expand those max frequent g-itemsets and essential g-rules into the much larger set of ordinary frequent g-itemsets and strong g-rules. While that expansion is not recommended in most practical cases, we do so in order to present a comparison with existing algorithms that only handle ordinary frequent g-itemsets. In this case, the new algorithm is shown to be thousands, and in some cases millions, of the time faster than previous algorithms. Further, the new algorithm succeeds in analyzing deeper taxonomies, with the depths of seven or more. Experimental results for previous algorithms limited themselves to taxonomies with depth at most three or four.

In each of the two problems, a straightforward lattice-based approach is briefly discussed and then a classification-based algorithm is developed. In particular, the two classification-based algorithms are MFGI_class for mining max frequent g-itemsets and EGR_class for mining essential g-rules. The classification-based algorithms are featured with conceptual classification trees and dynamic generation and pruning algorithms.

**Keywords** generalized association rules, frequent generalized itemsets, redundancy avoidance

## 1 Introduction

Mining generalized itemsets (g-itemset) and generalized association rules (g-rules) are well-motivated existing problems[1−5]. The set of all generalized items (g-items) forms a taxonomy $\mathcal{T}$. For instance, the items "apple" and "strawberry" may have a common parent in $\mathcal{T}$ called "fruit". The transactions in the transactional database are ordinary itemsets drawn from the leaves of $\mathcal{T}$ (such as "apple" or "strawberry"). However, a g-item at higher levels of $\mathcal{T}$ (such as "fruit") may also appear in g-itemsets or g-rules. It could be that (fruit, milk) is frequent, while none of (apple, milk) and (strawberry, milk) is frequent. Meaningful associations can be identified by allowing non-leaf g-items to appear in g-itemsets and g-rules. The problems have many real-life applications besides the standard example of market-basket analysis, as long as the applications have the notion of taxonomy. Here are some examples: diseases have a natural hierarchy, e.g., as given 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 classifies (`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. For example, DMOZ (`http://dmoz.org`) is known as a comprehensive volunteer-edited directory of the Web. Mining frequent g-itemsets and strong g-rules can help identify associations among (categories of) diseases, associations among (groups of) occupations, and so on.

Regular Paper

*A shorter version of this work appeared in CIKM'06 as a two-page poster[32].

However, there is a serious limitation of existing work on this topic. It is well-known that one g-itemset has an exponential number of subsets, and for any frequent g-itemsets, all its subsets are also frequent. Therefore if the user of the data mining engine (based on existing work) uses a slightly smaller *minsupport* threshold, there will be a significant increase in the number of frequent g-itemsets. We observe the following dilemma:

*A small minsupport value will overwhelm the user with many redundant frequent g-itemsets. A large minsupport value will fail to identify some interesting associations.*

The same dilemma also occurs for the case of mining g-rules. The users of the mining engine are typically limited to large *minsupport* and *minconf* threshold values (which will fail to identify interesting associations), or are presented with many redundant results.

The solution to this dilemma is to mine a *compact representation* of all frequent g-itemsets and strong association rules. In particular, in this paper we address the problem of mining *max frequent g-itemsets* and *essential g-rules.*

The novelty of this paper is three-fold:

1) By removing redundant results, we allow the end user to lower the *minsupport* and *minconf* thresholds, and examine additional interesting patterns.

2) Even for a fixed threshold, our algorithm can produce the set of frequent itemsets (a non-compact representation) many times faster. This is because, from the set of max frequent g-itemsets (essential g-rules), one can derive the set of all frequent g-itemsets (strong g-rules) without examining the transactional database.

3) Furthermore, we are able to analyze taxonomies of depth seven and more, while previous experimental results present examples for at most three or four. That limitation makes the earlier algorithms impractical for such natural data sets as DMOZ, with a taxonomy of depth nine and more.

### 1.1 How Are Closed Frequent g-Itemsets and Max Frequent g-Itemsets Complementary?

There exists an alternative compact representation of frequent g-itemsets: the *closed frequent g-itemsets*[6]. The representation of closed frequent g-itemsets has the following goal: from this compact representation, one should be able to derive not only the set of all frequent g-itemsets, but also to rapidly retrieve the exact support of each frequent g-itemset. To this end, a frequent g-itemset returned by a closed frequent algorithm may additionally contain information on some subsets of the frequent g-itemset, whose support values are larger than that of the superset. The compactness of this alternative sits somewhere between max frequent g-itemsets and the complete set of all frequent g-itemsets.

On the other hand, the representation of max frequent g-itemsets is more compact, because it contains absolutely no proper subset of a frequent g-itemset, or the additional support information. A max frequent g-itemset algorithm is not delayed by the need to report additional support information. Hence, it can both be faster than closed frequent algorithms, and more compact. This is important if the user wishes to present multiple queries for different values of *minsupport.*

### 1.2 Can Existing Solutions Be Adapted to Solve This New Problem?

We propose our new algorithms because no existing solution can be adapted to efficiently mine max frequent g-itemsets or essential g-rules.

For mining max frequent g-itemsets, let us consider three possible straightforward solutions as follows.

First, one may use existing solutions to mine all frequent g-itemsets, and then eliminate the non-max ones. But this solution is extremely inefficient and, in practical cases, infeasible. The point of introducing max frequent g-itemsets is to avoid enumerating all frequent g-itemsets. Meaningful algorithms should *directly* find the set of max frequent g-itemsets.

Second, there exist solutions to mine max frequent itemsets in the ordinary case. Unfortunately, the introduction of g-items brings significant difficulty, and thus the solutions to the ordinary cases do not apply. As we will see in Section 3, even the set operators of $\in$ and $\subseteq$ have changed their meanings in the generalized environment.

A third choice is to dynamically browse the lattice of all g-itemsets, where ancestor nodes are supersets of descendant nodes. If we browse the lattice in a top-down fashion, whenever we see a frequent g-itemset we can prune the search of all its descendants. The algorithm is discussed in Subsection 5.1. However, both our theoretical analysis and experimental result show that this is an inefficient approach.

Similarly, for mining essential g-rules, the approach of mining all strong g-rules and then removing redundancies is meaningless, and the lattice-based solution (Subsection 6.1) is inefficient.

### 1.3 Outline of Results

The novelty of this paper is that it efficiently solves, for the first time, the problem of mining max frequent g-itemsets and essential g-rules.

Section 3 formally defines the problems of mining max frequent generalized itemsets and essential g-rules.

Section 4 provides closed form upper and lower bound on the total number of g-itemsets and g-rules, for the case of $\mathcal{T}$ having constant fanout. For example, consider a height=3 complete binary taxonomy, where there are 15 g-items including 8 leaf-level g-items. Our closed form bound says that there are at most 1525 g-itemsets, while a naive estimation says that there are at most $2^{15} - 1 = 32\,767$ g-itemsets. These results add insight into the scope of generalized itemset mining problems.

Section 5 proposes Algorithm MFGI_class, which efficiently mines max frequent g-itemsets based on a novel classification tree. This is reminiscent of the MaxMiner tree[7] that finds max frequent itemsets in the ordinary case. However, due to the complexity of the generalized environment, our classification tree for g-itemsets (Subsection 5.2) is quite different (and more complex than the MaxMiner tree). Note that the tree is conceptual, and to mine all max frequent g-itemsets our proposed algorithm MFGI_class (Subsection 5.3) only generates a small part of the tree dynamically, employing three pruning techniques. As part of the solution, a method to remove false positives in an online fashion is provided (Subsection 5.4). Also, 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 (Subsection 5.5), which aims to reduce the number of database scans by batch-computing frequencies. Further, the size of the database that needs to be scanned is substantially reduced by a transaction filtering optimization (Subsection 5.6).

Section 6 presents Algorithm EGR_class, which efficiently mines essential g-rules based on a classification tree of g-rules. The classification tree of g-rules is constructed by extending the classification tree of g-itemsets. At each tree node, we compute two confidence values: *MINCONF* and *MAXCONF*. They are a lower bound and an upper bound on the confidence of all g-rules in the subtree. Therefore, if $MAXCONF < minconf$, the whole tree can be pruned. If $MINCONF \geqslant minconf$, we provide an algorithm, named MINCONFProcessing, that quickly generates the essential g-rules in the subtree, without generating the sub-tree or checking any frequency information from the transactional database. Besides the MAXCONF Pruning and the MINCONF Pruning, the algorithm contains yet another pruning technique called *Implication Pruning*, which enables us to prune subtrees using already identified essential g-rules.

Section 7 provides an experimental analysis of our two algorithms, MFGI_class and EGR_class. Our primary experimental comparison is against the algorithm BASIC[3]. We choose BASIC as the baseline algorithm for comparison, following the tradition of previous authors[3,7–13]. BASIC has been widely used as a baseline algorithm because it has a clear, standard implementation whose speed will not be greatly biased by the implementation. This is not true of many of the other algorithms in the literature.

We show that these new algorithms are significantly faster than BASIC, and related algorithms, especially with deeper taxonomies. For example, the results in Fig.11 demonstrate up to a 1 000 000-fold speedup for a taxonomy of depth seven.

## 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 (Subsection 2.1). The table entry (max frequent itemsets, ordinary) lists work on mining max frequent itemsets, as described in Subsection 2.2. Also in Subsection 2.2, we review other compact representations of frequent itemsets, e.g., the *closed frequent itemsets*. In the generalized case, there exists work on mining frequent g-itemsets and closed frequent g-itemsets, as discussed in Subsection 2.3. Table 1 highlights that there is a blank entry with no existing work: to mine max frequent g-itemsets. This paper fills that blank.

**Table 1.** Structured View of Selected Results on Mining Frequent Itemsets

| | Ordinary | Generalized |
|---|---|---|
| Frequent Itemsets | [8, 9, 14], etc. | [1–5] |
| Closed Frequent Itemsets | [11, 12] | [6] |
| Max Frequent Itemsets | [7, 10, 15–17] | This Paper |

### 2.1 Earlier Work on Mining Frequent Itemsets

The concept of mining frequent itemsets was first introduced by Agrawal *et al.*[14] Earlier work on mining frequent itemsets focused on the Apriori algorithm and its improved versions. Recently, Han *et al.*[9] proposed the FP-tree technique, which finds frequent itemsets without generating candidates. Agarwal *et al.*[8] 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 Compact Representations of Frequent Itemsets

Mining max frequent itemsets was introduced by Bayardo[7], where the MaxMiner algorithm was presented. The Pincer algorithm was proposed by Lin and Kedem[10]. The MAFIA algorithm was proposed by Burdick *et al.*[16], which finds supersets of max frequent itemsets. Gouda and Zaki[17] 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.*[15] 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.

Another compact representation of all frequent itemsets was that of *closed frequent itemsets*, introduced by Pasquier *et al.*[11] Later, another solution to the closed itemset problem was proposed by Pei *et al.*[12]

Recently, [18] and [19] described another type of compact representation of frequent itemsets, one focusing on an exact solution and the other focusing on an approximate solution. There also exists the *non-derivable patterns*[20], the *quantitative correlated patterns*[21], and the *hyperclique patterns*[22].

Also related is the best paper of VLDB'05[23], which addresses the problem of improving the performance of several frequent-itemset-mining algorithms like FP-Growth, by making them cache-conscious.

### 2.3 Mining Generalized Itemsets

The problem of mining generalized itemsets was first introduced by Srikant and Agrawal[3]. They proposed three algorithms for frequent generalized itemsets: Basic, Cumulate and EstMerge. Later, Hipp *et al.*[1] 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[4,5] proposed the SET algorithm to mine frequent g-itemsets. Pramudiono and Kitsuregawa[2] proposed the FP-tax method, which extends the top-down FP-growth method[13] to the generalized environment.

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[6].

Han and Fu[24] proposed the *multiple-level itemsets*. Similar to g-itemsets, a taxonomy $\mathcal{T}$ is involved. But, a multi-level itemset is restricted to only contain items from the same level of $\mathcal{T}$.

### 2.4 (Ordinary) Association Rules Mining

Table 2 provides a structured view of selected works on various association-rule mining problems.

**Table 2.** Selected Work on Mining Association Rules

| | Ordinary | Generalized |
|---|---|---|
| Strong Rules | [14], etc. | [1, 3–5, 25] |
| Essential Rules | [26, 27] | This Paper |

The concept of mining strong rules was first introduced by Agrawal *et al.*[14] The proposed solution is a two-step approach. First, all frequent itemsets are identified. Then, for each frequent itemset, generate the strong rules whose antecedent and consequent are subsets of the itemset. Earlier work focused on finding frequent itemsets. There was an abundance of follow-on work on this and related topics. For the purpose of this paper we only focus on the closely-related ones. It is worth mentioning that in the data mining research community there seems to be a re-emerged interest in this topic[18,19,23].

As pointed out by Aggarwal and Yu[26], a strong rule with $k$ items in the consequent implies $3^k - 2^k - 1$ other strong rules. That is, a strong rule implies an exponential number of other strong rules. Thus Aggarwal and Yu pointed out the interesting task to mine *essential rules*: strong rules not implied by any other strong rule. The set of essential rules is a compact representation of the set of all strong rules.

Another compact representation of all strong rules is called the *non-redundant rules*[27]. This compact representation can be used not only to derive the set of all strong rules, but also to tell the support and confidence values of each derived rule. On the other hand, the set of essential rules typically is much smaller than the set of non-redundant rules. If one wants to quickly generate a compact representation of all strong rules without caring about the support and confidence of each individual rule, the essential rule is a better choice. In this paper we focus on extending the concept of essential rules to the generalized case.

### 2.5 Generalized Association Rule Mining

Most existing work on mining generalized itemsets also solved the problem of mining generalized association rules[1,3−5].

Huang and Wu[25] proposed the GMAR algorithm to find generalized rules (between the items at different levels in the taxonomy tree) under the assumption that the original frequent itemsets and rules have already been generated beforehand. It was reported that the GMAR algorithm is much better than BASIC and Cumulate algorithms.

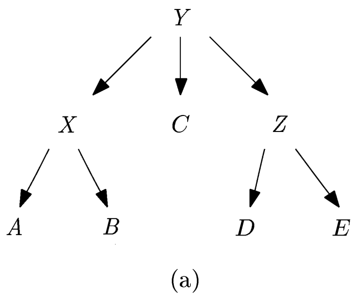
Methods to mine generalized rules with multiple *minsupport* threshold values were proposed by [28, 29].

## 3 Problem Definition

This section first reviews the problem of mining frequent g-itemsets[3], then defines the new problems of mining max frequent g-itemsets and mining essential g-rules.

### 3.1 Generalized Itemsets

The set of all items form a taxonomy $\mathcal{T}$, which is a tree structure. An example is shown in Fig.1(a). The leaf-level g-items $A$, $B$, $C$, $D$, and $E$ are regular items that may appear in the transactional database $\mathcal{D}$ (Fig.1(b)). A transaction that contains a g-item is also considered to "contain" all its ancestor g-items in $\mathcal{T}$. For instance, anybody who bought $A$ (apple) or $B$ (banana) is considered to have also bought $X$ (fruit).



| | Trans. |
|---|---|
| 1 | $AC$ |
| 2 | $AC$ |
| 3 | $ABC$ |
| 4 | $C$ |
| 5 | $D$ |
| 6 | $E$ |

(b)

Fig.1. Taxonomy and database. (a) Taxonomy $\mathcal{T}$. (b) Trans. database $\mathcal{D}$.

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

Hence, $ACZ$ in Fig.1(a) is a g-itemset, but $AXZ$ is not. Intuitively, since anyone who bought apple is also considered to have bought fruit, the set {apple, fruit} is not compact, and so is not considered a valid g-itemset. (The equivalent compact itemset is {apple}.) We need every itemset to be in its most compact representation.

### 3.2 Some Operators of Itemsets Redefined with Regard to $\mathcal{T}$

Given a g-item $i \in \mathcal{T}$ and a g-itemset $S$, we say $i$ belongs to $S$ with respect to $\mathcal{T}$, denoted as $i \in_{\mathcal{T}} S$, if $\exists j \in S$ such that $i = j$ or $i$ is an ancestor of $j$ in $\mathcal{T}$. For example, in Fig.1(a), $X \in_{\mathcal{T}} \{AC\}$ as $X$ is an ancestor of $A$ in $\mathcal{T}$. Intuitively, $A \in_{\mathcal{T}} \{AC\}$ since any one who bought $A$ and $C$ is considered to have bought $A$. In a similar vein, anybody who bought apple ($A$) and something else is considered to have bought fruit ($X$). So $X \in_{\mathcal{T}} \{AC\}$.

Given two g-itemsets $S_1$ and $S_2$, we say $S_1$ is a subset of $S_2$ with respect to $\mathcal{T}$, denoted as $S_1 \subseteq_{\mathcal{T}} S_2$, if $\forall i \in S_1$, $i \in_{\mathcal{T}} S_2$. For example, in Fig.1(a), $\{XC\} \subseteq_{\mathcal{T}} \{ACD\}$. Among all g-itemsets in this example taxonomy, the largest one (i.e., superset of everything) is $\{ABCDE\}$, and the smallest one is $\{Y\}$. That is, for any g-itemset $S$, we have: $\{Y\} \subseteq_{\mathcal{T}} S \subseteq_{\mathcal{T}} \{ABCDE\}$. We also have the proper subset notation ($\subset_{\mathcal{T}}$) with its obvious meaning.

The union and intersection operators with respect to $\mathcal{T}$ are also defined for two g-itemsets $S_1$ and $S_2$. $S_1 \cup_{\mathcal{T}} S_2$ is the smallest g-itemset that is a superset of both $S_1$ and $S_2$ with respect to $\mathcal{T}$. $S_1 \cap_{\mathcal{T}} S_2$ is the largest g-itemset that is a subset of both $S_1$ and $S_2$ with respect to $\mathcal{T}$. For instance, in Fig.1, $\{XC\} \cup_{\mathcal{T}} \{BZ\} = \{BCZ\}$, $\{XC\} \cap_{\mathcal{T}} \{BZ\} = \{X\}$.

### 3.3 Max Frequent Generalized Itemsets

The *support* of a g-itemset $S$ is the percentage of transactions in $\mathcal{D}$ that are supersets of $S$ with respect to $\mathcal{T}$. For instance, in Fig.1, the support of $\{Z\}$ is 1/3. The reason is that among the six transactions, two of them contain either $D$ or $E$, and thus are supersets of $\{Z\}$ with respect to $\mathcal{T}$. As in the ordinary case, a superset has a smaller or equal support (since all transactions that contain the superset also contain its subsets). As an example, $\{D\}$ is a superset of $\{Z\}$, and thus the support of $\{D\}$ (= 1/6) is smaller than the support of $\{Z\}$.

**Definition 2.** *Given minsupport, a* **frequent g-itemset** *is a g-itemset whose support is at least* minsupport. *A* **max frequent g-itemset** *is a frequent* *g-itemset whose proper super g-itemsets are all non-frequent.*

In the example of Fig.1, if *minsupport* $= 1/3$, there are two max frequent g-itemsets: $\{AC\}$ and $\{Z\}$.

### 3.4 Essential Generalized Association Rules

For ease of presentation, later on we omit the parenthesis around g-itemsets. For instance, we would use $A \rightarrow BC$ instead of $\{A\} \rightarrow \{BC\}$.

**Definition 3.** *A generalized rule, or g-rule in short, has the form* $S_1 \rightarrow S_2$, *where* $S_1$ *and* $S_2$ *are g-itemsets and* $\nexists i \in S_2$ *such that* $i \in_{\mathcal{T}} S_1$.

That is, an item in a **valid** g-rule's consequent shall not be equal to or be an ancestor (in $\mathcal{T}$) of any item in the antecedent. Since apple is a fruit, if apple is in the antecedent of a rule while fruit is in the consequent, the rule is trivial and thus is considered to be *invalid*. Also, a valid g-rule should have a non-empty antecedent and a non-empty consequent.

The support of a g-rule $S_1 \rightarrow S_2$ is the percentage of transactions in $\mathcal{D}$ that are a superset of both $S_1$ and $S_2$ with respect to $\mathcal{T}$. The confidence of the g-rule is, among the transactions that are supersets of $S_1$ with regard to $\mathcal{T}$, the percentage of transactions that are a superset of $S_2$ with regard to $\mathcal{T}$. A g-rule is *strong* if its support and confidence are above or equal to the given threshold values *minsupport* and *minconf*, respectively.

**Definition 4.** *A g-rule* $r_1$ **implies** *another g-rule* $r_2$, *if support*$(r_1) \leqslant$ *support*$(r_2)$ *and confidence*$(r_1) \leqslant$ *confidence*$(r_2)$, *independent of the transactional database used. We denote this as* $r_1 \Longrightarrow r_2$.

For example, the rule $A \rightarrow BC$ implies $AB \rightarrow C$. This holds because *support*$(A \rightarrow BC)$ = *support*$(AB \rightarrow C)$, and *confidence*$(A \rightarrow BC)$ = *support*$(ABC)$/*support*$(A) \leqslant$ *support*$(ABC)$/*support*$(AB)$ = *confidence*$(AB \rightarrow C)$. Note that this holds for all transactional databases. If it happens that for some particular transactional database, *support*$(A \rightarrow B) \leqslant$ *support*$(C \rightarrow D)$ and *confidence*$(A \rightarrow B) \leqslant$ *confidence*$(C \rightarrow D)$, we cannot say the first rule implies the second one, for the inequalities may not hold for other transactional databases.

In case $r_1 \Longrightarrow r_2$, if $r_1$ is a strong g-rule, $r_2$ must also be a strong g-rule. The *implication* relationship satisfies *transitivity*: if $r_1 \Longrightarrow r_2$ and $r_2 \Longrightarrow r_3$, then $r_1 \Longrightarrow r_3$.

**Theorem 1.** $S_1 \rightarrow S_2 \Longrightarrow S_1' \rightarrow S_2'$, *if and only if* $S_1 \subseteq_T S_1'$, $S_1 \cup_T S_2 \supseteq_T S_1' \cup_T S_2'$ *and the two rules are different.*

*Proof.* Intuitively, if a g-rule implies another one, the first g-rule's support $\leqslant$ the second g-rule's support for an arbitrary transactional database. This suggests that $S_1 \cup_T S_2$ is a super set of $S_1' \cup_T S_2'$ with respect to $\mathcal{T}$. Furthermore, the first g-rule's confidence $\leqslant$ the second g-rule's confidence. Or,

$$\frac{support(S_1 \cup_T S_2)}{support(S_1)} \leqslant \frac{support(S_1' \cup_T S_2')}{support(S_1')}. \qquad (1)$$

In order for (1) to be true for all transactional databases, we should have: *support*$(S_1) \geqslant$ *support*$(S_1')$, or $S_1$ is a subset of $S_1'$ with respect to $\mathcal{T}$. Finally, as a convention, a g-rule does not imply itself. □

It follows from Theorem 1 that if $S_1 \rightarrow S_2 \Longrightarrow S_1' \rightarrow S_2'$, then $S_2 \supseteq_T S_2'$.

**Definition 5.** *A g-rule* $r_1$ *is an* **essential g-rule** *if* $r_1$ *is a strong g-rule and there does not exist a strong g-rule* $r_2$ *such that* $r_2 \Longrightarrow r_1$.

This paper addresses the problems of efficiently mining max frequent g-itemsets and essential g-rules.

## 4 Counting the Number of g-Itemsets and g-Rules

### 4.1 Bounds on the Number of g-Itemsets

Consider a taxonomy $\mathcal{T}$ of $N$ g-items, out of which $n$ g-items are leaves. An interesting question is: how many g-itemsets are there? A lower bound is $2^n - 1$. For this is the number of itemsets that only involve leaf-level g-items. A coarse upper bound is $2^N - 1$. For this is the number of non-empty sets composed of g-items in $\mathcal{T}$, not excluding the case where in the same set there exist both an ancestor and a descendant. To better understand the insights of the generalized rule mining problem, it is desirable to get a better estimate.

In this section, we consider a *regular taxonomy*, which is a complete, balanced tree with depth (or height) $d$ and branching factor $b$. In this case, $n = b^d$, and $N = 1 + b + b^2 + \cdots + b^d = b^d + \frac{b^d - 1}{b-1} \leqslant b^d\big(1 + \frac{1}{b-1}\big)$. So the straightforward estimation of the number of g-itemsets is:

$$2^{(b^d)} \leqslant f(d) \leqslant (2 \cdot 2^{\frac{1}{b-1}})^{(b^d)}. \qquad (2)$$

Here $f(d)$ is the number of g-itemsets plus one (for ease of presentation we temporarily include the empty set in our calculation).

For this regular taxonomy (constant fanout), we provide a recursive formula that is the exact number of valid g-itemsets. From this formula, we also derive a closer upper bound for $f(d)$.

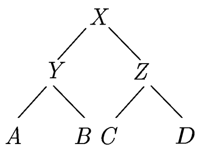
**Theorem 2.**

$$f(0)=2,\quad f(d)=1+f(d-1)^b,$$
$$2^{(b^d)} \leqslant f(d) \leqslant \Big(2\Big(1+\frac{1}{2^b}\Big)^{\frac{1}{b-1}}\Big)^{(b^d)}.$$

*Proof.* For a g-item $X$, let $\Phi(X)$ be the set of g-itemsets (plus the empty set) where g-items are in the sub-taxonomy rooted by $X$. We have: $\Phi(X) = \{X, \Phi(X_1)\dots\Phi(X_b)\}$, where $X_1$ through $X_d$ are children of $X$ in the taxonomy. Here we use the multiplication $\Phi(Y)\Phi(Z)$ to denote

$$\Phi(Y)\Phi(Z) = \{yz : y \in \Phi(Y),\ z \in \Phi(Z)\}.$$

For example, consider the following taxonomy of generalized items with $b = 2$ and $d = 2$.



If $y = B$ and $z = CD$, then $yz = BCD$.
Hence,

$$f(d) = 1 + f(d-1)^b \tag{3}$$

for $d > 0$. And $f(0) = 2$, since $\Phi(A) = \{A, \emptyset\}$ for a leaf g-item $A$.

We next show how to replace the expression $2^{\frac{1}{b-1}}$ in the upper bound of (2) with the strictly smaller expression $\big(1+\frac{1}{2^b}\big)^{\frac{1}{b-1}}$. For example, if $b = 2$, while the straightforward upper bound is $4^{b^d}$, we get $2.5^{b^d}$.

Note that

$$\frac{f(d)}{2^{b^d}} = \frac{(f(1))^{b^{d-1}}}{2^{b^d}}\,\frac{(f(2))^{b^{d-2}}}{(f(1))^{b^{d-1}}}\,\frac{(f(3))^{b^{d-3}}}{(f(2))^{b^{d-2}}}\cdots \frac{(f(d))}{(f(d-1))^b}.$$

Note that for the $i$-th factor in the right hand side above, and using $b \geqslant 2$,

$$\begin{aligned}\frac{(f(i))^{b^{d-i}}}{(f(i-1))^{b^{d+1-i}}} &= \frac{(1+f(i-1)^b)^{b^{d-i}}}{((f(i-1))^b)^{b^{d-i}}}\\ &= \Big(1+\frac{1}{(f(i-1))^b}\Big)^{b^{d-i}}\\ &\leqslant \Big(1+\frac{1}{2^b}\Big)^{b^{d-i}} = \Big((1+\frac{1}{2^b})^{\frac{1}{b^i}}\Big)^{b^d}.\end{aligned}$$

Applying this inequality with $\varphi = 1 + \frac{1}{2^b}$, we have

$$\begin{aligned}\frac{f(d)}{2^{b^d}} &\leqslant (\varphi^{\frac{1}{b^1}}\varphi^{\frac{1}{b^2}}\cdots\varphi^{\frac{1}{b^d}})^{b^d}\\ &= (\varphi^{\frac{1}{b-1}})^{\frac{b}{d}} \leqslant \Big(\Big(1+\frac{1}{2^b}\Big)^{\frac{1}{b-1}}\Big)^{\frac{b}{d}}.\end{aligned}$$

Hence,

$$2^{(b^d)} \leqslant f(d) \leqslant \Big(2\Big(1+\frac{1}{2^b}\Big)^{\frac{1}{b-1}}\Big)^{(b^d)}. \tag{4}$$

□

In summary, for a taxonomy of height $d$ and branching factor $b$, we have derived a recursive formula (3) to calculate the number of g-itemsets (plus 1). While straightforward estimation gives a coarse bound (2) for this number, we derived a tighter bound (4). For example, when $b = 2$, the straightforward upper bound is $4^{b^d}$, and our bound is $2.5^{b^d}$. With $b = 2$ and $d = 3$ (a complete taxonomy where there are eight leaf g-items and where every non-leaf g-item has two children), the real number of g-itemsets (plus 1) as given by (3) is 677. But the straightforward estimation as given by (2) is 65 536 which is about 100 times more. The refined estimation as given in (4) is 1526 which is much better than the straightforward estimation.

## 4.2 Bounds on the Number of g-Rules

Let $g(d)$ be the number of possible g-rules for a balanced tree with branching factor $b$ and depth $d$. Then,

$$f(d) \leqslant g(d) \leqslant f(d)^2$$

where $f(d)$ is the number of possible g-itemsets.

To see this, note that a g-rule is defined by two g-itemsets, an antecedent and a consequent, each of which has $f(d)$ possible forms.

By utilizing the bounds for $f(d)$ derived in the previous section, we see that

$$\begin{aligned}2^{(b^d)} &\leqslant f(d) \leqslant g(d) \leqslant f(d)^2\\ &\leqslant \Big(4\cdot\Big(1+\frac{1}{2^b}\Big)^{\frac{2}{b-1}}\Big)^{(b^d)}.\end{aligned} \tag{5}$$

## 5 Mining Max Frequent g-Itemsets

This section briefly discusses the straightforward lattice-based solution (Subsection 5.1), and then focus on the classification-tree based approach. The classification-based solution has five components. Subsection 5.2 defines a conceptual classification tree. Subsection 5.3 describes the algorithm **MFGI_class** which dynamically generates the needed part of the tree, while using three pruning techniques. Since this algorithm produces a superset of the max frequent g-itemsets, which includes false positives, Subsection 5.4

describes how to efficiently eliminate those false positives in an online fashion. Subsection 5.5 presents the **PHDB** technique, which allows multiple frequencies to be computed with each database scan. Finally, Subsection 5.6 presents a database filtering optimization that significantly reduces the size of the database that must be scanned.

### 5.1 Lattice-Based Solution

A straightforward solution to mining 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 $\nexists S_3$ 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 has a single root, which is the g-itemset composed of all leaf-level g-items in $\mathcal{T}$. It is a superset of all g-itemsets with regards to $\mathcal{T}$.

The lattice-based algorithm that mines max frequent g-itemsets, which we call **MFGI_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 max frequent g-itemset and we do not browse its children (or descendants).

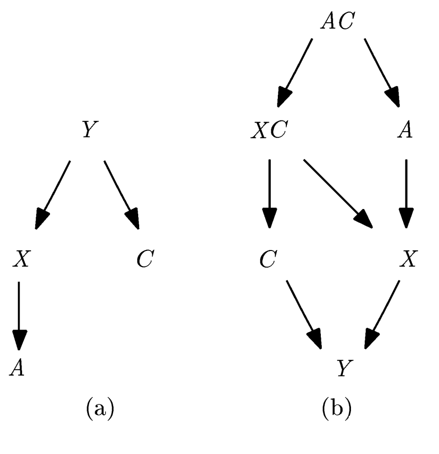


Fig.2. Sub-taxonomy and its corresponding lattice of g-itemsets. (a) Sub-taxonomy. (b) Lattice of g-itemsets.

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 $\mathcal{T}$. 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 $\mathcal{T}$ of some other g-item in the set, the item $i$ should be removed.

For example, in the lattice of Fig.2(b), one child of $AC$ is $XC$, by generalizing $A$ to $X$. Another child of $AC$ can be generated by generalizing $C$ to $Y$. Since $Y$ and $A$ have an ancestor-descendant relationship in $\mathcal{T}$, the generated child is not $AY$, but $A$.

In most practical cases, the use of **MFGI_lattice** is infeasible. 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. The rest of this section presents a more efficient algorithm.

### 5.2 Conceptual Classification Tree

This subsection 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 sub-

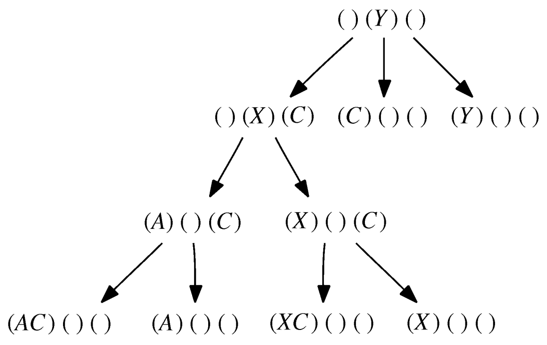


Fig.3. Example classification tree of g-itemsets.

section we focus on defining the complete classification tree, without considering pruning. The complete classification tree, corresponding to the taxonomy in

Fig.2(a), is shown in Fig.3.

### *5.2.1 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 *subtree*$(N)$ must contain every g-item in $S_1$. For instance, in Fig.3, $(XC)()()$ is a leaf node in the classification tree, which corresponds to a g-itemset $XC$.

• *MUST-HAVE-PART-OR-ALL-OF* $(S_2)$: the $S_2$ part consists of zero or one g-item. If it contains a g-item $i$, some g-item in $tax(i)$ must appear in each g-itemset in *subtree*$(N)$. Here, $tax(i)$ 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 $\mathcal{T}$. The node means that every g-itemset in the subtree contains at least one of the g-items in $\mathcal{T}$.

• *MAY-HAVE-PART-OR-ALL-OF* $(S_3)$: every g-itemset in *subtree*$(N)$ may, but is not required to, contain g-items in the sub-taxonomies rooted by g-items in $S_3$. For instance, consider the node "$(X)()(C)$". Every g-itemset in the subtree must literally contain $X$, and may or may not contain $C$. Hence, the sub-tree has two g-itemsets: $XC$ and $X$.

Notice that if at some node $N$, the $S_2$ part contains some leaf item $i$ in $\mathcal{T}$, moving $i$ to $S_1$ will create an equivalent node $N'$. For instance, $()(A)(C)$ is equivalent to $(A)()(C)$. 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.

To define the classification tree, it remains to discuss how to generate the child nodes for an arbitrary node. If this is done, the whole classification tree can be produced by keep generating the child nodes of all nodes, starting from the root node $()(Y)()$. A further requirement is that the $S_2$ part of any tree node is either a single non-leaf g-item, or the empty set. For instance, the root node of the classification tree satisfies this requirement. And it will be clear that the way we generate the child nodes ensures this. In the discussion below we differentiate these two cases.

### *5.2.2 Child Nodes of* $(S_1)(X)(S_3)$

Here $X$ is a non-leaf g-item in $\mathcal{T}$. Let the children of $X$ in $\mathcal{T}$ 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:

$$\begin{array}{rl} 1. & (S_1)(X_1)(X_2 \ldots X_k S_3) \\ 2. & (S_1)(X_2)(X_3 \ldots X_k S_3) \\ 3. & (S_1)(X_3)(X_4 \ldots X_k S_3) \\ & \vdots \\ k. & (S_1)(X_k)(S_3) \\ k+1. & (S_1 X)()(S_3) \end{array}$$

The g-itemsets in *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 $tax(i)$ 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 $\mathcal{T}$. In this case, the child node $(S_1)(X_i)(X_{i+1} \ldots X_k S_3)$ should be replaced by the equivalent node

$$(S_1 X_i)()(X_{i+1} \ldots X_k S_3).$$

### *5.2.3 Child Nodes of* $(S_1)()(S_3)$

Here we differentiate three cases. First, if $S_3$ is also empty this is a leaf node: no child node is needed.

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

The third case is when $S_3$ contains some non-leaf g-item in $\mathcal{T}$. Let $S_3 = \{X\} \cup S'_3$ where $X$ is a non-leaf g-item in $\mathcal{T}$. 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 $tax(X)$.

## 5.3 Classification-Tree-Based Mining Algorithm MFGI_Class

This subsection outlines the classification-tree-based algorithm to mine max frequent g-itemsets, given a taxonomy $\mathcal{T}$, a transactional database, and *minsupport*. The algorithm dynamically generates the classification tree as defined in Subsection 5.2, 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 Subsection 5.4.

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 us formally define the concept of *corresponding g-itemset* for an arbitrary tree node.

**Definition 6.** *The* **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, given the taxonomy of Fig.2(a), the corresponding g-itemset for $(X)()(C)$ is $XC$, and the corresponding g-itemset for $()(Y)()$ is $AC$.

**Theorem 3.** *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.*

*Proof.* Let $C$ be the corresponding g-itemset for a classification tree node $N$. We first prove that $C$ is a superset of all g-itemsets in the sub-tree of $N$. Consider an arbitrary g-itemset $S$ in the sub-tree of $N$. By definition of a classification tree node, the g-items in $S$ consists of $N.S_1$ literally, and may contain some g-items in the sub-taxonomies of $N.S_2$ and $N.S_3$. Since $C$ contains $N.S_1$ literally, and contains all leaf g-items in all sub-taxonomies of $N.S_2$ and $N.S_3$, every g-item in $S$ belongs to some g-item in $C$ with regards to the taxonomy $\mathcal{T}$. Therefore $C$ is a superset of all g-items in the sub-tree of $N$. It remains to point out that there exists a node in the sub-tree of $N$ whose corresponding g-itemset is $C$, according to the definitions of a classification tree node and the corresponding g-itemset for a tree node. $\square$

To mine max frequent g-itemsets, an efficient solution should perform pruning of sub-trees whenever possible, instead of generating the complete classification tree.

• *Pruning Technique* 1: if the corresponding g-itemset of a node $N$ is frequent, prune *subtree*$(N)$.

As an example, at the root node $()(Y)()$ we check the frequency of $AC$. If $AC$ 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 $\mathcal{T}$. If $S_1 \cup \{X_i\}$ is not frequent, prune $X_i$ before generating the child nodes.

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

• *Pruning Technique* 3: when generating the child nodes of some index node $(S_1)()(S_3)$, where $S_3$ only contains leaf g-items in $\mathcal{T}$, instead of enumerating all subsets of $S_3$, we should use MaxMiner[7] (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 $S_3$.

```
Algorithm. MFGI_class
Input: A transactional database $\mathcal{D}$, and a taxonomy $\mathcal{T}$
Action: Generate all max frequent g-itemsets.
1. Starting from the root node $()(Y)()$, where $Y$ is the root
   of $\mathcal{T}$, dynamically and recursively generate the classifi-
   cation tree, in a depth-first manner.
2. At each node $(S_1)(S_2)(S_3)$, according to Pruning Tech-
   nique One, we check the frequency of the corresponding
   g-itemset. If it is frequent, it is identified as a 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 ap-
   ply Pruning Technique Two and then generate the child
   nodes as defined in Subsection 5.2.2.
   (a) Let $\{X_{s_i}\}$ $(i \in [1..k])$ be the set of child g-items
       of $X$ in $\mathcal{T}$, 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:
       $(S_1)(X_{s_1})(X_{s_2} \dots X_{s_k} S_3)$,
       $(S_1)(X_{s_2})(X_{s_3} \dots X_{s_k} S_3)$,
       $\vdots$
       $(S_1)(X_{s_k})(S_3)$,
       $(S_1 X)()(S_3)$.
       Again if some $X_{s_i}$ is a leaf g-item in $\mathcal{T}$, add it to
       the $S_1$ part instead.
4. If $S_2$ is empty, we generate the child nodes as defined
   in Subsection 5.2.3.
   (a) If $S_3$ is empty, this is a leaf node. Since the cor-
       responding 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 $\mathcal{T}$, according
       to Pruning Technique Three, we should plug-in
       MaxMiner (or similar tools) to process the sub-
       tree.
   (c) Otherwise, let $S_3$ be $\{X\} \cup S'_3$ where $X$ is a non-
       leaf g-item in $\mathcal{T}$. Generate the two child nodes
       $(S_1)(X)(S'_3)$ and $(S_1)()(S'_3)$ in the given order.
```

Fig.4. Classification-based algorithm for mining max frequent g-itemsets.

Our algorithm for finding max frequent g-itemsets can be summarized in Fig.4.

## 5.4 Online Elimination for False Positives

The algorithm of Subsection 5.3 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 $\mathcal{T}$ ($S_1 \supseteq_{\mathcal{T}} S'_1$), and if both are produced by our algorithm, then the algorithm will generate $S_1$ before $S'_1$.

**Theorem 4.** *Algorithm* **MFGI_class** *can find all max frequent g-itemsets. The generated candidates satisfy the superset-before-subset property.*

*Proof.* To see that **MFGI_class** finds all max frequent g-itemsets, we point out that: (a) due to the classification nature of the complete classification tree, all g-itemsets exist in the tree; and (b) whenever **MFGI_class** stops expanding a sub-tree, either a superset of all g-itemsets in the sub-tree is identified to be frequent, or it is certain that no g-itemset in the sub-tree can be frequent. The remainder of the proof focuses on proving the superset-before-subset property.

**MFGI_class** expands the tree in such a way that $(S_1)()()$ is always visited before $(S'_1)()()$ when $S_1 \supseteq_{\mathcal{T}} S'_1$. The ordering of the children in Subsections 5.2.2 and 5.2.3 guarantee that this is the case when the tree is expanded depth-first.

To see this, first observe that for any node $(S_1)(S_2)(S_3)$ in the classification, if $i_1 \in S_1$, $i_2 \in S_2$ and $i_3 \in S_3$, then $tax(i_1)$, $tax(i_2)$ and $tax(i_3)$ are pairwise disjoint. This follows from induction on the definition of child nodes in Subsection 5.2.

We can now demonstrate the superset-before-subset property. In the context of Subsection 5.2.3, note that any g-itemset produced from child node 1 must include at least one g-item from $tax(X)$. Since $tax(S_1)$ and $tax(S_3)$ are distinct from $tax(S_2)$, any g-itemset generated from child node 2 cannot include any g-item of $tax(X)$.

A similar argument applies in the context of Subsection 5.2.2. Any g-itemset produced by the $i$-th child node must include at least one g-item from $tax(X_i)$. Any g-itemset produced by a later child node cannot include any g-item from $tax(X_i)$. This is obvious for child nodes $i+1$ through $k$. For child node $k+1$, note that it literally includes $X$, but none of its descendants from $tax(X_i)$. □

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.

## 5.5 PHDB: Optimization to Batch-Compute Frequencies

So far we have ignored the discussion on how to compute frequencies. The discussion of **MFGI_class** implied a naive way: go through the transactional database each time the frequency of some g-itemset is needed. This is obviously inefficient, as 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 should be computed? 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 should we 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 that 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 $L = (l_0, l_1, \ldots, l_m)$ be a list of live levels, sorted in ascending order of depth.

2. Probabilistically choose a level, where the probability of choosing level $i$ is:

$$p_i = \begin{cases} \dfrac{(1-\sigma)^i}{\sum\limits_{j=0}^{m}(1-\sigma)^j}, & \text{if} \quad \sigma \geqslant 0; \\ \dfrac{(1+\sigma)^{m-i}}{\sum\limits_{j=0}^{m}(1+\sigma)^j}, & \text{if} \quad \sigma < 0. \end{cases}$$

3. Expand one child of the left-most live node from the chosen level.

4. If we have chosen *num* g-itemsets, stop; otherwise, goto Step 1.

To understand PHDB, let us study an example of probability distribution of choosing a level among four levels, with different values of $\sigma$. Here $S = 1 + 0.2 + 0.2^2 + 0.2^3$.

| | $\sigma = -1$ | $\sigma = -0.8$ | $\sigma = 0$ | $\sigma = 0.8$ | $\sigma = 1$ |
|---|---|---|---|---|---|
| Level 0 | 0 | $0.2^3/S$ | 1/4 | $1/S$ | 1 |
| Level 1 | 0 | $0.2^2/S$ | 1/4 | $0.2/S$ | 0 |
| Level 2 | 0 | $0.2/S$ | 1/4 | $0.2^2/S$ | 0 |
| Level 3 | 1 | $1/S$ | 1/4 | $0.2^3/S$ | 0 |

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.

### 5.6 Transaction Filtering: Optimization to Reduce Scan Sizes

Subsection 5.5 above described an optimization to greatly reduce the number of database scans needed by **MFGI_class**. However, even a small number of scans can be expensive for very large databases. To address this concern, we propose a database filtering method that greatly reduces the size of the database that must be scanned. This has the effect not only of scanning smaller databases, but also of allowing most of the database scans to happen in main memory, where the full database would not fit.

Each node in the classification tree used by **MFGI_class** specifies a subset of all g-itemsets. For a given set of g-itemsets, only a subset of the transactions in the database can possibly support one of those g-itemsets. Specifically, only transactions that contain all of the g-items in the $S_1$ part of the node and at least one of the g-items in $tax(S_2)$ can support the g-itemsets classified at a given node. So, at each classification node, we filter the database and scan only those transactions $t$ where $t \supset_T S_1 \cup S_2$. So, at each level of the classification tree we are working with smaller and smaller subsets of the transaction database.

## 6 Mining Essential Generalized Association Rules

Subsection 6.1 briefly discusses the lattice-based solution. The remaining of this section focuses on developing the classification-based solution.

### 6.1 Lattice-Based Solution

#### 6.1.1 Overview and Challenges

Here we define a lattice of g-rules, using the implication relationship among g-rules. An ancestor g-rule implies all its descendant g-rules in the lattice. To mine essential g-rules, we dynamically browse the lattice in a top-down fashion, generating child g-rules on the fly. Whenever a strong g-rule is found, all its descendants in the lattice can be pruned, because they are implied by the identified strong g-rule and therefore are not essential. The algorithm is denoted as **EGR_lattice**.

Clearly, the algorithm relies on a subroutine to generate the child g-rules of a given g-rule. It turned out that to design such a subroutine was very challenging.

Let us first study the algorithm that generates the child rules in the ordinary case. Again $r_2$ is a child of $r_1$ if and only if $r_1 \Longrightarrow r_2$ and $\nexists r_3$ s.t. $r_1 \Longrightarrow r_3 \Longrightarrow r_2$.

**Theorem 5.** *In the ordinary case (no taxonomy), a rule $r_2$ is a child rule of $r_1$ if and only if $r_2$ can be derived from $r_1$ by deleting one item from the consequent or by moving an item from the consequent to the antecedent.*

For instance, $A \rightarrow BC$ has four child rules. Two of them result from deleting an item from the consequent. They are $A \rightarrow B$ and $A \rightarrow C$. Two of them result from moving an item from the consequent to the antecedent. They are $AB \rightarrow C$ and $AC \rightarrow B$.

However, in the generalized case, the problem becomes much more subtle. For instance, instead of moving a g-item from the consequent to the antecedent, our study shows that we should either add some selected g-item (not necessarily in the original g-rule) to the antecedent, or replace some selected g-item in the antecedent by some other g-item (again, not necessarily in the original g-rule). Designing such an algorithm and guaranteeing its correctness are therefore very challenging.

### 6.1.2 *Algorithm GenChildGRules*

We hereby propose an algorithm called **GenChildGRules** (Fig.5) which generates the children of a g-rule *ante* $\rightarrow$ *cons* in the lattice, and a theorem which proves its correctness.

In all three cases, in order to perform some operation (e.g., **generalize** *i*) to generate a child rule, a g-item *i* may need to be deleted from *cons*. However, if *i* was the only g-item in *cons* and thus cannot be deleted (otherwise the child rule has an empty consequent, which is not valid), we do not perform the operation in the first place.

Let us examine an example. Consider the taxonomy and transactional database of Fig.1. Let *minsupport* $= \frac{1}{3}$ and *minconf* $= 1$. There are two max frequent g-itemsets: $\{AC\}$ and $\{Z\}$. The lattice of g-rules corresponding to $\{Z\}$ contains a single g-rule, $Y \rightarrow Z$, which is not strong (its confidence is $\frac{1}{3}$, which is smaller than *minconf*). The lattice corresponding to $\{AC\}$ is shown in Fig.6.

```
Algorithm. GenChildGRules
Input: A g-rule $ante \rightarrow cons$, and a taxonomy $\mathcal{T}$
Action: Generate all child g-rules in the lattice.
1.  for every item $i \in cons$,
        generalize $i$ to its parent in $\mathcal{T}$. If this results in
        an invalid rule, delete $i$ instead. Report the new
        rule.
2.  for every item $i \in cons$ such that $\nexists a \in ante$ more
    general than $i$
        Let $n$ be the most general ancestor of $i$ (including
        $i$ itself) in $\mathcal{T}$ such that $n \notin_{\mathcal{T}} ante$. Add $n$ to $ante$.
        If $n = i$, also delete $i$ from $cons$. Report the new
        rule.
3.  for every item $a \in ante$, for every child $c$ of $a$ in $\mathcal{T}$
    such that $c \in_{\mathcal{T}} cons$ (and hence $a \in_{\mathcal{T}} cons$),
        specialize $a$ from $ante$ to $c$. If $c \in cons$ (not just
        $c \in_{\mathcal{T}} cons$), also delete $c$ from $cons$. Report the
        new rule.
```

Fig.5. Generate all child g-rules.

To illustrate Case 1, consider generalizing g-items from the consequent of the root g-rule $Y \rightarrow AC$. If we generalize $A$ to $X$, we get a child g-rule $Y \rightarrow XC$. If we generalize $C$ to $Y$, we get an invalid g-rule $Y \rightarrow AY$. By deleting $Y$ from the consequent, we get the correct child g-rule $Y \rightarrow A$.

To illustrate Case 2, consider the g-rule $C \rightarrow A$. Case 2 can be applied here because there does not exist an item in *ante* more general than $A$. There are three candidate g-items which may be added to the antecedent in order to generate a child g-rule: $A$, $X$, and $Y$. Among them, we want to pick the most general one which $\notin_T$ *ante*. We pick $X$ here, since $Y \in_T C$, and $X$ is more general than $A$. The resulting child g-rule is $XC \rightarrow A$.

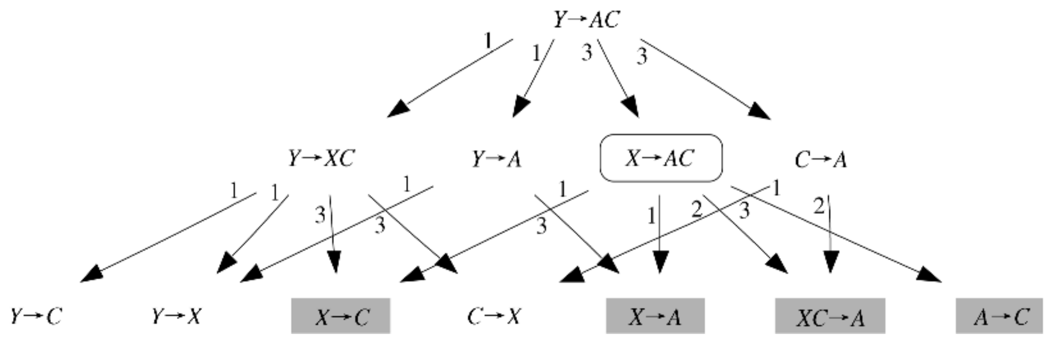


Fig.6. Lattice of g-rules implied by $Y \rightarrow AC$. Labels on edges refer to case numbers from Algorithm GenChildGRules.

To illustrate Case 3, consider the root g-rule $Y \rightarrow AC$ again. $Y$ has three children in $\mathcal{T}$. Namely, $X$, $C$, and $Z$. We cannot specialize $Y$ to $Z$, as $Z \notin_T \{AC\}$. Otherwise, there is no guarantee that the resulted g-rule ALWAYS has a larger or equal support and confidence. By specializing $Y$ to $X$ or $C$, we get the child g-rules $X \rightarrow AC$ and $C \rightarrow A$, respectively. Note that in the latter case, $C$ is removed from the consequent.

In the dynamic execution of **EGR_lattice**, at the second level $X \rightarrow AC$ is identified to be strong. Thus the examination of its children (the shaded g-rules of Fig.6) are omitted. Eventually, $X \rightarrow AC$ is reported as the only essential g-rule.

**Theorem 6.** *Given a g-rule, the algorithm* **GenChildGRules** *finds exactly the set of its child g-rules. By recursively applying algorithm* **GenChildGRules** *on the set of essential rules, one can find exactly the set of all strong rules.*

We have a long proof of the correctness of the above theorem, by relating the problem with its counterpart in propositional logic. The proof is omitted as the lattice-based solution is not the main focus of this paper. The theorem further emphasizes the importance of essential rule mining. In other words, given a set of essential g-rules, we can easily derive the set of all strong g-rules (if needed) without checking any frequency information from the transactional database.

## 6.2 Overview of the Classification-Based Algorithm

Similar to the lattice-based algorithm, the classification-based solution, **EGR_lattice**, starts with a set of max frequent g-itemsets. For each max frequent g-itemset $S$, a classification tree is dynamically constructed. Without pruning, the complete classification tree corresponds to all g-rules where the antecedent and the consequent are both subsets of $S$ (with regard to $\mathcal{T}$). The key to the algorithm's efficiency lies in its pruning techniques, when browsing the conceptual tree in a top-down fashion.

One may wonder: given that the lattice-based approach also performs a top-down browsing (of a lattice which is somewhat similar to a tree) with pruning, why do we need a classification algorithm? The answer is that in the classification tree, we have better pruning. As we will see, we can perform pruning "in both ways". That is, at each classification-tree node, we compute a lower bound *MINCONF* and an upper bound *MAXCONF* of the confidence of all g-rules in the subtree. The examination of the subtree can be pruned either because *MINCONF* $\geqslant$ *minconf* or because *MAXCONF* $<$ *minconf*. On the other hand, the lattice can only perform pruning "in one way". Only when a lattice node is confident can we prune its descendants.

The underlying reason for this difference is as follows. While each node of the lattice is a single g-rule, each node of the classification tree is a compact form representing all g-rules in the subtree. This carefully designed form should enable us to compute *MINCONF* and *MAXCONF*. Needless to say, this form of tree node should also enable us to classify the subtree. That is, from one tree node, we should be able to generate its child nodes, which are also in the same form, so that the classification can be recursively performed. The set of g-rules that a node represents should be exactly partitioned into the sets of g-rules that its children represent. In other words, any g-rule in the set of g-rules that a node represents should belong to one and only one set that some child node represents.

## 6.3 Conceptual Classification Tree

This subsection defines the conceptual classification tree of g-rules corresponding to one max frequent g-itemset. It is conceptual in the sense that no pruning is performed. (Pruning will be discussed in Subsection 6.4.)

Suppose $AC$ is a max frequent g-itemset. We build the classification tree of all g-rules whose antecedent and consequent $\subseteq_T AC$. Notice that any such g-rule only consists of g-items $A$, $C$ or their ancestor g-items in $\mathcal{T}$. Thus when performing the classification, we only need to consider the sub-taxonomy of Fig.2(a).

The root node of the classification tree should correspond to the complete set of g-rules (for any g-item from Fig.2(a)). Yet it should be in a succinct representation with constant space. That is to say, enumerating all g-rules at the root node is not a choice. By borrowing ideas from the itemset-classification tree, we can use the following to label the **root node**:

$$(\,)(Y)(\,) \rightarrow (\,)(Y)(\,)$$

Recall that $(\,)(Y)(\,)$ in the itemset-classification tree represents the set of all g-itemsets (which contain some g-item in the sub-taxonomy rooted by $Y$). Thus the above proposed root node succinctly represents all g-rules in the following sense: the tree contains exactly the g-rules where both the antecedent and the consequent contain g-items in Fig.2(a).

The root node indicates that a node $N$ in our proposed classification tree of g-rules has the following form:

$$(\textit{left}.S_1)(\textit{left}.S_2)(\textit{left}.S_3)$$
$$\rightarrow (\textit{right}.S_1)(\textit{right}.S_2)(\textit{right}.S_3),$$

where *left* and *right* refer to the left- and right-hand sides of a node of the classification tree. The components $S_1$, $S_2$ and $S_3$ then refer to the three components, which correspond to a node in the itemset-classification tree. Hence, for the node $(\,)(Y)(\,) \rightarrow (\,)(Y)(\,)$, the components *left*.$S_2$ and *right*.$S_2$ have the value $Y$, while all other components are the empty set. It remains to discuss how to generate the child nodes of $N$.

The basic idea is as follows. Consider the *left* part of $N$ as a node in the itemset-classification tree and get its child nodes. Similarly for the *right* part of $N$, get its child nodes in the itemset-classification tree. Then join the two sets of child nodes.

An arbitrary child of the left part of $N$ and an arbitrary child of the right part may not form a valid child node in the classification tree. Let us see an example. Consider the root node of the classification tree, i.e., $(\,)(Y)(\,) \rightarrow (\,)(Y)(\,)$. The left part and the right part are the same, which has three child nodes in the itemset classification tree (Fig.2(b)). They are: $(\,)(X)(C)$, $(C)(\,)(\,)$, and $(Y)(\,)(\,)$. A self-join of the three nodes leads to nine temporary child nodes, e.g., $(\,)(X)(C) \rightarrow (Y)(\,)(\,)$. However, any g-rule represented by this example child node has as the consequent $Y$ itself. But since $Y$ is the root of the taxonomy and is more general than any other g-item, whatever the antecedent is, the g-rule is invalid.

By removing the temporary child nodes where the right part is $(Y)(\,)(\,)$, we get six temporary child nodes. Note that when examining the child nodes of $N$'s left part, we read from right to left, e.g., in the order of $(Y)(\,)(\,)$ and then $(C)(\,)(\,)$ and finally $(\,)(X)(C)$. (This is needed to ensure that a depth-first traversal of the classification tree always meets a g-rule only after meeting all g-rules that imply it.) The six temporary children are:

$$(Y)(\,)(\,) \rightarrow (\,)(X)(C),\ (Y)(\,)(\,) \rightarrow (C)(\,)(\,),$$
$$\underline{(C)(\,)(\,) \rightarrow (\,)(X)(C)},\ \underline{(C)(\,)(\,) \rightarrow (C)(\,)(\,)},$$
$$(\,)(X)(C) \rightarrow (\,)(X)(C),\ \underline{(\,)(X)(C) \rightarrow (C)(\,)(\,)}.$$

Here the three underlined temporary children need to be changed. First of all, it is clear that $(C)(\,)(\,) \rightarrow (C)(\,)(\,)$ is not a valid node. It represents a single g-rule $C \rightarrow C$ which is invalid. In general, a temporary node in the classification tree is invalid, if any g-rule represented by it contains a g-item in the consequent which is more general than or equal to a g-item in the antecedent. More formally,

• a classification tree node, *left* $\rightarrow$ *right*, is invalid and thus should not be generated, if $\exists r \in \textit{right}.S_1$, such that $r \in_T \textit{left}.S_1 \cup \textit{left}.S_2$.

It is critical not to confuse $\in$ (literally a member) with $\in_T$ (belongs to, with respect to a taxonomy $\mathcal{T}$) in this criterion. Note that in $(Y)(\,)(\,) \rightarrow (C)(\,)(\,)$, even though $Y \in_T \textit{right}.S_1$ and $Y \in_T \textit{left}.S_1$, $(Y)(\,)(\,) \rightarrow (C)(\,)(\,)$ still remains valid.

The other two underlined nodes also need change. Unlike the previous case, these nodes are valid and should remain in the classification tree. However, for efficiency reason they need to be altered according to the following rules.

• Remove every item $l$ in *left*.$S_3$ if $\exists$ a g-item $r$ in *right*.$S_1$ such that $r \in_T \{l\}$. As an example, in node $(\,)(X)(C) \rightarrow (C)(\,)(\,)$, $C$ in *left*.$S_3$ is omitted. We know that any represented g-rule has a consequent $C$. Thus such a g-rule is invalid if its antecedent contains $C$ (or some descendant g-item of $C$ in the taxonomy, if $C$ were an non-leaf g-item). So there is no point keeping $C$ in *left*.$S_3$. The equivalent node is $(\,)(X)(\,) \rightarrow (C)(\,)(\,)$.

• Remove every item in *right*.$S_3$ if it is a leaf g-item which is equal to some g-item in *left*.$S_1$. As an example, in node $(C)(\,)(\,) \rightarrow (\,)(X)(C)$, $C$ in *right*.$S_3$ should be omitted.

The complete classification tree of g-rules corresponding to the sub-taxonomy of Fig.2(a) is shown in Fig.7.

## 6.4 Algorithm EGR_Class

The classification-based algorithm that generates essential g-rules, which is named **Algorithm EGR_class**, is summarized in Fig.8.

Below we discuss the three pruning techniques and the subroutine **MINCONFProcessing**.

### *6.4.1 MINCONF and MAXCONF Pruning*

Let us start with computing $\textit{MINCONF}(N)$ (the computation of $\textit{MAXCONF}(N)$ is similar). Below we will use *left* and *right* to represent the left and right part of $N$, both in the form of $(S_1)(S_2)(S_3)$. Consider an arbitrary g-rule *ante* $\rightarrow$ *cons* represented by $N$. The confidence of *ante* $\rightarrow$ *cons* is:

$$\frac{\textit{support}(\textit{ante} \cup_T \textit{cons})}{\textit{support}(\textit{ante})}.$$

To minimize the confidence (i.e., the ratio), we want to minimize the numerator and maximize the denominator. We know a subset has a larger (or equal) support

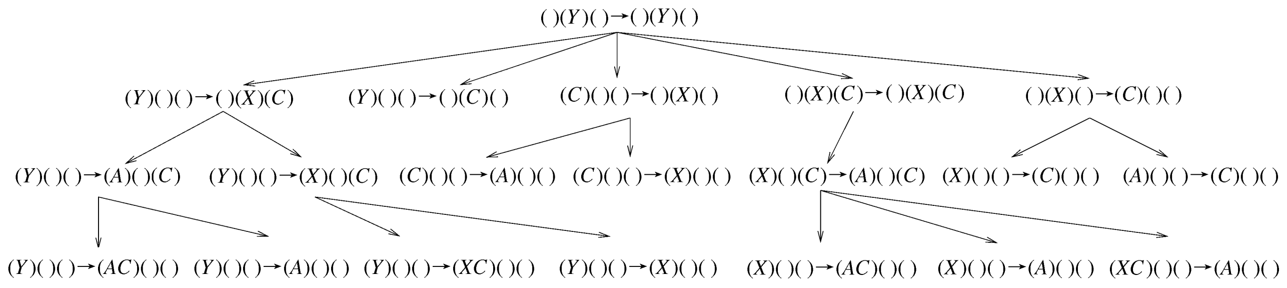


Fig.7. Classification tree of g-rules.

```
Algorithm. EGR_class
Input: A set of max frequent g-itemsets, and a taxonomy $\mathcal{T}$
Action: Generate all essential g-rules
1.  for every max frequent g-itemset $S$
      (a) Derive a sub-taxonomy $\mathcal{T}_{sub}$ from $\mathcal{T}$, treating the
          g-items in $S$ as leaf g-items, and only keeping these
          g-items and their ancestors in $\mathcal{T}$. (As an example,
          Given $\mathcal{T}$ in Fig.1(a) and a max frequent g-itemset
          $AC$, the derived sub-taxonomy $\mathcal{T}_{sub}$ is shown in
          Fig.2(a).)
      (b) Generate the root node of the classification tree as
          $()(Y)() \rightarrow ()(Y)()$, where $Y$ is the root of $\mathcal{T}_{sub}$.
      (c) Dynamically generate the descendant nodes in the
          classification tree, following a depth-first fashion.
      (d) At each node $N$, try to prune it using the following
          pruning techniques:
            i.  The MAXCONF Pruning:
                If $MAXCONF(N) < minconf$, prune the sub-
                tree as no g-rule can be confident.
            ii. The MINCONF Pruning:
                If $MINCONF(N) \geqslant minconf$, call Algorithm
                MINCONFProcessing to directly generate
                the essential g-rules in the subtree.
            ii. The Implication Pruning:
                If $\exists$ an identified essential g-rule $r$ such that all
                g-rules represented by $N$ are implied by $r$, the
                subtree rooted by $N$ can be pruned.
2.  end for
```

Fig.8. Classification-based algorithm for mining essential g-rules.

than that of its supersets. So to maximize the denominator, we want the antecedent to have as few g-items as possible, while each g-item should be as general as possible. Such an antecedent should be: $left.S_1 \cup left.S_2$. It is easily understandable that every g-item in $left.S_1$ should appear in *ante* as it is, according to the definition of $S_1$. If the $S_2$ part is non-empty, it should be a single g-item, and either this g-item or some of its descendant g-item should appear in the antecedent of every g-rule in the subtree. To maximize support (*ante*), we pick the g-item (given in $left.S_2$) itself and not a descendant of it.

On the other hand, to minimize the numerator, we should pick *ante* to be $left.S_1 \cup_T leaf(left.S_2 \cup left.S_3)$, and we should pick *cons* to be $right.S_1 \cup_T leaf(right.S_2 \cup right.S_3)$.

**Definition 7.** *Given a classification tree node $N$ in the form of left $\rightarrow$ right,*

$$MINCONF(N) =$$

$$\frac{support(left.S_1 \cup_T leaf(left.S_2 \cup left.S_3) \cup_T right.S_1 \cup_T leaf(right.S_2 \cup right.S_3))}{support(left.S_1 \cup left.S_2)}$$

$$MAXCONF(N) = \frac{support(left.S_1 \cup_T left.S_2 \cup_T right.S_1 \cup_T right.S_2)}{support(left.S_1 \cup_T leaf(left.S_2 \cup left.S_3))}$$

**Theorem 7.** *$MINCONF(N)$ and $MAXCONF(N)$ are a lower bound and an upper bound of the confidence of every g-rule represented by a classification tree node $N$.*

One may have noticed that when calculating *MINCONF*, we picked different *ante* for the numerator and the denominator. This is necessary to compute a lower bound of confidence for g-rules in the subtree. However, it raises a significant issue. Picking different *ante* may not lead to any g-rule. So if we omit the generation of the subtree, what g-rule(s) shall we report as essential g-rules?

We first observe that the essential g-rules in the subtree should be able to be identified without consulting the transactional database for any frequency information. Since all g-rules in the subtree are strong, we can directly generate all the g-rules in the subtree and compare them to each other and remove the implied ones. The g-rules not implied by any other one are reported as essential g-rules. But of course this is very inefficient as there may have many g-rules in the subtree. We hereby propose an algorithm to efficiently identify the essential g-rules among them.

### 6.4.2 Algorithm MINCONFProcessing

Given a classification tree node $N$ : *left* $\rightarrow$ *right*, suppose we know all g-rules in the subtree are strong, our task is to design an algorithm that efficiently identifies the set of essential g-rules in the subtree.

**Observation 1.** *Any essential g-rule in the subtree of $N$ must have $right.S_1 \cup_T leaf(right.S_2 \cup right.S_3)$ as consequent.*

*Proof.* Let $cons_0 = right.S_1 \cup_T leaf(right.S_2 \cup right.S_3)$. By the definition of the form of a classification tree node, $cons_0$ is a valid consequent. It remains to prove that any g-rule whose consequent is not $cons_0$ cannot be essential. Let $ante \to cons_1$ be an arbitrary g-rule in the subtree of $N$, where $cons_1 \neq cons_0$. We have: $cons_1 \subseteq_T cons_0$. Thus according to Theorem 1, $ante \to cons_0 \Longrightarrow ante \to cons_1$. Therefore $ante \to cons_1$ is not essential. □

**Observation 2.** *The g-rule $r_0 = left.S_1 \cup left.S_2 \to right.S_1 \cup_T leaf(right.S_2 \cup right.S_3)$ is essential.*

*Proof.* If $r_0$ were not essential, there must exist another g-rule $r_1 : ante \to cons$ that implies it. It follows that $ante \subseteq_T left.S_1 \cup left.S_2$ and $cons \supseteq_T right.S_1 \cup_T leaf(right.S_2 \cup right.S_3)$. But according to the definition of the form of a classification tree node, $r_0$ already has the "smallest" antecedent and the "largest" consequent among all g-rules in the subtree of $N$. It follows that $r_1$ and $r_0$ are exactly the same. □

For the discussion below we will keep the notations:

$$ante_0 = left.S_1 \cup left.S_2,$$
$$cons_0 = right.S_1 \cup_T leaf(right.S_2 \cup right.S_3),$$
$$r_0 = ante_0 \to cons_0.$$

Any other essential g-rule in the subtree of $N$ (other than $r_0$) can be viewed as a transformation from $r_0$ in the following way: *add some g-items from $tax(left.S_2 \cup left.S_3)$ to the antecedent.* Of course, if some g-item in $tax(left.S_2)$ is added to the antecedent, the g-item in $S_2$ itself (which was in the original antecedent of $r_0$) should be removed.

**Observation 3.** *If one of the g-items, selected from tax $(left.S_2 \cup left.S_3)$ and added to $ante_0$, belongs to $cons_0$ with regard to $\mathcal{T}$, the resulted g-rule is* not *essential.*

*Proof.* Suppose we select the itemset $S \cup \{i\}$ from $tax$ $(left.S_2 \cup left.S_3)$ and add to $ante_0$, where $i \in_T cons_0$. The resulted g-rule is $S \cup \{i\} \cup_T ante_0 \to cons_0$. We argue that it is not essential because it is implied by $S \cup_T ante_0 \to cons_0$. To see the implication relationship (Theorem 1), it is enough to point out that $S \cup \{i\} \cup_T ante_0 \cup_T cons_0 \subseteq_T S \cup_T ante_0 \cup_T cons_0$ since $i \in_T cons_0$. □

It is now clear how to generate the essential g-rules in the subtree of $N$. We should examine $tax(left.S_2 \cup left.S_3)$ and eliminate all g-items $i$ such that $i \in_T cons_0$. From the remaining g-items, we enumerate all valid g-itemsets. Adding every such g-itemset to the antecedent of $r_0$ will result in a new essential g-rule. We denote this algorithm as **MINCONFProcessing**.

As an example, consider the classification tree node $N = ()(X)(CZ) \to (A)(Z)()$, regarding to the taxonomy of Fig.1(a). Let $MINCONF(N) \geqslant minconf$. Let us run **MINCONFProcessing** to find the essential g-rules in the subtree of $N$.

First, $r_0 = X \to ADE$ is an essential g-rule. Then, if we consider $tax(XCZ)$ and remove all g-items $i$ such that $i \in_T ADE$, we get two g-items $B$ and $C$ left (Fig.9). Therefore there are three more essential g-rules: $B \to ADE$, $XC \to ADE$, and $BC \to ADE$.

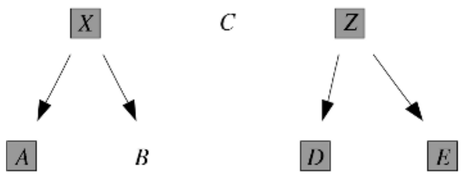


Fig.9. Illustration of $tax(XCZ)$ with g-items $\in_T$ $ADE$ being removed.

### 6.4.3 Implication Pruning

It is possible to prune subtrees using the identified essential g-rules. As long as we are sure that all g-rules represented by a classification tree node $N : left \to right$ are implied by an identified essential g-rule $ante_1 \to cons_1$, we can omit the expansion of $N$. This is called **Implication Pruning**. In more detail, the pruning condition is:

- $ante_1 \subseteq_T left.S_1 \cup left.S_2$, and
- $ante_1 \cup_T cons_1 \supseteq_T left.S_1 \cup_T leaf(left.S_2 \cup left.S_3 \cup right.S_2 \cup right.S_3)$.

### 6.4.4 Example

To conclude this section, let us run the algorithm **EGR_class** to find essential g-rules in Fig.1, where $minsupport = 1/3$ and $minconf = 1$. The result is illustrated in Fig.10.

At the root node $()(Y)() \to ()(Y)()$,

$$MINCONF = \frac{support(AC)}{support(Y)} = \frac{1}{2} < minconf,$$

$$MAXCONF = \frac{support(Y)}{support(AC)} = 2 \geqslant minconf.$$

Therefore none of MINCONF Pruning or MAXCONF Pruning can be applied. The Implication Pruning cannot be applied either since there is no essential g-rule identified so far.

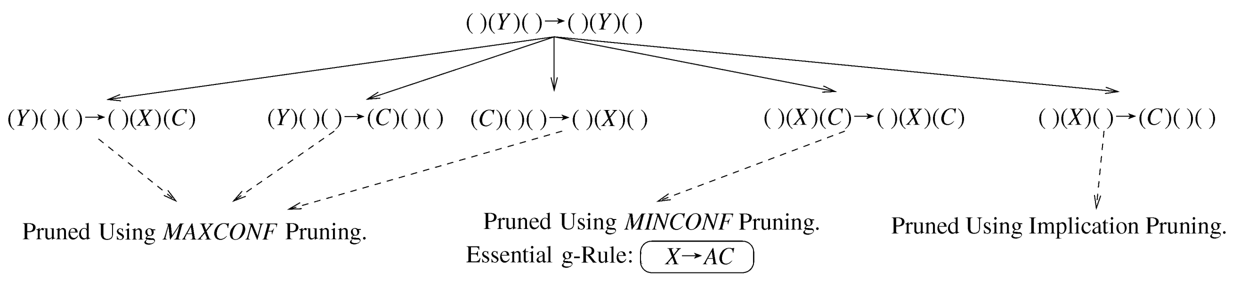


Fig.10. Finding essential g-rules using the algorithm **EGR_class**.

At the first child node $(Y)()() \rightarrow ()(X)(C)$, since

$$MAXCONF = \frac{support(X)}{support(Y)} = \frac{1}{2} < minconf,$$

the corresponding subtree can be pruned due to MAXCONF Pruning. Similarly, the second and third child subtree can also be pruned using the MAXCONF Pruning (the corresponding MAXCONF are $\frac{support(C)}{support(Y)} = 2/3$ and $\frac{support(CX)}{support(C)} = 3/4$, respectively).

The fourth child node, $()(X)(C) \rightarrow ()(X)(C)$, can be pruned using the MINCONF Pruning. This is because

$$MINCONF = \frac{support(AC)}{support(X)} = 1 \geqslant minconf.$$

There is a single essential g-rule in the subtree: $X \rightarrow AC$.

Finally, the last child node $()(X)() \rightarrow (C)()()$ can be pruned by the Implication Pruning, using the identified essential g-rule.

## 7 Experimental Analysis

Here, we provide an experimental analysis of both **MFGI_class** and **EGR_class**.

As far as we know, the naive lattice-based algorithm MFGI_ lattice is the only other algorithm that has been designed to mine the set of max frequent g-itemsets. So the first set of experiments we conduct is to compare our algorithm **MFGI_class** with MFGI_lattice.

Next, we compare **MFGI_class** with BASIC[3]. Note that BASIC was proposed to find all frequent g-itemsets. So we give **MFGI_class** the additional handicap of producing all frequent g-itemsets from the set of identified max frequent g-itemsets. Furthermore, in the first two sets of experiments we do not apply the PHDB optimization in the comparison graphs.

We also provide the following additional results for **MFGI_class**: a measurement of the effect of applying the PHDB optimization and experiment with different choices of the parameter $\sigma$; the effects of the transaction filtering optimization; the performance as database size scales.

Then, we analyze **EGR_class**, including a comparison to **EGR_lattice**, including variations in minimum support, minimum confidence, taxonomy depth, and database size.

Finally, we examine the performance of **MFGI_class** and **EGR_class** with respect to a real dataset, the Microsoft Anonymous Web Data set[30].

The algorithms were implemented in Sun Java 1.5.0, and executed on a Sun Blade 1500 with 1GB of memory running SunOS 5.9.

The synthetic experimental data were generated with the widely used Quest Synthetic Generator[31]. The specific properties of each of the datasets will be described in detail in the following subsections.

### 7.1 MFGI_class vs. Naive Lattice-Based Enumeration

We compare **MFGI_class** with the lattice-based approach MFGI_lattice. Because of the large relative inefficiency of the lattice approach, this comparison uses 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 3 demonstrates the practical infeasibility of lattice-based methods for mining max frequent g-itemsets.

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.

**Table 3.** Comparison Against the Lattice-Based Approach

| *N* | Lattice | | |
|---|---|---|---|
| | Nodes | g-Itemsets | Time (s) |
| 15 | 2645 | 668 | 3.9 |
| 31 | $3.6 \times 10^6$ | $4.5 \times 10^5$ | $3 \times 10^4$ |
| *N* | MFGI_class | | |
| | Nodes | g-Itemsets | Time (s) |
| 15 | 33 | 92 | 1.2 |
| 31 | 83 | 243 | 1.5 |

## 7.2 MFGI_class vs. BASIC

We choose BASIC as the baseline algorithm for comparison, following the tradition of previous authors[3,7−13]. BASIC has been widely used as a baseline algorithm because it has a clear, standard implementation whose speed will not be greatly biased by the implementation. This is not true of many of the other algorithms in the literature.

For example, we choose not to compare with the FP-tax[2] because its performance may vary significantly depending on the implementation. As one specific issue, it is not clear how to handle the case when the FP-tree does not fit in memory.

The speedup over BASIC that we achieve, especially for taxonomies of depth 4 and greater, are significantly beyond what is achieved by other algorithms for mining frequent g-itemsets.

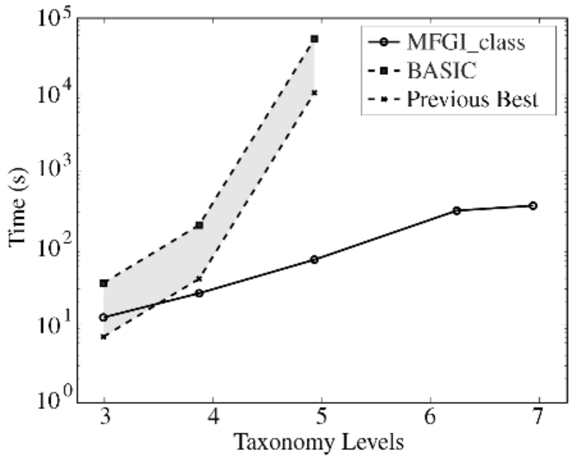
Since Srikant and Agrawal also presented Cumulate and EstMerge[3] and reported that they are 2 to 5 times faster than BASIC, in the performance graphs we include a band of a factor of 5 in the speed of BASIC.

The performance of Cumulate and EstMerge should both lie inside that band. Additionally, Srikant and Agrawal report that for one natural data set (department store), Cumulate and Estmerge were shown to be 100 times faster than BASIC. However, that example occurs only for an 8 level taxonomy. Extrapolating our own data, **MFGI_class** appears to be more than 100 million times faster than BASIC for this case.

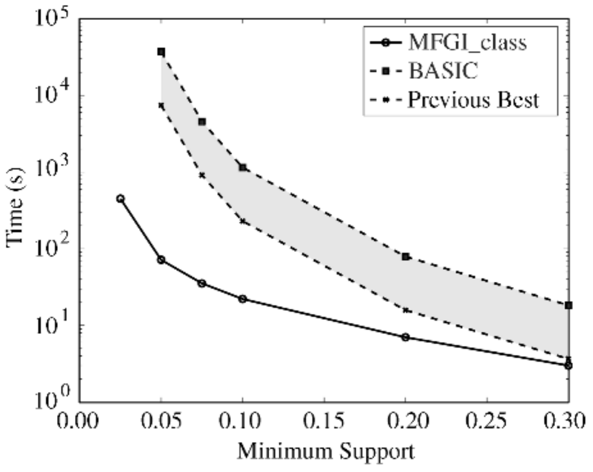
**Table 4.** Parameters for Dataset Generation

| Parameter | Default Value |
|---|---|
| Taxonomy | |
| Number of Items | 1 000 |
| Number of Levels | 5 |
| Transactional Database | |
| Number of Transactions | 10 000 |
| Average Size of Transaction | 5 |
| Transaction Size Distribution | Poisson |
| Number of Patterns | 300 |
| Average Length of Patterns | 4 |

Table 4 presents the default parameters used for experimental data generation.



(a)



(b)

Fig.11. Comparison against BASIC. (a) Varying levels. (b) Varying min_support.

Fig.11(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 *minsupport* = 0.05. In the graph, the "previous best" line was manually generated by taking 1/5 of the execution time of 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 12 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.11(b) shows the performance comparison by varying *minsupport* 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.

## 7.3 Effect of the PHDB Optimization

We use the same data as described in Table 4. 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 5. 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*.

**Table 5.** Speedup on the Number of Database Scans Due to PHDB

| *num* | 1 | 5 | 10 | 20 | 50 | 100 |
|---|---|---|---|---|---|---|
| Speedup | 1 | 2.5 | 4.4 | 6.2 | 11.7 | 12.3 |

Finally, we experiment with the choice of parameter $\sigma$. In Fig.12(a), when the number of frequencies to check per scan is *num* = 100, the best choice of $\sigma$ is $-1$. In fact, our experiments revealed that for most practical *num*, the depth-first approach is the best. When *num* is very small, values of $\sigma > -1$ can be the best choice. For example, in Fig.12(b) where *num* is as small as 7, the best choice of $\sigma$ is $-0.8$. We recommend using $\sigma = -1$ (i.e., the depth-first approach) as a default value in most scenarios, though application specific tuning may provide additional benefits.

Even though we find the depth-first expansion order to be optimal in most cases, we still believe that the PHDB expansion method is worth mentioning here for the following reasons. First, it was not previously known that depth-first expansion is optimal in this case. Second, the depth-first expansion order may not by optimal for all possible data sets. Other expansion orders can easily be explored through the use of PHDB for application specific tuning. Finally, PHDB is a general tree expansion ordering technique that can be applied outside the algorithm **MFGI_class**.

## 7.4 Effect of the Transaction Filtering Optimization

Here we again use the same data as described in Table 4, with a taxonomy of depth 5 and *minsupport* = 0.05.

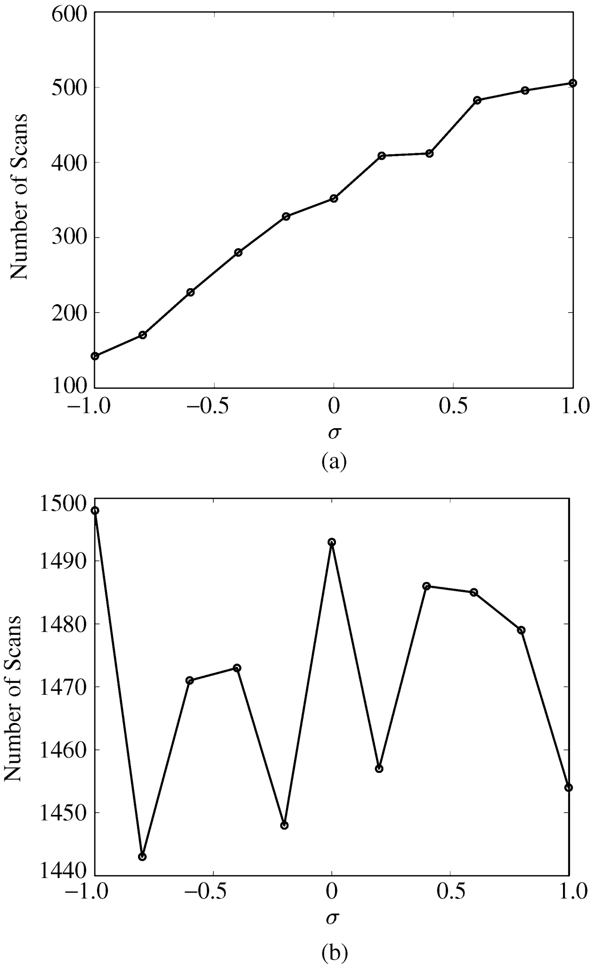


Fig.12. Impact of the parameter $\sigma$ in PHDB. (a) *num* = 100. (b) *num* = 7.

As described in Subsection 5.6, we only need to scan a subset of the transactions in the database at each node in the classification tree used by **MFGI_class**. Fig.13 analyzes how effective this optimization is. Here, the x-axis represents the size of a subset of the database, as fraction of the total database size. The y-axis is the number of database scans, as a fraction of the total number of scans used by the algorithm, that scan a subset of the database smaller than the values on the x-axis. So, approximately 61% of the scans in this case were on a database less than 10% the size of the entire database. Further, approximately 94% of the scans were on a database less than 25% the size of the entire database. In practical terms, the x-axis can be considered the size of main memory and the y-axis the number of scans that can be completed without accessing disk.

Note, these results are independent of the scan combining optimization using PHDB. Combining the two optimizations leads to a very small number of scans, most of which can be completed in main memory.

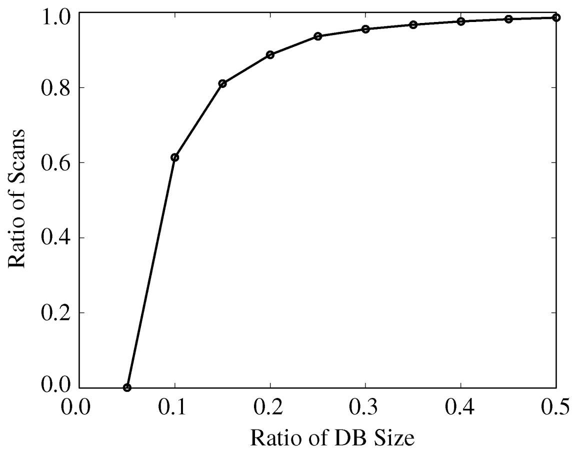


Fig.13. Impact of the transaction filtering optimization.

## 7.5 Scalability, Number of Results, and Percentage of False Positives

Some other experimental results are reported in this subsection. Fig.14 shows that the running time **MFGI_class** scales linearly with increasing database size. That is, the running time per transaction is approximately constant. In this case, databases with a number of transactions ranging from ten thousands to one million were used.

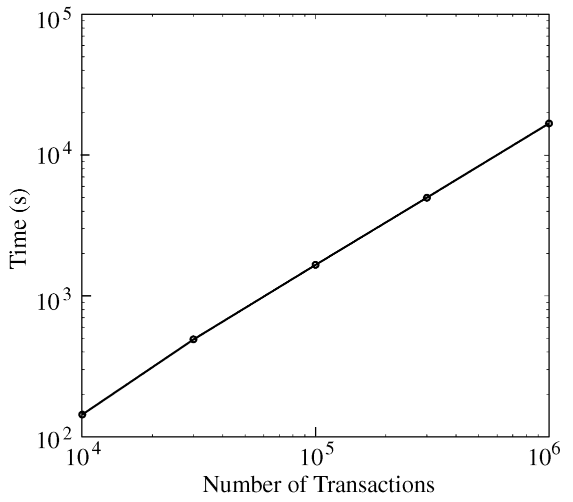


Fig.14. Scaling up database size.

Fig.15 shows the number of MFGI found for various levels of minimum support. The exponential increase in the number of MFGI with decreasing minimum support corresponds closely to the increase in running time shown by Fig.11(b), suggesting that the running time of **MFGI_class** is dependent on the number of MFGI found.

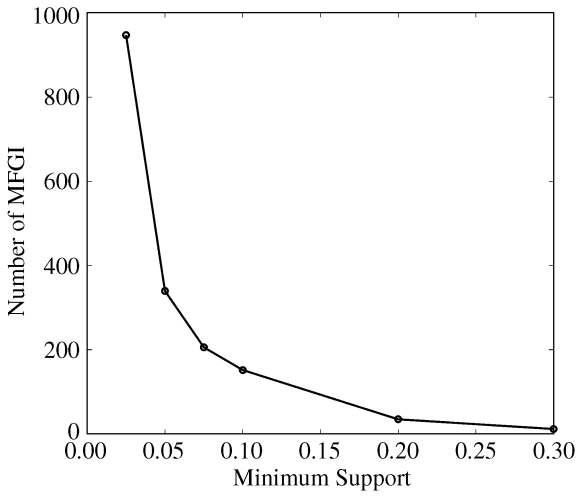


Fig.15. Number of MFGI with varying minimum support.

**Table 6.** Ratio of False Positives

| Min. Support | MFGI | False MFGI | Ratio |
|---|---|---|---|
| 0.3 | 12 | 44 | 3.7 |
| 0.2 | 35 | 115 | 3.3 |
| 0.1 | 152 | 517 | 3.4 |
| 0.075 | 206 | 969 | 4.7 |
| 0.05 | 340 | 2122 | 6.2 |
| 0.025 | 947 | 7645 | 8.1 |

As mentioned in Subsection 5.4, **MFGI_class** produces a number of "false" MFGI along with the actual set of MFGI. Table 6 shows the ratio between the number of false positives to the number of actual max frequent g-itemsets, corresponding to the number of max frequent g-itemsets shown in Fig.15.

## 7.6 Comparison of Algorithms for Mining g-Rules

This section provides an experimental comparison of the algorithms **EGR_lattice** and **EGR_class**.

Fig.16 compares the running time of **EGR_lattice** and **EGR_class**, varying *minconf* from 0.3 to 0.9. Here we fix *minsupport* at 0.3. At low minimum confidence levels, 0.3 to 0.5, both methods are fast as the lattices/trees are shallow. The lattice method is somewhat faster at these levels because there is no computation to determine if pruning should occur and there are very few pruning opportunities. As the minimum confidence increases the lattices/trees become much deeper and the pruning becomes more important. In this case, **EGR_class** outperforms **EGR_lattice** by a wide margin. For example, **EGR_class** is about 10 times faster when using a minimum confidence of 0.9.

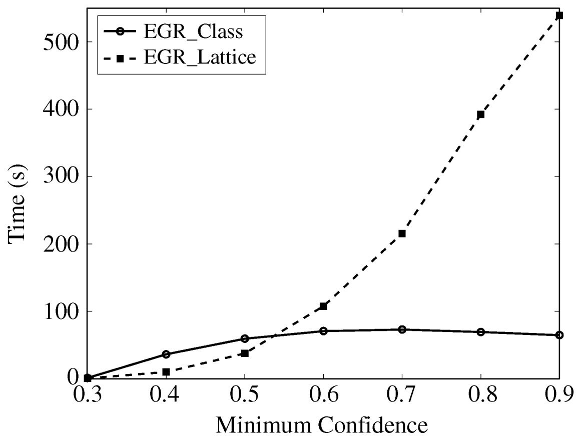


Fig.16. Varying *minconf*.

Fig.17 shows another comparison between **EGR_lattice** and **EGR_class**, this time fixing the minimum confidence at 0.8 and ranging minimum support from 0.05 to 0.3 (note the logarithmic scale on the y-axis). **EGR_class** is faster than **EGR_lattice** in all cases. In fact, below a minimum support of 0.2 **EGR_lattice** becomes infeasible due to excessive time and memory requirements. At a minimum support of 0.2, **EGR_class** is about 700 times faster than **EGR_lattice**.

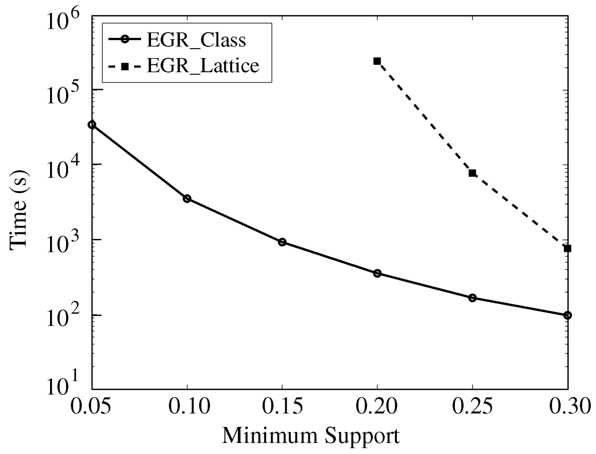


Fig.17. Varying *minsupport*.

Fig.18 fixes the minimum support at 0.3 and the minimum confidence at 0.8 and ranges the depth of the taxonomy from three to seven. The rest of the taxonomy and transaction database parameters were at their defaults, detailed in Table 4. **EGR_lattice** is faster with very shallow taxonomies because there are fewer pruning opportunities to take advantage of. With taxonomies of depth 5 and above, **EGR_class** is much faster than **EGR_lattice**. For instance, when the taxonomy depth is 6, **EGR_class** is about 200 times faster than **EGR_lattice**. The reason is that as the taxonomy increases in depth the size of the conceptual (i.e., unpruned) lattice and classification tree grows. In the case of **EGR_lattice**, there is no effective method for countering this growth and therefore the running time increases exponentially. However, **EGR_class** utilizes a number of pruning methods to combat this growth. As the taxonomy increases in depth these pruning methods increase in effectiveness. After depth 6, the growing effectiveness of the pruning overtakes the growing size of the conceptual classification tree and the computation time begins to decrease.

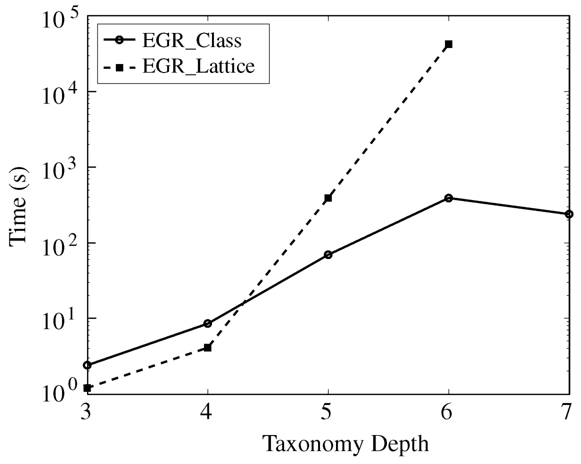


Fig.18. Varying taxonomy depth.

Overall, **EGR_class** is far superior to **EGR_lattice**, unless the minimum support is very high, the minimum confidence is very low, or the taxonomy is very shallow.

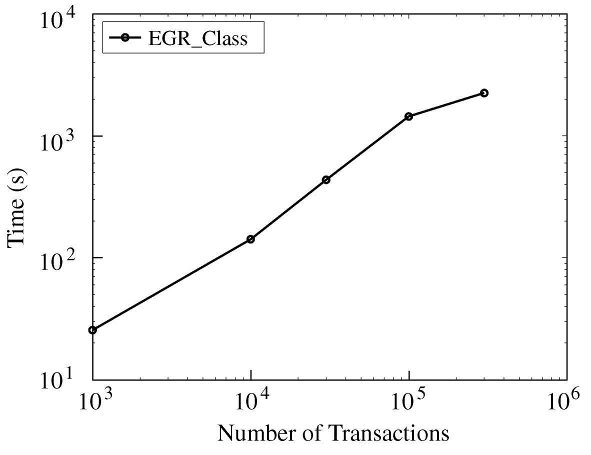


Fig.19. Database size scale-up.

Fig.19 shows the running time of **EGR_class** for increasing sizes of the transactional database. Here we use logarithmic scales for both X and Y axes. Clearly, **EGR_class** scales linearly with the number of transactions.

Finally, Fig.20 compares the usefulness of the three pruning techniques by showing the number of occurrences of each type of pruning. The greater the number of occurrences, the more useful the corresponding pruning technique. Here we choose *minsupport* = 0.3 and vary *minconf*. The figure reveals that at low minimum confidence levels, the number of MINCONF pruning occurrences is much greater than MAXCONF pruning occurrences. With increasing minimum confidence this relationship reverses. The level of implication pruning is relatively constant and low throughout.

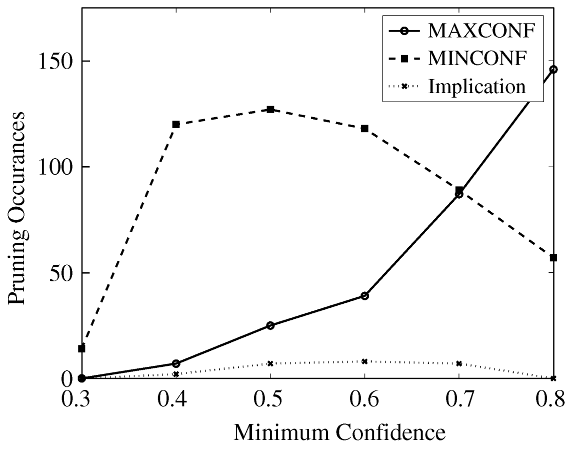


Fig.20. Comparing the effectiveness of the three pruning techniques.

## 7.7 Experimental Results on Real Data

Here we provide results for our algorithms using the Microsoft Anonymous Web Data set, available from the UCI KDD Archive[30]. This data represents the areas of the web site `http://www.microsoft.com` that were visited by 37 711 random users for a one-week time period in 1998. There is one transaction for every user. There are 294 items, representing the areas of `www.microsoft.com` that users visited. Transactions have an average of 3.02 items. Originally, the data had no taxonomy. However, descriptions of each item allowed us to classify each as being a child item of one of 19 g-items. These 19 g-items represent classes of web pages that users visited. They included such classes as "Consumer Products", "Support", "Development", and "News".

Fig.21 shows performance measurements for mining max frequent g-itemsets using **MFGI_class**. The running time and number of MFGI are shown for minimum support values ranging from 0.025 to 0.4.

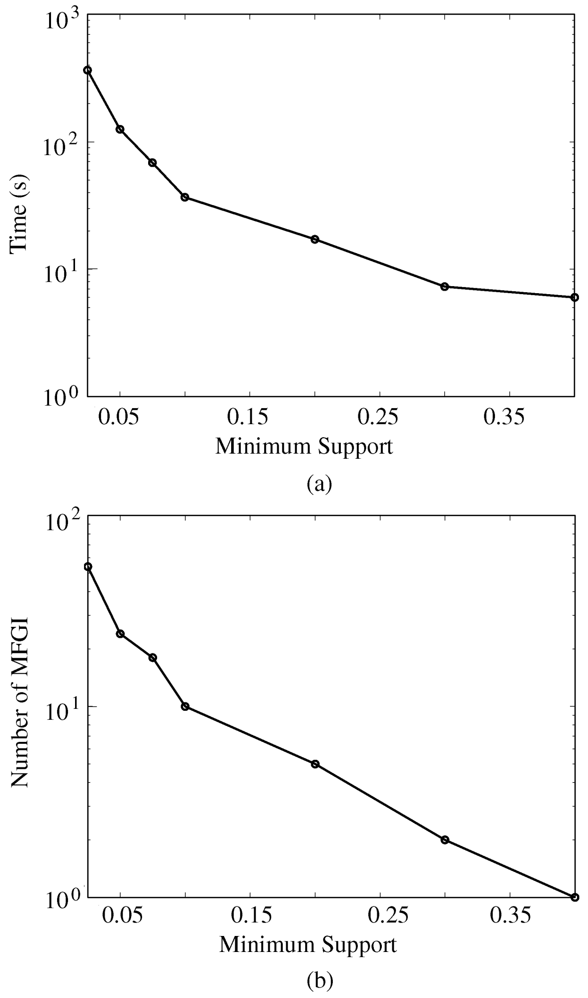


Fig.21. MFGI_class applied to MS Web Data. (a) Running time. (b) Number of MFGI.

Fig.22 shows performance measurements for mining essential g-rules using **EGR_class**. First, the running time and number of g-rules are shown for a minimum confidence value of 0.2 and minimum support values ranging from 0.025 to 0.4. Second, minimum support is fixed at 0.075 and minimum confidence ranges from 0.2 to 0.8. Notice that the running time and number of resulting rules are more sensitive to changes in minimum support than in minimum confidence (hence the logarithmic scales on the first two plots).

Note that the results above correspond to the base version of the algorithms, without any of the optimization techniques previously examined.

Table 7 shows a sampling of generalized association

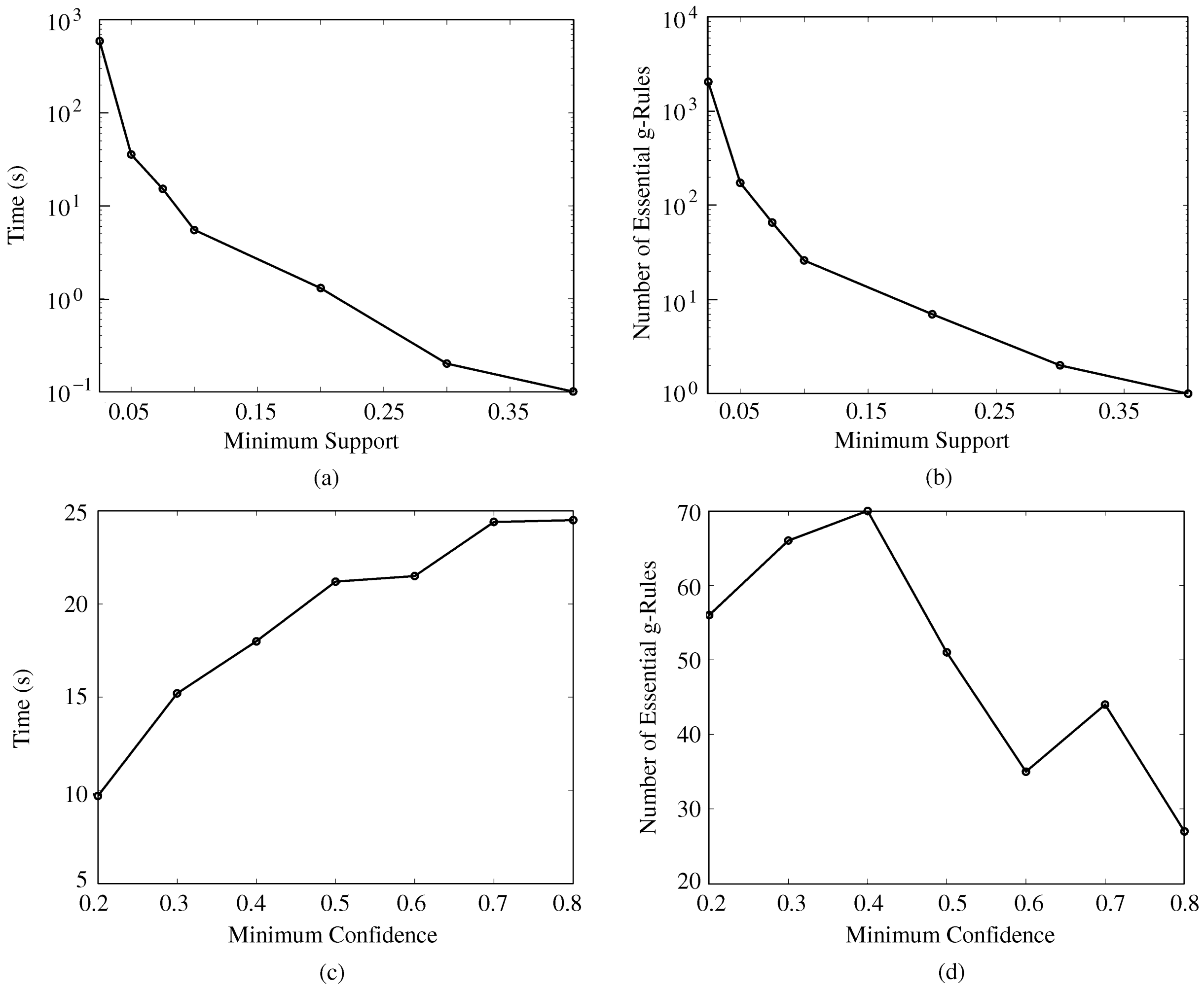


Fig.22. EGR_class applied to MS Web Data. (a) Running time. (b) Number of essential rules. (c) Running time. (d) Number of essential rules.

**Table 7.** Some Generalized Association Rules Mined from MS Web Data

| | | |
|---:|:---:|:---|
| {*Consumer Products, *Downloads} | → | {Internet Explorer, Free Downloads} |
| {*Operating Systems} | → | {Windows Family of OSes} |
| {*Consumer Products, *Web Site} | → | {Microsoft.com Search} |
| {Internet Site Construction} | → | {*Online} |
| {*Consumer Products, *Development} | → | {*Downloads} |

rules found with a minimum support level of 0.1 and a minimum confidence level of 0.5. A "$*$" indicates that the item is a generalized item representing a category of web site areas. All other items are actual web site areas present in the transactional data.

## 7.8 A Note on Memory Usage

Both **MFGI_class** and **EGR_class** are forms of depth-first search (DFS) with pruning. As such, they have the same memory requirements as DFS, namely logarithmic in the size of the classification trees. From the counting arguments presented in Section 4, we know that the size of the classification trees is exponential in the number of g-items. So, the two algorithms use space proportional to the number of g-items (in the worst case).

This will typically be only a small fraction of the available memory. The remaining memory can therefore be put to use in two optimizations for **MFGI_class**: PHDB for batch-computing frequencies; and, transaction filtering (see Subsections 5.5 and 5.6). In PHDB, extra memory allows more g-itemsets to be scanned for frequency in one pass. For transaction filtering, extra memory allows even larger filtered databases to be scanned in memory. So, these optimizations can avoid costly disk-scans, especially when larger amounts of memory are available.

## 8 Conclusions

This paper solved, for the first time, the problems of mining max frequent g-itemsets and essential g-rules. Our algorithms, **MFGI_class** and **EGR_class** are both based on a conceptual classification tree (of g-itemsets or g-rules). The key to these approaches is the efficient dynamic generation and pruning of the classification trees.

Further, several optimizations are proposed. **MFGI_class** 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, we proposed the **PHDB** optimization to batch-compute frequencies. To reduce the size of those scans, we proposed a transaction filtering optimization.

Experimental results showed that both **MFGI_class** and **EGR_class** are superior to their lattice-based counterparts. Further, these algorithms are shown to be significantly faster than existing algorithms for mining all frequent g-itemsets and strong g-rules, such as those based on BASIC.

As a complimentary result, we provide closed form lower and upper bounds on the number of possible g-itemsets and g-rules.

We believe these algorithms and results will allow the efficient discovery of more interesting patterns in transactional data with taxonomies.

## References

[1] Hipp J, Myka A, Wirth R, Güntzer U. A new algorithm for faster mining of generalized association rules. In *Proc. European Conference on Principles of Data Mining and Knowledge Discovery* (*PKDD*), Nantes, France, 1998, pp.74–82.

[2] Pramudiono I, Kitsuregawa M. FP-tax: Tree structure based generalized association rule mining. In *Proc. ACM/SIGMOD International Workshop on Research Issues on Data Mining and Knowledge Discovery* (*DMKD*), Paris, France, 2004, pp.60–63.

[3] Srikant R, Agrawal R. Mining generalized association rules. In *Proc. International Conference on Very Large Data Bases* (*VLDB*), Zurich, Switzerland, 1995, pp.407–419.

[4] Sriphaew K, Theeramunkong T. A new method for finding generalized frequent itemsets in generalized association rule mining. In *Proc. International Symposium on Computers and Communications* (*ISCC*), Taormina, Italy, 2002, pp.1040–1045.

[5] Sriphaew K, Theeramunkong T. Fast algorithms for mining generalized frequent patterns of generalized association rules. *IEICE Transactions on Information and Systems*, March 2004, E87-D(3).

[6] Sriphaew K, Theeramunkong T. Mining generalized closed frequent itemsets of generalized association rules. In *Proc. International Conference on Knowledge-Based Intelligent Information and Engineering Systems* (*KES*), Oxford, United Kingdom, 2003, pp.476–484.

[7] Bayardo Jr R J. Efficiently mining long patterns from databases. In *Proc. ACM/SIGMOD Annual Conference on Management of Data* (*SIGMOD*), Seattle, WA, 1998, pp.85–93.

[8] Agarwal R C, Aggarwal C C, Prasad V V V. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel Distributed Computing*, 2001, 61(3): 350–371.

[9] Han J, Pei J, Yin Y. Mining frequent patterns without candidate generation. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data* (*SIGMOD*), Dallas, TX, 2000, pp.1–12.

[10] Lin D I, Kedem Z M. Pincer-Search: An efficient algorithm for discovering the maximum frequent set. *IEEE Trans. Knowledge and Data Engineering* (*TKDE*), 2002, 14(3): 553–566.

[11] Pasquier N, Bastide Y, Taouil R, Lakhal L. Discovering frequent closed itemsets for association rules. In *Proc. International Conference on Database Theory* (*ICDT*), Jerusalem, Israel, 1999, pp.398–416.

[12] Pei J, Han J, Mao R. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *Proc. ACM/SIGMOD International Workshop on Research Issues on Data Mining and Knowledge Discovery* (*DMKD*), Dallas, TX, 2000, pp.21–30.

[13] Wang K, Tang L, Han J, Liu J. Top down FP-growth for association rule mining. In *Proc. Pacific-Asia Conference on Knowledge Discovery and Data Mining* (*PAKDD*), Taipei, Taiwan, 2002, pp.334–340.

[14] Agrawal R, Imielinski T, Swami A M. Mining association rules between sets of items in large databases. In *Proc. ACM/SIGMOD Annual Conference on Management of Data* (*SIGMOD*), Washington DC, 1993, pp.207–216.

[15] Agarwal R C, Aggarwal C C, Prasad V V V. Depth first generation of long patterns. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (*KDD*), Boston, MA, 2000, pp.108–118.

[16] Burdick D, Calimlim M, Gehrke J. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Proc. International Conference on Data Engineering* (*ICDE*), , Heidelberg, Germany, 2001, pp.443–452.

[17] Gouda K, Zaki M J. Efficiently mining maximal frequent itemsets. In *Proc. International Conference on Data Mining* (*ICDM*), San Jose, CA, 2001, pp.163–170.

[18] Xin D, Han J, Yan X, Cheng H. Mining compressed frequent-pattern sets. In *Proc. International Conference on Very Large Data Bases* (*VLDB*), Trondheim, Norway, 2005, pp.709–720.

[19] Yan X, Cheng H, Han J, Xin D. Summarizing itemset patterns: A profile-based approach. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (*KDD*), Chicago, IL, 2005, pp.314–323.

[20] Calders T, Goethals B. Depth-first non-derivable itemset mining. In *Proc. the SIAM International Conference on Data Mining* (*SDM*), Newport Beach, CA, 2005.

[21] Ke Y, Cheng J, Ng W. Mining quantitative correlated patterns using an information-theoretic approach. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (*KDD*), Philadelphia, PA, 2006, pp.227–236.

[22] Xiong H, Tan P N, Kumar V. Hyperclique pattern discovery. *Data Mining and Knowledge Discovery*, 2006, 13(2): 219–242.

[23] Ghoting A, Buehrer G, Parthasarathy S, Kim D, Nguyen A, Chen Y K, Dubey P. Cache-conscious frequent pattern mining on a modern processor. In *Proc. International Conference on Very Large Data Bases* (*VLDB*), Trondheim, Norway, 2005, pp.577–588.

[24] Han J, Fu Y. Mining multiple-level association rules in large databases. *IEEE Trans. Knowledge and Data Engineering* (*TKDE*), 1999, 11(5): 798–805.

[25] Huang Y F, Wu C M. Mining generalized association rules using pruning techniques. In *Proc. International Conference on Data Mining* (*ICDM*), Maebashi City, Japan, 2002, pp.227–234.

[26] Aggarwal C C, Yu P S. Online generation of association rules. In *Proc. International Conference on Data Engineering* (*ICDE*), Orlando, FL, 1998, pp.402–411.

[27] Zaki M J. Generating non-redundant association rules. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (*KDD*), Boston, MA, 2000, pp.34–43.

[28] Lui C L, Chung K F. Discovery of generalized association rules with multiple minimum supports. In *Proc. European Conference on Principles of Data Mining and Knowledge Discovery* (*PKDD*), Lyon, France, 2000, pp.510–515.

[29] Tseng M C, Lin W Y. Mining generalized association rules with multiple minimum supports. In *Proc. International Conference on Data Warehousing and Knowledge Discovery* (*DaWaK*), Munich, Germany, 2001, pp.11–20.

[30] Newman D J, Asuncion A. UCI machine learning repository. University of California, Irvine, 2007, http:mlearn.ics.uci.edu/MLRepository.html.

[31] Synthetic Data Generation Code for Associations and Sequential Patterns (IBM Almaden Research Center). http://www.almaden.ibm.com/software/quest/Resources/datasets/syndata.html.

[32] Kunkle D, Zhang D, Cooperman G. Efficient mining of max frequent patterns in a generalized environment. In *Proc. International Conference on Information and Knowledge Management* (*CIKM*), Arlington, VA, 2006, pp.810–811.

**Daniel Kunkle** is a Ph.D. candidate in the College of Computer and Information Science at Northeastern University. He received his B.S. degree in information technology in 2001 and his M.S. degree in computer science in 2003, both from the Rochester Institute of Technology. His research interests include combinatorial optimization, high performance computing, and adaptive systems.

**Donghui Zhang** received his Ph.D. degree in 2002 from the University of California – Riverside. Since then, he has been working as an assistant professor in the College of Computer & Information Science, Northeastern University. Professor Zhang's primary research area is databases. In particular, indexing and query optimization in spatial, temporal, and spatiotemporal databases. Many real application data have spatial and/or temporal dimensions. For instance, the locations of apartment buildings, cars, mobile-phone users which may or may not change over time. The concern is how to index such objects and how to efficiently compute the result of interesting queries. Professor Zhang received the NSF CAREER Award: Fast Query Support for Emerging Spatial Database Applications. He has written five book chapters and published over twenty peer-refereed research papers. He has served on the panels of two NSF programs, on the Program Committees of various international conferences including ICDE'07, SSTD'07, VLDB'05, ICDE'04 and EDBT'04, and as referee for over 10 journals such as TODS and VLDBJ.

**Gene Cooperman** received his B.S. degree from the University of Michigan in honors math and physics, and his Ph.D. degree in applied mathematics from Brown University. He is currently a professor of computer and information science at Northeastern University, Boston. His research interests include parallel and high performance computing, and algebraic computations. Dr. Cooperman is an associate editor of ACM Transactions on Mathematical Software (TOMS) and on the advisory board of the European Union SCIEnce project (Symbolic Computation in Europe). Dr. Cooperman currently serves on three program committees of technical conferences in computational and parallel algebra and network computing. He is also head of the Institute for Complex and Scientific Computing (http://www.icss.neu.edu) at Northeastern University. Dr. Cooperman has published over 70 refereed publications.