An Approach for Mining Concurrently Closed Itemsets and Generators

Chapter · January 2013

DOI: 10.1007/978-3-319-00293-4_27

3 authors:

Anh Tran
University of Dalat
11 PUBLICATIONS **65** CITATIONS
SEE PROFILE

Tin C Truong
University of Dalat
21 PUBLICATIONS **117** CITATIONS
SEE PROFILE

Bac Le
Ho Chi Minh City University of Science
62 PUBLICATIONS **300** CITATIONS
SEE PROFILE

An Approach for Mining Concurrently Closed Itemsets and Generators

Anh Tran[1], Tin Truong[1], and Bac Le[2]

[1]Department of Mathematics and Computer Science, University of Dalat, Dalat, Vietnam
anhtn@dlu.edu.vn, tintc@dlu.edu.vn
[2]University of Natural Science Ho Chi Minh, Ho Chi Minh, Vietnam
lhbac@fit.hcmus.edu.vn

Abstract. Closed itemsets and their generators play an important role in frequent itemset and association rule mining since they lead to a lossless representation of all frequent itemsets. The previous approaches discover either frequent closed itemsets or generators separately. Due to *their properties and relationship*, the paper proposes *GENCLOSE* that *mines them concurrently*. In a level-wise search, it enumerates the generators using *a necessary and sufficient condition for producing (i+1)-item generators from i-item ones*. The condition is designed based on object-sets which can be implemented efficiently using diff-sets, is very convenience and *is reliably proved*. Along that process, *pre-closed itemsets are gradually extended using three proposed expanded operators*. Also, we prove that they bring us *to expected closed itemsets*. Experiments on many benchmark datasets confirm the efficiency of *GENCLOSE*.

Keywords. Frequent closed itemsets, generators, concept lattice, data mining.

Introduction

Association rule mining [1] from transaction datasets was popularized particularly and is one of the essential and fundamental techniques in data mining. The task is to find out the association rules that satisfy the pre-defined minimum support and confidence from a given dataset. As usual, it includes two phases of (1) to extract all frequent itemsets of which the occurrences exceed the minimum support, and (2) to generate association rules for the given minimum confidence from those itemsets. If we know all frequent itemsets and their supports, the association rule generation is straightforward. Hence, most of the researchers concentrate on the studies of finding a solution for mining frequent itemsets. Many algorithms have been proposed in order to do that mining task such as *Apriori* [2], *D-Eclat* [14]. Those algorithms all show good performance with sparse datasets having short itemsets such as market data. For dense ones where contain many long frequent itemsets, the class of frequent itemsets can grow to be unwieldy [6]. Thus, discovering frequent itemsets usually gets much duplication, wastes much time and is then infeasible. Mining only maximal frequent itemsets is a solution to overcome that disadvantage. There are many proposed algorithms for mining them [6-7]. An itemset is maximal frequent if none of its immediate super-

N.T. Nguyen, T.Van Do and H.A.Le Thi (Eds.): ICCSAMA 2013, SCI 479, pp. 355-366
DOI: 10.1007/978-3-319-00293-4_27

sets is frequent. The number of maximal itemsets is much smaller than the one of all frequent itemsets [16]. From them, we can generate all sub frequent itemsets. However, since frequent itemsets can come from different maximal ones, we spend much time for finding them again. Further, it is not easy to determine their supports exactly. Hence, maximal itemsets are not suitable for rule generation within the itemsets.

Mining frequent closed itemsets and generators has been received the attention of many researchers [8-10, 12, 13, 15-16] for their importance in the mining of frequent itemsets as well association rules. An itemset is closed if it is identical to its closure [17]. A generator of an itemset is its minimal proper subset having the same closure with it. Some studies [10, 11, 15] concerned only the generators of closed itemset as a mean to achieve the purpose of mining either closed itemsets or association rules. They are also called minimal generators [9, 15] or free-sets [8]. Dong et al. in [9] concentrated on the mining of non-redundant generators, but, it exceeds the scope of this paper. Which is the role of frequent closed itemsets and generators in the frequent itemset and association rule mining? First, one can see that a frequent itemset contains its generators and is contained in its closure. They all have the same closure, share the same set of transactions. Hence, *frequent closed itemsets and generators lead to a lossles representation of all frequent itemsets* in the sense that we can determine not only the support of any itemset [8, 15] but also a class of frequent itemsets having the same closure (so the same support) [4, 12]. Since their cardinality is typically orders of magnitude much fewer than all frequent itemsets [16], discovering them can be of a great help to purge many redundant itemsets. Therefore, they play *an important role in* many studies of *rule generation* such as [3, 5, 11, 15]. Pasquier et al. [11] used the minimal rules to generate a condensed representation for association rules. Zaki in [15] proposed a framework for reduction of association rule set which is based on the non-redundant rules. Two those rule kinds are obtained from frequent closed itemsets and the generators. Also based on them, we showed some structures of association rule set [3]. For mining frequent itemsets with constraint, the task of extracting frequent closed itemsets and generators restricted on contraint from the original ones is essential [4].

The problem of mining frequent closed itemsets and generators is stated as follows: *Given a transaction dataset and a minsup threshold, the task is to find all frequent closed itemsets together their generators*.

Related work. *There have been several algorithms* proposed for *mining closed itemsets*. They can be divided into three approaches, namely *generate-and-test*, *divide-and-conquer* and *hybrid*. *Close* [10] is in the first which executes a level-wise progress. In each step, it does two phases. The first is to generate the candidates for generators by joining generators found in previous step. If the support of a candidate equals to the one of any its subset, it could not be a generator. The second computes the closures of generators, but unfortunately, it is very expensive since there are a big number of transaction intersection operations. The second approach uses divide-and-conquer strategy to search over tree structures, e.g., *Closet+*[13]. The hybrid one which combines two above approaches includes *Charm* [16] that executes a highly efficiently hybrid search that skips many levels of the *IT-Tree* to quickly identify the frequent closed itemsets, instead of having to enumerate many possible subsets. A hash-based approach is applied to speed up subsumption checking. To compute quickly frequency as well as to reduce the sizes of the intermediate tidsets, the diffset

technique is used. Experiments in [16] showed that *Charm* outperforms the existing algorithms on many datasets. However, *very little studies* concentrate on *mining* the *generators* of frequent closed itemsets. *SSMG-Miner* was developed in [9] based on a depth-first search. While mining non-redundant generators in addition, it does not output the closed itemsets. Further, we need to access the dataset for generating local generators. Boulicaut et al. [8] presented *MineEX* for generating frequent free-sets. Unfortunately, the algorithm has to scan the dataset at each step for testing if a candidate is free. *Talky-G* [12] is presented for mining generators. Since it is stand-alone algorithm, we have to apply an algorithm, e.g. *Charm*, for mining closed itemsets and then group the generators of the same closed itemset. This combination is called *Touch* algorithm [12]. Further, we find that *Charm-L* [16] outputs explicitly the closed itemset lattice in a non-considerable additional amount of time compared to *Charm* (see [16]). *MinimalGenerator* [15] is an algorithm that discovers all generators of a frequent closed itemsets using only its immediate sub ones (in term of the set containment order). Hence, in a hybrid approach, it seems to be feasible to mine first the frequent closed itemset lattice by *Charm-L* and then to apply *MinimalGenerator* for discovering their generators. We call this hybrid algorithm *CharmLMG*.

Contribution. Almost of those algorithms discover either frequent closed itemsets or generators separately. The fact brings up a natural idea that it should mine concurrently both of them. *Close* is such an example, however, its execution is very expensive. In Section 2, we give some properties of closed sets, generators and their relationship. Those bring about our idea in the development of *GENCLOSE* presented in Section 3. It includes two following key features (reliably proven by the theorems of 1, 2):

1) *In a level-wise search, it mines first the generators by breadth-first search using a necessary and sufficient condition to determine the class of (i+1)-item generators from the class of i-item ones (i≥1) based on the sets of transactions.* On the other hand, *Close* uses a necessary condition based on closures.
2) Based on the way that *Charm* applies four properties of itemset-tidset pairs to extend itemsets to closed ones and the relationship of generators and closed itemsets, *we develop* and use *three new operators for expanding itemsets to their closures simultaneously with the mining of generators.*

The rest of the paper is organized as follows. We compare *GENCLOSE* against *CharmLMG* and *Touch* in Section 4. The conclusion is shown in Section 5.

Closed itemset, generator and their relationship

Given non-empty sets $\mathcal{O}$ containing *objects* (or a dataset of transactions) and $\mathcal{A}$ *items* (attributes) related to objects in $\mathcal{O}$. A set A of items in $\mathcal{A}$, $A \subseteq \mathcal{A}$, is called an *itemset*. A set O of objects in $\mathcal{O}$ is called an *object-set*. For $O \subseteq \mathcal{O}$, $\lambda(O)$ is the itemset that occurs in all transactions of O, defined as $\lambda(O) := \{a \in \mathcal{A} \mid a \in o, \forall o \in O\}$. The set of objects, in which itemset A appears as subset, is named by $\rho(A)$, $\rho(A) := \{o \in \mathcal{O} \mid A \subseteq o\}$. Define h and h' as union mappings of λ and ρ: $h = \lambda_o\rho$, $h' = \rho_o\lambda$, we say $h(A)$ and $h'(O)$ the *closures* of A, O. An itemset A is called a *closed itemset* if and only if (iff for short) $A=h(A)$ [17]. Symmetrically, O is called a *closed object-set* iff $O=h'(O)$.

Let *minsup* be the user-given *minimum support*, $minsup \in [1; |\mathcal{O}|]$. The number of transactions containing *A* is called the *support* of *A*, $supp(A) = |\rho(A)|$. If $supp(A) \geq minsup$ then *A* is called a *frequent itemset* [1]. If a frequent itemset is closed, thus, we call it a *frequent closed itemset*. For two non-empty itemsets of *G*, *A* such that $G \subseteq A \subseteq \mathcal{A}$, *G* is called a *generator* of *A* iff $h(G) = h(A)$ and ($\forall \varnothing \neq G' \subset G \Rightarrow h(G') \subset h(G)$). Let *Gen*(A) be the class of all generators of *A*. Since it is finite, we can enumerate it $Gen(A) = \{G_i : i = 1, 2, \ldots, |Gen(A)|\}$.

Proposition 1. $\forall A, A_1, A_2 \in 2^{\mathcal{A}}$, $\forall O, O_1, O_2 \in 2^{\mathcal{O}}$, the following statements hold true:

(a) $A_1 \subseteq A_2 \Rightarrow \rho(A_1) \supseteq \rho(A_2)$; $O_1 \subseteq O_2 \Rightarrow \lambda(O_1) \supseteq \lambda(O_2)$ *(1)*

(b) $A \subseteq h(A)$, $O \subseteq h'(O)$ *(2)*

(c) $A_1 \subseteq A_2 \Rightarrow h(A_1) \subseteq h(A_2)$; $O_1 \subseteq O_2 \Rightarrow h'(O_1) \subseteq h'(O_2)$ *(3)*

(d) $\rho(A_1)=\rho(A_2) \Leftrightarrow h(A_1)=h(A_2)$ and $\rho(A_1) \subset \rho(A_2) \Leftrightarrow h(A_1) \supset h(A_2)$ *(4)*

(e) $\rho(\cup_{i \in I} A_i) = \cap_{i \in I} \rho(A_i)$. Thus, $h(A) = \cap_{o:A \subseteq \lambda(o)} \lambda(\{o\}) = \cap_{o \in \rho(A)} \lambda(o)$ *(5)*

Proof: Due to space limit, we omit the proof. □

Proposition 2 *(Features of generators)*. Let $A \subseteq \mathcal{A}$. At the same time:

(a) $Gen(A) \neq \varnothing$ *(6)*

(b) $G \in Gen(A) \Leftrightarrow [\rho(G)=\rho(A)$ and $\forall G' \subset G \Rightarrow \rho(G') \supset \rho(G)]$ *(7)*

(c) *If G is a generator then* $\forall \varnothing \neq G' \subset G$: *G' is a generator of h(G')* *(8)*

Proof:

(a) Assuming that: $|A| = m$. Let us consider finitely the subsets of *A* that each of them is created by deleting an item of *A*: $A_{i1} = A \backslash \{a_{i1}\}$, $a_{i1} \in A$, $\forall i_1 = 1..m$.

Case 1: If $\rho(A_{i1}) \supset \rho(A)$, $\forall i_1 = 1..m$, then *A* is a generator of *A*.

Case 2: Otherwise, there exists $i_1 = 1..m$: $\rho(A_{i1}) = \rho(A)$. The above steps are repeated for A_{ij} $(j=1..m-1)$ until:

- *case 1* happens, thus, we get the generator $A_{i,j-1}$ of *A*; or
- *case 2* comes when $|A_{i,m-1}| = 1$, i.e. $\rho(A_{i1}) = \rho(A_{i2}) = .. = \rho(A_{i,m-1}) = \rho(A)$. Hence, $A_{i,m-1}$ is a generator of *A*. Since *A* is finite, we always get a generator of *A*.

(b) Based on property 4 of proposition 1, we have: $\lambda(\rho(G)) = h(G) = h(A) = \lambda(\rho(A)) \Leftrightarrow \rho(G) = \rho(h(G)) = \rho(h(A)) = \rho(A)$. Further, $\forall G' \subset G$: $\lambda(\rho(G')) = h(G') \subset h(G) = \lambda(\rho(G)) \Leftrightarrow \rho(G') = \rho(h(G')) \supset \rho(h(G)) = \rho(G)$.

(c) Supposing that the contrary happens. So, there exists a proper subset *G''* of *G'*, $G'' \neq \varnothing$ such that $\rho(G'') = \rho(G')$. For $G_0 = G' \backslash G''$, $G_1 = G \backslash G_0$, we have: $\varnothing \neq G_1 = (G \backslash G') +^{1} (G' \backslash G_0) = (G \backslash G') + G'' \subset (G \backslash G') + G' = G$. Otherwise, by *(5)*, $\rho(G) = \rho(G \backslash G') \cap \rho(G') = \rho(G \backslash G') \cap \rho(G'') = \rho((G \backslash G') + G'') = \rho(G_1)$, a contradiction! □

Property (c) of proposition 2 implies the *Apriori principle of generators*. Following from it, we find that any *(i+1)*-item generator $G = g_1 g_2 \ldots g_{i-1} g_i g_{i+1}$ [2] is created by the combining two i-item generators of $G_1 = g_1 g_2 \ldots g_{i-1} g_i$, $G_2 = g_1 g_2 \ldots g_{i-1} g_{i+1}$. Theorem 1 proposed hereafter give us an efficient way to *mine the class of (i+1)-item generators from the class of i-item ones.*

[1] The notation "+" represents the union of two disjoint sets.

[2] We write the set $\{a_1, a_2, .., a_n\}$ simply $a_1 a_2 .. a_n$.

Theorem 1 *(The necessary and sufficient condition to produce generators)*. For $\emptyset \neq G \subseteq \mathcal{A}$, $G_g := G \setminus \{g\}$, $g \in G$. The following conditions are equivalent:

(a) *G is a generator of h(G)*

(b) $\rho(G) \notin \bigcup_{g \in G} \{\rho(G_g)\}$ *(9)*

(c) $|\rho(G)| < |\rho(G_g)|$ *(i.e. supp(G) < supp(G_g))*, $\forall g \in G$ ***(10)***

(d) $|h(G_g)| < |h(G)|$, $\forall g \in G$ *(11)*

(e) $not(G \subseteq h(G_g))$, $\forall g \in G$ *(12)*

Proof:

(a) ⇔ (b): "⇒": If *G* is a generator of *h(G)* then every non-empty strict subset *G'* ($\emptyset \neq G' \subset G$, especially for $G' = G_g$) of *G* is also a generator of *h(G')*. Then, $\rho(G) \subset \rho(G_g)$. Hence, $\rho(G) \neq \rho(G_g)$, $\forall g \in G$. "⇐": On the contrary, suppose that *G* is not a generator of *h(G)*. Thus: $\exists G' \subset G: \rho(G') = \rho(G), G' \neq \emptyset$. It follows from *(1)* that $|G \setminus G'| = 1$ and $G' = G_g$, with $g \in G$. Therefore, $\rho(G_g) = \rho(G)$. That contradicts to *(9)*!

(b) ⇔ (c) ⇔ (d): By *(1)*, we have: $\rho(G) \subseteq \rho(G_g) \Leftrightarrow h(G) \supseteq h(G_g)$, $|\rho(G)| \leq |\rho(G_g)|$ and $|h(G)| \geq |h(G_g)|$, for any $g \in G$. Since $\mathcal{O}$ is finite, $\rho(G) = \rho(G_g) \Leftrightarrow |\rho(G)| = |\rho(G_g)| \Leftrightarrow |h(G)| = |h(G_g)|$, i.e. $not(9) \Leftrightarrow not(10) \Leftrightarrow not(11)$. Hence, we have: *(9) ⇔ (10) ⇔ (11)*.

(b) ⇔ (e): $not(9) \Leftrightarrow h(G) \in \cup_{g \in G} h(G_g)$ (since *(4)*) $\Leftrightarrow \exists g \in G: G \subseteq h(G_g) \Leftrightarrow not(12)$ *(*)*. We first have $g \in G \subseteq h(G)$ by property (b) of proposition 1. If $G \subseteq h(G_g)$ or $g \in h(G_g)$ then, based on *(3)*, *(4)* and *(5)*, $h(\{g\}) \subseteq h(G) \subseteq h(G_g) \subseteq h(G)$ and $h(G) = h(h(G_g) \cup h(\{g\})) = h(G_g)$. Thus, *(*)* is hold. Therefore, we have *(9) ⇔ (12)*. □

Remark 1. For $G_1 = g_1 g_2 \ldots g_{i-1} g_i$, $G_2 = g_1 g_2 \ldots g_{i-1} g_{i+1}$, $G = G_1 \cup G_2$, $G_k \in Gen(h(G_k))$, $k = 1,2$, the sufficient condition "$\rho(G) \neq \rho(G_g)$, *for* $g \in G_1 \cap G_2$" in *(9)* can not be skipped! This follows that the union of two generators could be not a generator. In fact, let us consider the dataset containing four transactions of *abcd*, *abc*, *abd*, *bc*, *c* and *d*. It is easy to see that $G_1 = bd$ is a generator of *abd* and $G_2 = bc$ a generator of *bc*. But, their union $G = G_1 \cup G_2 = bcd$ is not a generator since $\rho(bcd) = \rho(cd)$ but $cd \subset bcd$.

Remark 2. Let us review *consequence 2* in [10]. We find that the conclusion $h(I) = h(s_a)$ is not true because of the assumption that *I* is an *i*-generator and $\emptyset \neq s_a \subset I$. In fact, *I* is an *i*-generator (*i*-item generator) iff ($\forall \emptyset \neq s_a \subset I \Rightarrow h(s_a) \subset h(I)$). The consequence will become a necessary condition to an itemset is an i-item generator, if it is corrected as follows: *Let I be an i-itemset and* $S = \{s_1, s_2, .., s_j\}$ *a set of (i-1)-subsets of I where* $\bigcup_{s \in S} s = I$. *If* $\exists s_a \in S$ *such as* $I \subseteq h(s_a)$, *i.e.* <u>$h(I) = h(s_a)$</u>, *then* <u>*I is not a generator*</u>. We showed that *(12)* is more general and is also a sufficient one. This condition uses the closed itemset *h(G)*. But, at the time of discovering the generators, we do not know their closures. Hence, it seems reasonable to use object-set $\rho(G)$. Both *(9)* and *(10)* are designed for discovering generators, but *(10)* is simpler than *(9)*, especially on the datasets which object-set sizes can grow considerably!

Remark 3. In *Talky-G*, we need to check to see if a potential generator, which passed two tests of frequency and tidset, includes *a proper subset* with the same support in the set of mined generators (see *getNextGenerator* function [12]). Though a special hash table is used for doing the task, it seems to be very time-consuming. Condition *(10)*

shows that we only need to consider the candidate with its *immediate subsets* – the generators mined from previous step in a level-wise progress.

GENCLOSE algorithm

This section presents *GENCLOSE* that executes a breadth-first search over an *ITG-tree* (itemset-tidset-generator tree), like *Charm* which discovers an *IT-tree*, to discover generators. Simultaneously the corresponding closed itemsets are gradually explored. In each step, *GENCLOSE* tests the necessary and sufficient condition *(10)* for producing new generators from the generators of the previous step. Unlike *Close* that needs to scan the database, we propose three expanded operators (described formally in *3.2*) and apply them for discovering the closures along the process of mining generators.

Table 1. Dataset $\mathcal{D}_1$.

Objects	*Items*
o_1	a b c e g u
o_2	a c d f u
o_3	a d e f g u
o_4	b c e f g u
o_5	b c e
o_6	b c

Itemset-Tidset-Generator Tree. *Fig. 1* shows *ITG-tree* created from the execution of *GENCLOSE* on dataset $\mathcal{D}_1$, as shown in *Table 1*, with *minsup=1*. *This figure is used for the examples in the rest of the paper.* A tree node includes the following fields. The first, namely *generator set* (*GS*), contains the generators. The second, called *pre-closed itemset*, or *PC* for short, is the itemset that shares the same objects to them. *PC* is gradually enlarged to its closure – *h(PC)*. The third field *O* is the set of those objects. Thus, we have: $PC \supseteq \lambda(O)=h(G)$ and $\rho(PC)=O=\rho(G)$, $G \in GS$ (in implementation, we save their differences). The last one, called *supp*, stores the cardinality of *O*, i.e. *supp(PC)*. Initializing by *Root* (*Level 0*), its *PC* is assigned by $\emptyset$. By the convention, $\rho(\emptyset) := \mathcal{O}$; $Root.O = \mathcal{O}$. The *ITG-tree* is expanded level by level. Each one splits into the *folders*. Level *1*, called *L[1]*, includes only the folder containing the nodes as *Root*'s children. If the combination of two nodes *X* and Y_1 in level *i* (called *L[i]*) creates node *Z* at *L[i+1]*, we say *X* the *left-parent* of *Z*. If *X* and Y_2 also in *L[i]* form *T*, then *Z* and *T* have the *same left-parent X*, i.e. they are in the *same folder* according to *X*. However, *the nodes of the same folder* can have *different left-parents*!

3.1 GENCLOSE algorithm

The task of *GENCLOSE* is to output $\mathcal{LCG}$ – the *list of all frequent closed itemsets together the corresponding generators and supports*. Its pseudocode is shown in *Fig. 2*. We start by eliminating non-frequent items from $\mathcal{A}$ and sorting in ascending order

them first by their supports and then by their weights [16]. Each item $a \in \mathcal{A}^F$ forms a node in *L[1]*. Starting with *i=1, the i-th step* is broken into three phases as follows.

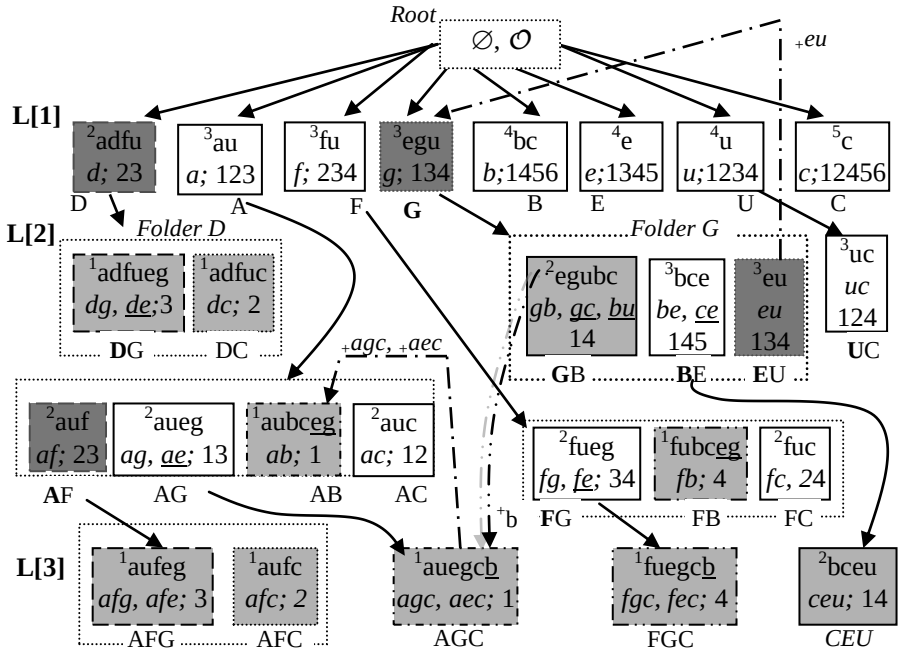

Fig. 1. *ITG-tree* created from $\mathcal{D}_1$ with *minsup=1*

GENCLOSE ($\mathcal{D}$, *minsup*):

1. $\mathcal{A}^F$ = *SelectFreqItems-Sort* ($\mathcal{A}$, *minsup*);
2. $\mathcal{LCG}$ = ∅; L[1] = ∅; HasNewLevel = true; i = 1;
3. **for each** a ∈ $\mathcal{A}^F$ **do** L[1] = L[1] ∪ {(|ρ(a)|, a, {a}, ρ(a))};
4. **while (** HasNewLevel **) do**
5. *ExpandMerge* (L[i]); // using $\mathcal{EOB}$.1
6. L[i+1] = ∅;
7. **for each** N_{Left} in L[i] **do**
8. **for each** N_{Right} in L[i]: Left < Right; N_{Left}, N_{Right} are in the *same folder* **do**
9. NewO = N_{Left}.O ∩ N_{Right}.O; NewSup = |NewO|;
10. **if** (NewSup ≠ N_{Left}.Supp and NewSup ≠ N_{Right}.Supp) **then**
11. **if** (NewSup ≥ minsup) **then**
12. NewPC = N_{Left}.PC ∪ N_{Right}.PC; // using $\mathcal{EOA}$
13. ***JoinGenerators*** (N_{Left}, N_{Right}, NewO, NewSup, NewPC, L[i+1]);
14. *InsertLevel* (L[i], $\mathcal{LCG}$); // using $\mathcal{EOC}$
15. **if** (L[i+1] = ∅) **then** HasNewLevel = false;
16. i = i+1;
17. return $\mathcal{LCG}$; // *all frequent closed itemsets, their generators and supports*

Fig. 2. *GENCLOSE* algorithm

The first phase is called by *ExpandMerge* procedure for *extending pre-closed itemsets of the nodes at L[i]* as well as *merging them* using operator $\mathcal{EOB}.1$ (see *3.2*). Let N_{Left} and N_{Right} be two nodes at *L[i]* that *Left* comes before *Right*. It tests which of three following cases is satisfied. If $N_{Left}.O \subset N_{Right}.O$, since property 4 of proposition 1, $h(N_{Left}.PC) \supset h(N_{Right}.PC)$. Toward to closures, $N_{Left}.PC$ should be extended by $N_{Right}.PC$. If $N_{Left}.O \supset N_{Right}.O$, $N_{Right}.PC := N_{Right}.PC \cup N_{Left}.PC$. Otherwise, we push all generators in $N_{Right}.GS$ into $N_{Left}.GS$ and add $N_{Right}.PC$ to $N_{Left}.PC$. Then, we *move the nodes having the same folder* with N_{Right} to the folder including N_{Left} and discard N_{Right}.

The second phase *produces the nodes at L[i+1]* by considering each pair (N_{Left}, N_{Right}), written N_{Left}-N_{Right}, at *Lines 7, 8*. Since $NewO = N_{Left}.O \cap N_{Right}.O$ is a new closed object-set, if its cardinality exceeds *minsup*, we need to determine new frequent closed itemset $\lambda(NewO)$. First, we create its core, called *NewPC*. We then jump into *JoinGenerators* (*Fig. 3*) for mining its generators from i-item generators containing in N_{Left}-N_{Right}. We check *(10)* only for $g \in G_0$ since it is always satisfied for $g=g_i$ and $g=g_{i+1}$. <u>*Case i=1*</u>, we have immediately *2*-item generators. <u>*Otherwise*</u>, if we touch a value of $g \in G_0$ such that $G_g = G \backslash \{g\}$ is not an *i*-item generator, it is obviously that *G* is not an *(i+1)*-item generator of *NewPC*. But we should not be in the hurry. It is necessary to consider the latter for *expanding* pre-closed itemsets. Let $Node_g$ be the node including G_g as an *i*-item generator. If it does not exist, i.e. *G* is not a generator, then we move to the next value of *g*. Otherwise, we check if $Node_g.Supp = NewSup$. If yes, i.e. *G* is not a generator, $Node_g.PC$ at *L[i]* is enlarged by *NewPC*. If no, we expand *NewPC* by $Node_g.PC$. Clearly, *not only new pre-closed itemsets but also the ones at L[i] are expanded*. If *(10)* is satisfied, *G* is a new generator produced from (G_l, G_r). We return to *Line 18* to process next generator pairs. If there exists at least a new *(i+1)*-item generator, we get new node $^{NewSup}NewPC\text{-}NewGS\text{-}NewO$ at *L[i+1]* and then return to *Lines 7, 8* for considering the remaining node pairs.

```
JoinGenerators (N_Left, N_Right, NewO, NewSup, NewPC, L[i+1]):
18.  for each (G_l∈N_Left.GS, G_r∈N_Right.GS: |G_l|=i, |G_r|=i and |G_l∩G_r|=i-1) do
19.      G = G_l∪G_r; G_0 = G_l∩G_r; G_is_Generator = true;
20.      for each g∈G_0 do
21.          G_g = G\{g};
22.          Node_g = SearchNodeWithGenerator (G_g);
23.          if ( Node_g is null  or  NewSup = Node_g.Supp ) then
24.          {
25.              G_is_Generator = false;
26.              if ( Node_g is not null ) then      // NewSup = |Node_g.O|
27.                  Node_g.PC = Node_g.PC ∪ NewPC;              // using EOB.2
28.          }
29.          else              // G can be a generator!
30.              NewPC = NewPC ∪ Node_g.PC;                      // using EOA
31.      if ( G_is_Generator ) then NewGS = NewGS ∪ {G};
32.  if ( |NewGS| ≥ 1 ) then
33.      L[i+1] = L[i+1] ∪ {(NewSup, NewPC, NewGS, NewO)};
```

Fig. 3. *JoinGenerators* procedure

At the **last phase**, we will *finish the extension* for the *PC*s of the nodes of *L[i]* by *InsertLevel* that inserts in turn the nodes at *L[i]* into $\mathcal{LCG}$. Before adding a node *X*, *GENCLOSE* makes a check *if X.PC is closed*. If *there exists P* in $\mathcal{LCG}$ such that *supp(P)=X.Supp* and $P \supseteq X.PC$, *P* is the closure that *X.PC* wants to touch. We *push* those *new generators* into the generator list of *P*. In contrast, since *X.PC* is a *new closed itemset*, we insert *X.PC*, its generators and support to $\mathcal{LCG}$.

An example. *Fig. 1* is used through the example. For short, a tree node is written in the form $^{Supp}PC\text{-}GS\text{-}O$. At the beginning, we have the following nodes $D:=^2d\text{-}\{d\}\text{-}23$ [3], $A:=^3a\text{-}\{a\}\text{-}123$, $F:=^3f\text{-}\{f\}\text{-}234$, $G:=^3g\text{-}\{g\}\text{-}134$, *B*, *E*, *U* and *C*. First, *D* is considered with *A*, *F*, *U* and it becomes $^2dafu\text{-}\{d\}\text{-}23$ since *D.PC* is contained in their *PC*s. By the similar computations, we get *L[1]*. Next, we obtain *L[2]* by combining the node pairs of *L[1]*. Then, $\mathcal{LCG}$ is $\{^2dafu_d, ^3au_a, ^3fu_f, ^3geu_g, ^4bc_b, ^4e_e, ^4u_u, ^5c_c\}$ where the generators were written in the right at the bottom. At the next step, *AG.PC* and *FG.PC* are taken into the *PC*s of *AB* and *FB*. Then, we merge *DE* into *DG*, thus, *DG* becomes $^1adfueg\text{-}\{dg, \underline{de}\}\text{-}3$. We try to consider the next combinations and find that the nodes *AE*, *FE*, *GC*, *BU*, *EC* should be merged into *AG*, *FG*, *GB*, *BE*, accordingly. Right after, *EC* is moved to the folder including *BE* and then this folder is merged into folder *G*. We look at pair *AG-AC*, we find out two new generators *agc*, *aec*. Here, *NewPC*, which is initially *aueg+auc*, *is enlarged by GB.PC* and becomes $auegc\underline{b}$. Then, *GENCLOSE* calls *InsertLevel* to insert *L[2]* into $\mathcal{LCG}$. Since *AF.PC=auf* is not closed, *af* is a new 2-item generator of *adfu*. Then, we take into $\mathcal{LCG}$ the *PCs* of the remaining nodes and their corresponding generators. The next computations of *GENCLOSE* are straightforward.

3.2 The expanded operators *EoA*, *EoB* and *EoC*

We call h_F, O_F the pre-closed itemset and object-set according to generator F. Suppose that $\mathcal{LCG}$ contained all frequent closed itemsets that includes *(i-1)*-item generators.

Let G be an i-item generator created by joining two nodes at L[i-1]:

EoA: h_G *is formed* by ($\forall g \in G$: $G_g = G\backslash\{g\}$ is an *(i-1)*-item generator):

$$h_G \leftarrow h_G \cup \bigcup_{g \in G} h_{G_g} \qquad (13)$$

EoB.1: h_G *was extended* by: $$h_G \leftarrow h_G \cup \bigcup_{G\sim \in L[i]: O_G \subseteq O_{G\sim}} h_{G\sim} \qquad (14)$$

After joining the nodes at L[i]:

EoB.2: Let $\{a\} \cup G$, called *aG*, be a new candidate *(i+1)*-item generator. If $|O_{aG}| = |O_G|$, then *aG is not an (i+1)-item generator*. Thus: $$h_G \leftarrow h_G \cup \bigcup_{a: |O_{aG}| = |O_G|} h_{aG} \qquad (15)$$

EoC: Consider *how to insert* (sup_G, h_G, GS) into $\mathcal{LCG}$. First, we check to see if *there exists a closed itemset P* being in $\mathcal{LCG}$ such that:

[3] We write the set of objects simply their identifiers.

$$supp(P) = supp(h_G) \text{ and } P \supseteq h_G \quad (16)$$

1) If *yes*, h_G is not a new closed itemset. Thus, we *add i-item generators* to *P*. 2) If *no*, we *insert new closed itemset* h_G *together its i-item generators* and support into $\mathcal{LCG}$.

Theorem 2 *(The completeness of the expanding operators). After we use those operators for every node at level i containing i-item generators,* h_G *is closed. Thus,* $\mathcal{LCG}$ *is added by frequent closed itemsets having i-item generators or only those generators.*

Proof:

Case i=1: After we consider the set containment relation for the object-sets according to *1*-item generators: $\forall b \in \mathcal{A}$: $\forall a \in h(b) \Leftrightarrow h(a) \subseteq h(b) \Leftrightarrow O_a \equiv \rho(a) \supseteq O_b \equiv \rho(b)$, therefore: $h_b \leftarrow h_b \cup \{a\}$. Then, $h(b) \subseteq h_b$. Clearly, $h(b) \supseteq h_b$. Hence $h(b) = h_b$.

Case i>1: suppose that the conclusion is true for *1, 2, .., i-1, i≥2*. After $\mathcal{EoA}$, $\mathcal{EoB}$ finishes: $h_G = \bigcup_{g \in G} h_{G_g} \cup \bigcup_{G\sim : O_G \subseteq O_{G\sim}} h_{G\sim} \cup \bigcup_{a : |O_{aG}| = |O_G|} h_{aG}$.

We make the assumption that h_G *is not closed*, $h_G \subset h(G)$. For $\forall a \in h(G) \backslash h_G$ *(**)*, we will prove that there exists $P \in \mathcal{LCG}$ such that any present generator of *P* has at most *i-1* items: $h(P)=h(G) \supset h_G \Leftrightarrow [h(P) \supset h_G \text{ and } \rho(P)=O_G] \Leftrightarrow [h(P) \supset h_G \text{ and } |\rho(P)|=|O_G|]$.

- First, we prove that aG_g is not an i-item generator of $h(aG_g)$. Conversely, joining *G* with aG_g generates *aG*. Since $h(G)=h(aG)$, so *aG* is not a generator. However, since $\mathcal{EoB}$ was used, *a* was added to h_G. That implies $a \in h_G$ which contradicts to *(**)*!

- Since aG_g is not an *i*-item generator of $h(aG_g)$, there exists a minimal generator $\emptyset \neq G' \subset aG_g$: $h(G') = h(aG_g)$. (A) If $a \notin G'$, then $G' \subseteq G_g$. Hence, $h(aG_g) = h(G') \subseteq h(G_g) \subseteq h(aG_g)$. It follows that $a \in h(aG_g)=h(G_g)$, i.e. a contradiction to the fact that $a \notin h_G$ (**)! (B) Therefore, $a \in G'$ and there exists $G'=aG_{gB}$, with $\emptyset \neq B \subseteq G_g$: $h(\underline{aG_{gB}})=h(aG_g)$ $(G_{gB}=G_g \backslash B=G \backslash (gB))$. Since $0 \leq |G_{gB}| \leq i-2$, $1 \leq |G'| \leq i-1$. In other words, generator *G'* of $h(aG_g)$ has at most i-1 items.

- What is left is to show that $\exists g \in G$: $h(\underline{aG_{gB}})=h(aG_g)=h(G) \supset h_G$. It means that we will find out the closure $h(G)$ of h_G from the ancestor nodes $h(G')$ ($\supset h_G$ and $|G'| \leq i-1$) in $\mathcal{LCG}$. Assume that the conversion comes: $\forall g \in G$: $h(aG_g) \subset h(G)$ $(\Leftrightarrow \rho(G) \subset \rho(aG_g))$ *(***)* and $\exists g' \in B \subseteq G_g$: $G'=aG_{gB}$ is a generator of $h(aG_g) = h(aG_{gB})$. Clearly, $g \neq g' \notin G'$. Then $G' = aG_{gB} \subseteq aG_{g'}$. But, we have also: $\rho(aG_{g'}) \subseteq \rho(G_{g'})$, $\rho(aG_{gB}) = \rho(aG_g) \subseteq \rho(G_g)$. Taking the intersection of two sides, we have: $\rho(aG_{g'})=\rho(aG_{g'}) \cap \rho(aG_{gB}) \subseteq \rho(G_{g'}) \cap \rho(G_g)=\rho(G_{g'} \cup G_g)=\rho(G) \subset \rho(aG_{g'})$ (by *(***)*). The contradiction happens! □

Experimental Results

The experiments below were carried out on a i5-2400 CPU, 3.10 GHz @ 3.09 GHz, with 3.16 GB RAM, running under Linux, Cygwin. To test the performance and correctness of *GENCLOSE*, we compare it against *CharmLMG* (http://www.cs.rpi.edu/~zaki) and *D-Touch* (a fast implementation of *Touch*, http://coron.wikidot.com/) on six following benchmark datasets at http://fimi.cs.helsinki.fi/data/: *C20d10k, C73d10k, Pumsb, Pumsbstar, Mushroom* and *Connect*. They are highly correlated and dense datasets in terms that they produce

many long frequent itemsets as well as only a small fraction of them is closed. The dataset characteristics can be found in [10, 16]. We did not choose the sparse ones since in which almost frequent itemsets are closed and they are generators themselves. We decided to get seven small values of *minsup* thresholds for each dataset, computed on percentages, ranged from: 80% down to 30% for *Connect*, 95% down to 70% for *Pumsb*, 75% down to 45% for *C73d10k*, 40% down to 20% for *Pumsbstar*, 15% down to 0.1% for *C20d10k*, and 18% down to 0.5% for *Mushroom*. The reason is that, for the big ones, the mining processes are often short, so, there is no difference in the performances of *GENCLOSE*, *CharmLMG* and *D-Touch*.

The experiments show that *the output of GENCLOSE* is *identical to* the ones of *CharmLMG* and *D-Touch*. Let *Time-GENCLOSE*, *Time-CharmLMG* and *Time-DTouch* be their running times, computed in seconds. *Fig. 4* shows them for *Connect*. For each dataset, we get the average number of the ratios of the running times of *CharmLMG* and *D-Touch* compared to *GENCLOSE (Time-CharmLMG / Time-GENCLOSE* and *Time-DTouch / Time-GENCLOSE*) on all minsup values. *Fig. 5* shows those numbers for all datasets. We find that, in general, *GENCLOSE are over many times faster than CharmLMG* and *D-Touch*. The reductions in the execution time of *GENCLOSE*, compared to *D-Touch* are lowered for *C20d10k* and *Mushroom*. But, one note that, *D-Touch* can not execute: for *C73d100k* with minsups of *50%* and *40%*; for *Pumsb* with minsups *75%*, *70%*; for *Pumsb** with minsup = *20%*.

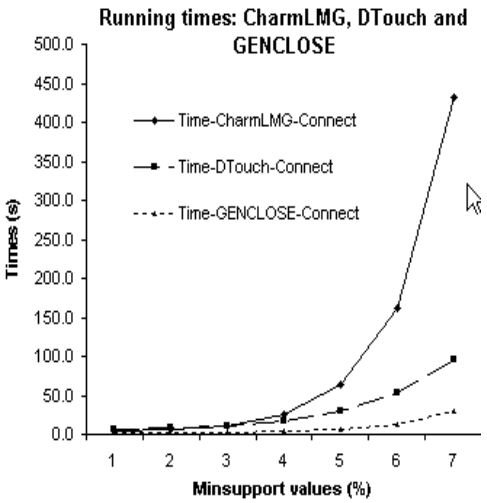

Fig. 4. The running times of *CharmLMG, D-Touch* and *GENCLOSE* on *Connect*

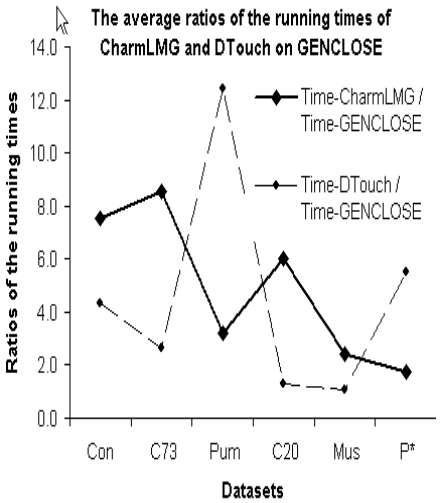

Fig. 5. Average ratios of the running times of *CharmLMG, D-Touch* on *GENCLOSE*

Conclusions

We gave some properties of closed sets and generators as well as their relations. Based on them, we developed *GENCLOSE*, an efficient algorithm for mining concurrently frequent closed itemsets and their generators. The background of the algorithm included the necessary and sufficient condition for producing generators and the oper-

ators for expanding itemsets were proven reliably. Many tests on benchmark datasets showed its efficiency compared to *CharmLMG* and *D-Touch*.

For mining either closed itemsets or generators separately, the depth-first algorithms usually outperform the level-wise ones. An interesting extension is to develop a depth-first miner based on the proposed approach.

References

1. Agrawal, R., Imielinski, T., Swami, N.: Mining association rules between sets of items in large databases. In Proceedings of the ACM SIGMOID, pp. 207-216 (1993).
2. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In Proceedings of the 20th International Conference on Very Large Data Bases, pp. 478-499 (1994).
3. Anh, T., Tin, T., Bac, L.: Structures of Association Rule Set. Lecture Notes in Artificial Intelligence, Part II, Springer-Verlag Berlin Heidelberg, pp. 361-370 (2012).
4. Anh, T., Hai, D., Tin, T., Bac, L.: Mining Frequent Itemsets with Dualistic Constraints. In PRICAI 2012, LNAI 7458, Springer-Verlag, pp. 807-813 (2012).
5. Balcazar, J.L.: Redundancy, deduction schemes, and minimum-size base for association rules. Logical Methods in Computer Sciences, 6(2:3), pp. 1-33 (2010).
6. Bayardo, R.J.: Efficiently Mining Long Patterns from Databases. In Proceedings of the SIGMOD Conference, pp. 85–93 (1998).
7. Burdick, D., Calimlim, M., Gehrke, J.: MAFIA: A maximal frequent itemset algorithm for transactional databases. In Proceedings of ICDE'01, pp. 443–452 (2001).
8. Boulicaut, J., Bykowski, A., Rigotti, C.: Free-Sets: A Condensed Representation of Boolean Data for the Approximation of Frequency Queries. Data Mining and Knowledge Discovery, 7, pp. 5–22 (2003).
9. Dong, G., Jiang, C., Pei, J., Li, J., Wong,L.: Mining Succinct Systems of Minimal Generators of Formal Concepts. DASFAA 2005, LNCS 3453, pp. 175–187 (2005).
10. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Efficient mining of association rules using closed item set lattices. Information systems, 24(1), pp. 25-46 (1999).
11. Pasquier, N., Taouil, R., Bastide, Y., Stumme, G., Lakhal, L.: Generating a condensed representation for association rules. J. of Intel. ligent Infor. Sys, 24(1), pp. 29-60 (2005).
12. Szathmary, L., Valtchev, P., Napoli, A.: Efficient Vertical Mining of Frequent Closed Itemsets and Generators. In IDA 2009, 393-404 (2009).
13. Wang J., Han, J., and Pei, J.: Closet+: Searching for the best strategies for mining frequent closed itemsets. In Proceedings of ACM SIGKDD'03 (2003).
14. Zaki, M.J., Gouda, K.: Fast Vertical Mining Using Diffsets. In Proc. 9th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (2003).
15. Zaki, M.J.: Mining non-redundant association rules. Data mining and knowledge discovery, no. 9, pp. 223-248 (2004).
16. Zaki, M.J., Hsiao, C.J.: Efficient algorithms for mining closed itemsets and their lattice structure. IEEE Trans. Knowledge and data engineering, 17(4), pp. 462-478 (2005).
17. Wille, R.: Concept lattices and conceptual knowledge systems. Computers and Math. with App., 23, pp. 493-515 (1992).