Minimum Description Length Principle: Generators are Preferable to Closed Patterns

Jinyan Li [1,*] **Haiquan Li** [1] **Limsoon Wong** [2]
[1] Institute for Infocomm Research, Singapore
[2] National University of Singapore, Singapore
[*] Email: jinyan@i2r.a-star.edu.sg

Jian Pei [3] **Guozhu Dong** [4]
[3] Simon Fraser University, Canada
[4] Wright State University, USA

Abstract

The generators and the unique closed pattern of an equivalence class of itemsets share a common set of transactions. The generators are the minimal ones among the equivalent itemsets, while the closed pattern is the maximum one. As a generator is usually smaller than the closed pattern in cardinality, by the Minimum Description Length Principle, the generator is preferable to the closed pattern in inductive inference and classification. To efficiently discover frequent generators from a large dataset, we develop a depth-first algorithm called Gr-growth. The idea is novel in contrast to traditional breadth-first bottom-up generator-mining algorithms. Our extensive performance study shows that Gr-growth is significantly faster (an order or even two orders of magnitudes when the support thresholds are low) than the existing generator mining algorithms. It can be also faster than the state-of-the-art frequent closed itemset mining algorithms such as FPclose and CLOSET+.

Introduction

A set of itemsets is said to form an *equivalence class* (Bastide *et al.* 2000) if they occur in the same set of transactions in a dataset. The maximum itemset (under set inclusion) in an equivalence class is called a *closed pattern*. The minimal itemsets of an equivalence class are called *generators* (Pasquier *et al.* 1999). An itemset that occurs in many transactions is said to be frequent (Agrawal, Imielinski, & Swami 1993). Frequent closed patterns (Pasquier *et al.* 1999) can form a concise and lossless representation of frequent itemsets. Thus they have been extensively studied (Pasquier *et al.* 1999; Zaki & Hsiao 2002; Pan *et al.* 2003; Wang, Han, & Pei 2003; Grahne & Zhu 2005; Uno, Kiyomi, & Arimura 2004). On the other hand, the minimal itemsets of an equivalence class are often shorter than the maximum one. Thus, by the Minimum Description Length Principle (MDL) (Rissanen 1978; Grunwald, Myung, & Pitt 2005), generators are preferable to closed patterns for model selection and classification.

However, existing algorithms (Boulicaut, Bykowski, & Rigotti 2003; Bastide *et al.* 2000; Luong 2002; Pasquier *et al.* 1999; Kryszkiewicz, Rybinski, & Gajek 2004) for mining frequent generators are slow. All of them adopt a breadth-first Apriori-join (level-wise bottom-up) approach to generate candidates. Such an approach generally requires a lot of memory, and also requires scanning the dataset many times. We propose here a novel algorithm called Gr-growth which uses a depth-first search strategy. Our extensive performance study shows that Gr-growth is usually 10-100 times faster than the existing generator mining algorithms. It is also faster than the state-of-the-art frequent closed itemset mining algorithms such as CLOSET+ (Wang, Han, & Pei 2003) and FPclose (Grahne & Zhu 2005).

The new data structure used by the Gr-growth algorithm is called Gr-tree, a classical trie structure similar to FP-tree used in the FP-growth algorithm (Han, Pei, & Yin 2000) for mining frequent itemsets. There are two critical differences between the two structures: (i) The header table of a Gr-tree (or a conditional Gr-tree) does not contain any *full-support items*, but that of an FP-tree always contains them; (ii) the header table of a conditional Gr-tree does not contain any frequent *pseudo-key items*, but that of an FP-tree always contains them. Here, a pseudo-key item is an item that is not a full-support item and whose union with the root of the tree is not a generator either. The manipulation of the trees is also different—the nodes in a single path Gr-tree is examined only linearly, but combinations of the nodes in a single path FP-tree are exhaustively enumerated. Thus a Gr-tree is very often smaller and has less computation than an FP-tree.

We elaborate in the next few sections why MDL favors generators, and how a depth-first search approach can produce all frequent generators from a dataset at a speed faster than the state-of-the-art algorithms for mining closed patterns.

Generators and Closed Patterns

A *dataset* is defined as $\mathcal{D} = \langle \mathcal{I}_\mathcal{D}, TDB_\mathcal{D} \rangle$, where $\mathcal{I}_\mathcal{D}$ is a non-empty finite set of *items* and $TDB_\mathcal{D} \subseteq 2^{\mathcal{I}_\mathcal{D}}$ a multiset of *transactions*. A subset $I \subseteq \mathcal{I}_\mathcal{D}$ is called an *itemset*. An itemset consisting of k items is called a k-itemset. The *support* of an itemset I in a dataset $\mathcal{D}$, denoted by $sup_\mathcal{D}(I)$, is the number of transactions in $TDB_\mathcal{D}$ that contain I. An itemset I is said to be *frequent* in a dataset $\mathcal{D}$ iff $sup_\mathcal{D}(I) \geq ms$ for a pre-specified threshold *ms*. For an itemset $I \subseteq \mathcal{I}_\mathcal{D}$, we define $f_\mathcal{D}(I) = \{T \in TDB_\mathcal{D} \mid I \subseteq T\}$; i.e., all transac-

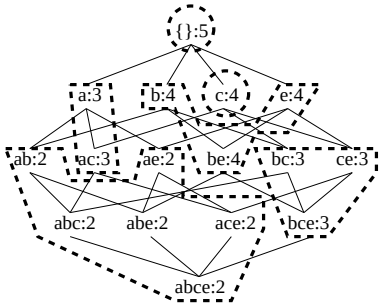
tions in the dataset containing itemset I. Hence $sup_{\mathcal{D}}(I) = |f_{\mathcal{D}}(I)|$. For a set of transactions $TDB' \subseteq TDB_{\mathcal{D}}$, we define $g_{\mathcal{D}}(TDB') = \{i \in \mathcal{I}_{\mathcal{D}} \mid \text{for all } T \in TDB', i \in T\}$; i.e., the set of items which are shared by all transactions in TDB'.

For an itemset I, $cl_{\mathcal{D}}(I) = g_{\mathcal{D}}(f_{\mathcal{D}}(I))$ is called the *closure* of I. The closure induces an *equivalence relation* $\sim_{\mathcal{D}}$ on $2^{\mathcal{I}_{\mathcal{D}}}$ by $I_1 \sim_{\mathcal{D}} I_2$ iff $cl_{\mathcal{D}}(I_1) = cl_{\mathcal{D}}(I_2)$. Thus the *equivalence class* $[I]_{\mathcal{D}}$ of an itemset I is defined as the set $\{A \subseteq \mathcal{I}_{\mathcal{D}} \mid cl_{\mathcal{D}}(A) = cl_{\mathcal{D}}(I)\}$. So, all itemsets in an equivalence class are contained in some common set of transactions. The one and only one maximal element in an equivalence class $[I]_{\mathcal{D}}$, namely $cl_{\mathcal{D}}(I)$, is called the *closed pattern* of this equivalence class. The minimal ones are called *generators*.

Example 1 *Consider the following TDB:*

Transaction-id	*Items*
T_1	a, c, d
T_2	b, c, e
T_3	a, b, c, e, f
T_4	b, e
T_5	a, b, c, e

Suppose the minimum support threshold is 2*. Then, there are in total* 16 *frequent itemsets, including the empty set* $\emptyset$ *which is trivially frequent. The frequent itemsets as well as their set inclusion relation are shown in the following figure. These frequent itemsets can be divided into* 6 *equivalence*



classes, as bounded by the dash lines in this figure. In each class, the itemset at the bottom is the closed itemset, and the itemsets at the top are the generators. In particular, the itemset $abce$ *is a closed pattern, and* $f_{\mathcal{D}}(abce)$ *consists of two transactions:* $T_3 = \{a, b, c, e, f\}$ *and* $T_5 = \{a, b, c, e\}$*. The equivalence class* $[abce]_{\mathcal{D}}$ *is* $\{abce, abc, abe, ace, ab, ae\}$*. The generators of* $[abce]_{\mathcal{D}}$ *are* ab *and* ae*. Some closed patterns are also generators, for example, the itemset* c*.*

MDL Favors Generators

The Minimum Description Length Principle (MDL) was proposed by (Rissanen 1978), developed by (Li & Vitanyi 1997), and recently surveyed by (Grunwald, Myung, & Pitt 2005). This principle provides a generic solution to the model selection problem. MDL has a sound statistical foundation rooted in the well-known Bayesian inference and Kolmogorov complexity.

A crude two-part version of MDL (Grunwald, Myung, & Pitt 2005) is as follows: Let $\mathcal{H} = \{H_1, H_2, \cdots, H_n\}$ be a set of hypothesis learned from a dataset D. The best hypothesis $H \in \mathcal{H}$ to explain D is the one which minimizes the sum $L(H, D) = L(H) + L(D|H)$, where

- $L(H)$ is the length, in bits, of the description of hypothesis H; and
- $L(D|H)$ is the length, in bits, of the description of the data when encoded with the help of hypothesis H.

We bring this principle into the context of generators and closed patterns in a similar way to (Gao, Li, & Vitanyi 2000), where robot arm learning and hand-written character recognition problems are discussed. Let Ec be an equivalence class of some dataset $\mathcal{D}$, C the closed pattern of Ec, and G a generator of Ec. Let $D_G^C = f_{\mathcal{D}}(C) = f_{\mathcal{D}}(G)$. Then C and G are two hypothesis describing the data D_G^C. For C, the description length $L(C, D_G^C) = L(C) + L(D_G^C|C)$. For G, the description length $L(G, D_G^C) = L(G) + L(D_G^C|G)$. The closed pattern C and the generator G occur in the same data D_G^C. So, $L(D_G^C|C) = L(D_G^C|G)$. Therefore, if $L(C) > L(G)$, then $L(C, D_G^C) > L(G, D_G^C)$. This is often true because the cardinality of C is often larger than that of G. So, by MDL, the generator G is preferable to the closed pattern C for describing its transaction set D_G^C.

This preference is particularly obvious in classification problems. For an application where only two classes of transactions are involved, suppose transactions in D_G^C all have the same class label, say, *positive* class. Also assume that the closed pattern C has n items $a_1, a_2, \cdots, a_n$ ($n > 2$), and the generator G has only 2 items a_1 and a_2. Then, we can get two rules:

- One derived from C: If a transaction contains a_1 *and* a_2 *and* $\cdots$ *and* a_n, then it is positive.
- The other derived from G: If a transaction contains a_1 *and* a_2, then it is positive.

Note that the two rules are both satisfied by all transactions in D_G^C but no other transactions in $\mathcal{D}$.

The second rule should be more predictive on independent test data than the first one, because a true test sample is more likely to satisfy the two items a_1 and a_2 than to satisfy the n items contained in C. So, when some noise is present in the test data, the second rule can better tolerate the noise errors than the first rule.

Gr-growth: Mining Frequent Generators

We next present some properties of generators. Then we introduce our depth-first Gr-growth algorithm for mining frequent generators with the help of a new data structure called *frequent generator tree*, or frequent Gr-tree in short. A Gr-tree is a typical trie structure, storing all relevant information of a dataset for mining frequent generators.

Proposition 1 *Let* $\mathcal{D} = \langle \mathcal{I}_{\mathcal{D}}, TDB_{\mathcal{D}} \rangle$ *be a dataset. Let* $\mathcal{G}_{\mathcal{D}}$ *be the set of generators. Let* A *be an itemset. Then* $A \in \mathcal{G}_{\mathcal{D}}$ *iff* $sup_{\mathcal{D}}(A) < sup_{\mathcal{D}}(B)$ *for every proper subset* B *of* A*.*

Proof: *We first consider the "only if" direction. Suppose* $A \in \mathcal{G}_{\mathcal{D}}$ *and* B *is a proper subset of* A*. Then we have*

$f_{\mathcal{D}}(A) \subseteq f_{\mathcal{D}}(B)$. *As* $A \in \mathcal{G}_{\mathcal{D}}$, *we have* B *is in a different equivalence class from that of* A. *Therefore* $f_{\mathcal{D}}(A) \subset f_{\mathcal{D}}(B)$. *So* $sup_{\mathcal{D}}(A) < sup_{\mathcal{D}}(B)$.

We next prove the "if" part by contradiction. Assume A *is not a generator in an equivalence class. Then there exists an itemset* B *in this equivalence class that is a proper subset of* A. *So,* $sup_{\mathcal{D}}(B) = sup_{\mathcal{D}}(A)$. *But, this is a contradiction because* $sup_{\mathcal{D}}(A) < sup_{\mathcal{D}}(B)$ *for every proper subset* B *of* A. *So,* A *must be a generator.* □

Proposition 2 (Apriori property of generators) *Let* $\mathcal{G}_{\mathcal{D}}$ *be the set of generators of a dataset* $\mathcal{D}$. *Then for all* $A \in \mathcal{G}_{\mathcal{D}}$, *every proper subset* B *of* A *is a generator, i.e.* $B \in \mathcal{G}_{\mathcal{D}}$; *and for all* $X \notin \mathcal{G}_{\mathcal{D}}$, *every proper superset* Y *of* X *is not a generator, i.e.* $Y \notin \mathcal{G}_{\mathcal{D}}$.

Proof: *We prove by contradiction. Suppose* A *is a generator of* $\mathcal{D}$ *and* $B \subset A$ *is not. Denote* $C = cl_{\mathcal{D}}(B)$. *Then, there is a generator* B' *of the equivalence class of* C *such that* $B' \subset B$. *Let* $A = B \cup V$ *such that* $B \cap V = \emptyset$. *As* B' *and* B *are in the same equivalence class, we have* $sup_{\mathcal{D}}(B' \cup V) = sup_{\mathcal{D}}(B \cup V) = sup_{\mathcal{D}}(A)$. *That is,* A *has a proper subset* $B' \cup V$ *with the same support as itself. However,* A *is a generator. By Proposition 1, all its proper subsets must have a larger support than that of* A. *This is a contradiction. Therefore,* B *is a generator.*

The above proof implies that for all $X \notin \mathcal{G}_{\mathcal{D}}$, *every proper superset* Y *of* X *is not a generator, i.e.* $Y \notin \mathcal{G}_{\mathcal{D}}$. □

Corollary 1 *If a frequent 1-itemset's support is less than the total number of transactions of a dataset, then this 1-itemset is a frequent generator.*

The items in frequent 1-itemset generators will be referred to as frequent key items. It is easy to see that a *full-support item* (an item occurring in every transaction of a *TDB*) is not a key item of *TDB*.

Frequent Gr-tree and Conditional Gr-tree

Let $\mathcal{D} = \langle \mathcal{I}_{\mathcal{D}}, TDB_{\mathcal{D}} \rangle$ be a dataset. Let a support threshold ms be given. A frequent Gr-tree of $\mathcal{D}$, denoted by Gr-tree$|^{\emptyset}_{\mathcal{D},ms}$, is constructed by the algorithm shown in Figure 1.

A *frequent conditional* Gr-tree is constructed from a *conditional TDB*:

Definition 1 (Conditional *TDB*) *Let* $\mathcal{D} = \langle \mathcal{I}_{\mathcal{D}}, TDB_{\mathcal{D}} \rangle$ *be a dataset. Let a support threshold* ms *be given. Let* $Tree$ *be a frequent* Gr-tree *of* $TDB_{\mathcal{D}}$ *with respect to* ms. *Let* $a_1, a_2, \cdots, a_n$ *be items in the header table of* $Tree$. *For* a_i $(i = 1, \cdots, n)$, *we define* a_i*'s conditional TDB, denoted by* $TDB|^{\{a_i\}}_{\mathcal{D},ms}$, *as the set of path segments exclusively between the root and* a_i *for all paths in* $Tree$ *containing* a_i.

Each path segment is equivalent to an itemset, with support equal to that of a_i in that path. So we can say that $TDB|^{\{a_i\}}_{\mathcal{D},ms}$ is a set of transactions.

Definition 2 (Full-support items) *An item* e *is called a full-support item of* $TDB|^{\{a_i\}}_{\mathcal{D},ms}$, *if its support in* $TDB|^{\{a_i\}}_{\mathcal{D},ms}$ *is equal to* a_i*'s support.*

Algorithm 1 (Frequent Gr-tree construction)
Input: A dataset $\mathcal{D} = \langle \mathcal{I}_{\mathcal{D}}, TDB_{\mathcal{D}} \rangle$ and a ms.
Output: A frequent Gr-tree$|^{\emptyset}_{\mathcal{D},ms}$.
Method: The construction consists of two steps:

1. Collect the set of frequent key items – Cf. Corollary 1. Create a header table to store them in a support descending order L.
2. Create the root of the tree T, and label it as $\emptyset$. For each transaction $Trans$ in $TDB_{\mathcal{D}}$ do the following.
   Remove those items from $Trans$ that are infrequent or that are full-support items. Sort the remaining items according to the order of L. Let the resulting list be $[p|P]$, where p is the first element and P is the remaining list. Call $insert\_tree([p|P], T)$.
   The function $insert\_tree([p|P], T)$ (Han, Pei, & Yin 2000) is performed as follows: If T has a child N such that N.item-name = p.item-name, then increment N's count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its node-link be linked to the nodes with the same item-name via a node-link structure. If P is nonempty, recursively call $insert\_tree(P, N)$.

Figure 1: Steps of constructing a frequent Gr-tree.

Definition 3 (Conditional Gr-tree) *Following the notations in Definition 1, we define* a_i*'s conditional frequent* Gr-tree, *denoted by* Gr-tree$|^{\{a_i\}}_{\mathcal{D},ms}$, *as a tree structure satisfying the following conditions:*

1. *It consists of one root labeled as "*$\{a_i\}$*", a set of item prefix subtrees as the children of the root, and a header table storing a list of items satisfying:*
   (a) *They are frequent but not full-support items in* $TDB|^{\{a_i\}}_{\mathcal{D},ms}$; *and*
   (b) *They are not pseudo-key items, that is, the union of any* e *from them and the root (i.e.,* $\{e, a_i\}$*) is a generator of* $\mathcal{D}$.
2. *The fields of each node of the tree and each entry in the header table have exactly the same meaning as a normal* Gr-tree.

Based on Gr-tree$|^{\{a_i\}}_{\mathcal{D},ms}$, for an item a_j in the header of Gr-tree$|^{\{a_i\}}_{\mathcal{D},ms}$, similarly we can define a_j's conditional TDB and a_j's conditional Gr-tree, which are denoted by $TDB|^{\{a_i,a_j\}}_{\mathcal{D},ms}$ and Gr-tree$|^{\{a_i,a_j\}}_{\mathcal{D},ms}$ respectively. Usually, we denote a conditional TDB as $TDB|^{\alpha}_{\mathcal{D},ms}$, and a conditional Gr-tree as Gr-tree$|^{\alpha}_{\mathcal{D},ms}$, for some itemset α. We also write $TDB|_{\alpha}$ and Gr-tree$|_{\alpha}$ if $\mathcal{D}$ and ms are understood.

Determining whether the union of a frequent item e with the root α is a generator—as required by Definition 3—costs most of the time in constructing the header table of the conditional tree. This is also a unique feature of conditional Gr-tree—a conditional FP-tree (Han, Pei, & Yin 2000) does not require this. The cost of such a determination operation can be achieved in constant time by using a hash-table consisting of generators already found. Also observe that as many frequent items e may be not in the header table, a frequent conditional Gr-tree is very often smaller than a conditional FP-tree.

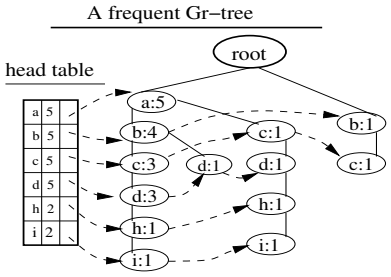
Example 2 *Let* $\mathcal{D} = \langle \mathcal{I}_{\mathcal{D}}, TDB_{\mathcal{D}} \rangle$ *be a dataset where* $\mathcal{I}_{\mathcal{D}} = \{a, b, c, d, e, f, g, h, i\}$ *and* $TDB_{\mathcal{D}}$ *consists of 6 transactions*

Table 1: A dataset for our running example.

Transactions	Transactions after item removal and re-ordering
$\{a, b, c, d, e, g\}$	$\{a, b, c, d\}$
$\{a, b, c, d, e, f\}$	$\{a, b, c, d\}$
$\{a, b, c, d, e, h, i\}$	$\{a, b, c, d, h, i\}$
$\{a, b, d, e\}$	$\{a, b, d\}$
$\{d, c, a, e, h, i\}$	$\{a, c, d, h, i\}$
$\{e, c, b\}$	$\{b, c\}$

as shown in the left column of Table 1. Here, we also set $ms = 2$.

The support information of the 9 items is a*:5,* b*:5,* c*:5,* d*:5,* e*:6,* g*:1,* f*:1,* h*:2,* i*:2. However, only 6 items as sorted in the list* $L = \langle a{:}5, b{:}5, c{:}5, d{:}5, h{:}2, i{:}2\rangle$ *are in the header table of* Gr-tree$|^{\emptyset}_{\mathcal{D},ms}$. *Item* e *is not there, since it is a full-support item. By Proposition 1 and Proposition 2, itemset* e *itself and all its supersets cannot be a generator. All frequent generators of this dataset can be formed using only the items in the header table. So, we can remove unnecessary items from the 6 transactions. The reduced transactions are shown in the right column of Table 1, which are then scanned to construct the tree using* Algorithm 1. *The whole tree, after constructing node-links from the header table to the first node of the tree carrying the item-name, is depicted in the following figure.*



Gr-growth: Discovering All Frequent Generators

Given a dataset $\mathcal{D} = \langle \mathcal{I}_{\mathcal{D}}, TDB_{\mathcal{D}}\rangle$ and a threshold ms. Let $L = \langle a_1, a_2, a_3, \cdots, a_n\rangle$ be the list of frequent key items in a support descending order of this dataset. These items are also precisely the items in the header table of the frequent Gr-tree$|^{\emptyset}_{\mathcal{D},ms}$. Suppose this frequent Gr-tree is not a single-path tree. Observe that, in addition to the default generator $\emptyset$, other frequent generators of this $\mathcal{D}$ can be divided into n non-overlapping groups:

- those containing item a_n;
- those containing item a_{n-1} but not a_n;
- $\ldots$;
- those containing item a_k but no item in $\{a_{k+1}, \cdots, a_n\}$; and so on until $k = 1$.

We denote the k-th group of generators as $group_k = \{G \mid G$ is a generator, $a_k \in G$, but no item in $\{a_{k+1}, a_{k+2}, \ldots, a_n\} \in G\}$, for $k = 1, \ldots, n$.

Algorithm 2 (Gr-growth—Mining frequent generators)
Input: Gr-tree$|^{\emptyset}_{\mathcal{D},ms}$ constructed by Algorithm 1 from $\mathcal{D}$.
Output: The complete set of frequent generators.
Method: Call Gr-growth(Gr-tree$|^{\emptyset}_{\mathcal{D},ms}, null$).

Gr-growth($Tree, \alpha$)
1: generate pattern α with support $= \alpha.support$;
2: **if** $Tree = \emptyset$ **then**
3: **return** ;
4: **end if**
5: **if** $Tree$ contains a single path P **then**
6: **for** each node e in the path P **do**
7: generate pattern $\{e.item\} \cup \alpha$ with support $= e.support$;
8: **end for**
9: **else**
10: let $a_1, a_2, \cdots, a_n$ be items in the header of $Tree$ in support-descending order;
11: **for** each a_i, i from $1, \cdots,$ to n **do**
12: $\beta = \{a_i\} \cup \alpha$ with support $= a_i.support$;
13: construct $TDB|^{\beta}_{\mathcal{D},ms}$ and Gr-tree$|^{\beta}_{\mathcal{D},ms}$;
14: call Gr-growth(Gr-tree$|^{\beta}_{\mathcal{D},ms}, \beta$);
15: **end for**
16: **end if**

Figure 2: The Gr-Growth algorithm.

We can obtain sufficient transaction information for mining $group_k$ from $TDB|^{\{a_k\}}_{\mathcal{D},ms}$. By definition, the construction of $TDB|^{\{a_k\}}_{\mathcal{D},ms}$ can be derived from the frequent Gr-tree$|^{\emptyset}_{\mathcal{D},ms}$ by a traversal of the a_k's node-links starting from a_k's in the header table of Gr-tree$|^{\emptyset}_{\mathcal{D},ms}$. Then we construct frequent Gr-tree$|^{\{a_k\}}_{\mathcal{D},ms}$. If Gr-tree$|^{\{a_k\}}_{\mathcal{D},ms}$ is a single path tree, then the mining of $group_k$ terminates; and the generators are $\{a_k\}$ in addition to those that are union of $\{a_k\}$ with every item in the header table of Gr-tree$|^{\{a_k\}}_{\mathcal{D},ms}$. If Gr-tree$|^{\{a_k\}}_{\mathcal{D},ms}$ is empty, then $group_k = \{\{a_k\}\}$; and the mining also terminates. Otherwise, if Gr-tree$|^{\{a_k\}}_{\mathcal{D},ms}$ is a multiple-branch tree, then we do a recursive repartition by dividing $group_k$ into subsets according to the items stored in the header table of Gr-tree$|^{\{a_k\}}_{\mathcal{D},ms}$. The details of this recursion are as follows. Suppose the items in the header table of Gr-tree$|^{\{a_k\}}_{\mathcal{D},ms}$ are the list $L' = \langle a'_1, a'_2, a'_3, \cdots, a'_{n'}\rangle$. Then, in addition to the generator $\{a_k\}$, we divide $group_k$ into n' subsets: those containing item $a'_{n'}$; those containing item $a'_{n'-1}$ but not $a'_{n'}$; $\ldots$; those containing item a'_i but none in $\{a'_{i+1}, \cdots, a'_{n'}\}$; and so on until $i = 1$.

Similarly, we can deal with other groups of generators. The whole recursive process eventually leads to a non-overlapping partition of all the generators—some divisions contain only one pattern each, and the other divisions each corresponds to a single-path conditional Gr-tree. The algorithm is shown in Figure 2.

We define an enumeration ordering on the nodes of Gr-tree to prove the soundness and the completeness of Gr-growth.

Definition 4 (Set-enumeration property) *Let* T *be the* Gr-tree *constructed from a dataset* $\mathcal{D}$ *with respect to a sup-*

port threshold ms. Let the header items be a_0, a_1, ..., a_n in descending order of support. For any path P formed by a combination of a_0, a_1, ..., a_n, let $\kappa_{\mathcal{D}}(P) = \sum_{a_i \in P} 2^i$. For any two paths P and Q in the Gr-tree*, we write $P \leq^{\kappa_{\mathcal{D}}} Q$ if $\kappa_{\mathcal{D}}(Q) \leq \kappa_{\mathcal{D}}(P)$. Note that $P \subseteq Q$ implies $Q \leq^{\kappa_{\mathcal{D}}} P$.*

The recursion in Gr-growth is arranged to produce generators in the reverse of the set-enumeration order. That is, for generators α and β, Gr-growth produces β after α iff $\beta \leq^{\kappa_{\mathcal{D}}} \alpha$. This gives an important optimization for line 13. Recall that in constructing the conditional Gr-tree$|^{\beta}_{\mathcal{D},ms}$, we are only allowed to put those items e into the header table of Gr-tree$|^{\beta}_{\mathcal{D},ms}$, provided $\{e\} \cup \beta$ is a frequent generator. To check whether $\{e\} \cup \beta$ is a generator, we need to check if each of its immediate subsets β' is a generator. Since $(\{e\} \cup \beta) \leq^{\kappa_{\mathcal{D}}} \beta'$, β' is produced before $\{e\} \cup \beta$. This way, we can store a β' that is a generator into a hash table. Then when it comes to checking whether the immediate subsets of $\{e\} \cup \beta$ is a generator, we can easily look that up from the hash table.

Theorem 1 (Correctness of Gr-growth) Gr-growth *is sound and complete. That is, given a dataset $\mathcal{D}$ and a support threshold ms, it produces all generators and only the generators of $\mathcal{D}$ with respect to ms.*

Limited by space, the proof of this theorem is omitted.

We use the frequent Gr-tree$|_{\emptyset}$ in Example 2 to demonstrate how Gr-growth proceeds, where the sorted list of the frequent key items $L = \langle$a:5, b:5, c:5, d:5, h:2, i:2$\rangle$. The Gr-growth algorithm starts with the first key item a:5 after outputting the generator $\emptyset$, the root-node of Gr-tree$|_{\emptyset}$. Generators containing a but not b, c, d, h or i are very limited. They are $a:5$ only. This is because $TDB|_{a:5}$ is empty.

Then Gr-growth moves to deal with the second key item b:5. That is to mine generators containing b but not c, d, h or i. The construction of $TDB|_{b:5}$ is easy; it is $\{\{a:4\}, \emptyset\}$. Then we construct Gr-tree$|_{b:5}$. At this time, we get generator b:5, the root node of the conditional tree. As Gr-tree$|_{b:5}$ is a single path tree, we can get another generator ba:4 for this generator group by concatenating b to a:4.

Then Gr-growth moves to handle the third key item c:5. For mining generators containing c but not d, h or i, sufficient transaction information for this subset of generators come from three path segments of Gr-tree: $\langle a$:5, b:4$\rangle$, $\langle a$:5$\rangle$, and $\langle b$:1$\rangle$. Therefore $TDB|_{c:5}$ is $\{ab$:3, a:1, b: 1$\}$. The support information of the items in $TDB|_{c:5}$ are: a:4 and b:4. Then the list of frequent items in the header table of Gr-tree$|_{c:5}$ is $\langle a$:4, b:4$\rangle$. At this moment, we get the first generator c:5 for this generator group. As Gr-tree$|_{c:5}$ is not a single path tree, we apply the same divide-and-conquer searching strategy again to find two subsets of frequent generators of $TDB|_{c:5}$: those containing a but not b, and those containing b. The former consists of only ca:4 as Gr-tree$|_{c:5,a:4}$ is an empty tree. The latter consists of cb:4 and cba:3. Similarly, Gr-growth handles the remaining three items d:5, h:2, and i:2 in the header table to finish the mining.

In summary, all frequent generators for the dataset in Table 1, in the order of the output by Gr-growth, are $\emptyset$:6, a:5, b:5, ba:4 c:5, ca:4, cb:4, cba:3, d:5, db:4, dc:4, dcb:3, h:2, and i:2.

Performance Study

In this section, we report the performance of Gr-growth in comparison to the performance of existing generator mining algorithms and two closed pattern mining algorithms CLOSET+ (Wang, Han, & Pei 2003) and FPclose (Grahne & Zhu 2005). The experiments are conducted on four benchmark datasets from the Frequent Itemset Mining Implementations Repository (`http://fimi.cs.helsinki.fi/`). The four datasets are: Mushroom (8124 transactions and 119 items), Connect-4 (67557 transactions and 129 items), Chess (3196 transactions and 75 items), and T40I10D100K (100000 transactions and 942 items).

The implementation of Gr-growth is based on the source codes available from `http://www.cs.concordia.ca/db/dbdm/dm.html`, written in C++ and compiled by Visual C++ (v6.0) and executed on a PC with a Pentium(R) 4 CPU, 2.4GHz, and 512MB of RAM. The executable code of CLOSET+ and FPclose are from their authors. To implement the breadth-first search strategy adopted by all existing algorithms for mining frequent generators, we used the source codes for the implementation of Apriori by Borgelt (2003).

The performance results are depicted in Figure 3. We can see that Gr-growth is consistently faster than the traditional generator mining algorithm. For some datasets, especially when the minimum support threshold is low, the speed-up by Gr-growth is at least an order of magnitude or even 2 orders sometimes. For example, in the case of $ms = 20\%$ on the connect-4 dataset, the speed-up is around 190.0 times. Other similar cases can be also observed from the connect-4 and chess datasets. All these results clearly indicate that our depth-first searching strategy is indeed efficient for mining frequent generators with significant speed-up over the traditional algorithms.

Gr-growth is usually 2-4 times faster than CLOSET+; the speed-up can reach up to 10 times for cases such as $ms = 30\%$ on chess and $ms = 0.5\%$ on T40I10D100K. Gr-growth is also faster than FPclose in general, but the speed-up is sometimes significant, sometimes slight, and sometimes the two algorithms are comparable. The reason that these two closed pattern mining algorithms are slower than Gr-growth is mainly because they use an additional tree structure to store candidate itemsets recommended by the FP-tree, and then to filter out those candidates that are not closed patterns. Gr-growth does not require such an extra, large, filtering tree structure.

The length difference between the closed pattern and the shortest generator of an equivalence class is interesting. For the mushroom dataset at the minimum support level of 20%, we ranked the top 100 equivalence classes in terms of the length difference. The difference of the top-ranked equivalence class is 13 where the length of the closed pattern is 14, and that of the shortest generator is 1. There are many other similar equivalence classes. Obviously, by MDL, those generators are much more preferred than the closed patterns.

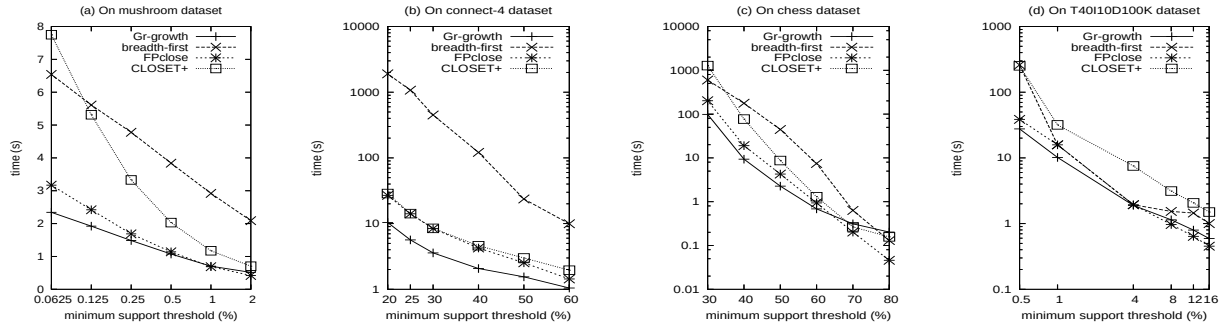


Figure 3: For mining frequent generators, our Gr-growth algorithm is significantly faster than the traditional breadth-first search strategy on the four benchmark datasets. Gr-growth is also faster than closed pattern mining algorithms.

Conclusions

We have proposed a new algorithm Gr-growth for mining frequent generators from a dataset. The success of Gr-growth is mainly attributed to the depth-first search strategy and the compact trie structure Gr-tree. With these two ideas, we have accelerated the speed of mining generators significantly, by at least an order of magnitude when the support threshold is low. Its speed is also faster than that of closed pattern mining algorithms FPclose and CLOSET+.

Based on MDL, we have demonstrated that generators are preferable to closed patterns in particular in rule induction and classification. Now that mining generators is faster than mining closed patterns, as a future work, we will study how generators are used for classification problems. We will also study in what situations using generators is significantly more reliable than using closed patterns in solving real-life prediction problems.

References

Agrawal, R.; Imielinski, T.; and Swami, A. 1993. Mining association rules between sets of items in large databases. In *Proceedings of 1993 ACM-SIGMOD International Conference on Management of Data*, 207–216. Washington, D.C.: ACM Press.

Bastide, Y.; Taouil, R.; Pasquier, N.; Stumme, G.; and Lakhal, L. 2000. Mining frequent patterns with counting inference. *SIGKDD Explorations* 2(2):66–75.

Borgelt, C. 2003. Efficient implementation of apriori and eclat. In *Proceedings of FIMI'03: Workshop on Frequent Itemset Mining Implementations*.

Boulicaut, J.-F.; Bykowski, A.; and Rigotti, C. 2003. Free-sets: A condensed representation of boolean data for the approximation of frequency queries. *Data Mining and Knowledge Discovery* 7(1):5–22.

Gao, Q.; Li, M.; and Vitanyi, P. 2000. Applying mdl to learning best model granularity. *Artificial Intelligence* 121:1–29.

Grahne, G., and Zhu, J. 2005. Fast algorithms for frequent itemset mining using fp-trees. *IEEE Transactions on Knowledge and Data Engineering* 17(10):1347–1362.

Grunwald, P.; Myung, I. J.; and Pitt, M. 2005. *Advances in Minimum Description Length: Theory and Applications*. MIT Press.

Han, J.; Pei, J.; and Yin, Y. 2000. Mining frequent patterns without candidates generation. In *Proceedings of 2000 ACM-SIGMOD International Conference on Management of Data*, 1–12. ACM Press.

Kryszkiewicz, M.; Rybinski, H.; and Gajek, M. 2004. Dataless transitions between concise representations of frequent patterns. *Journal of Intelligent Information Systems* 22(1):41–70.

Li, M., and Vitanyi, P. 1997. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag.

Luong, V. P. 2002. The closed keys base of frequent itemsets. In Kambayashi, Y.; Winiwarter, W.; and Arikawa, M., eds., *Proceedings of 4th International Conference on Data Warehousing and Knowledge Discovery*, 181–190.

Pan, F.; Cong, G.; Tung, A. K. H.; Yang, J.; and Zaki, M. J. 2003. CARPENTER: Finding closed patterns in long biological datasets. In *Proceedings of 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 637–642.

Pasquier, N.; Bastide, Y.; Taouil, R.; and Lakhal, L. 1999. Discovering frequent closed itemsets for association rules. In *Proceedings of 7th International Conference on Database Theory (ICDT)*, 398–416.

Rissanen, J. 1978. Modeling by shortest data description. *Automatica* 14:465–471.

Uno, T.; Kiyomi, M.; and Arimura, H. 2004. LCM ver.2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *IEEE ICDM'04 Workshop FIMI'04 (International Conference on Data Mining, Frequent Itemset Mining Implementations)*.

Wang, J.; Han, J.; and Pei, J. 2003. CLOSET+: Searching for the best strategies for mining frequent closed itemsets. In *Proceedings of 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'03), Washington, DC, USA*, 236–245.

Zaki, M. J., and Hsiao, C.-J. 2002. CHARM: An efficient algorithm for closed itemset mining. In *Proceedings of 2nd SIAM International Conference on Data Mining*.