International Journal of Advanced Computer Research (ISSN (print): 2249-7277 ISSN (online): 2277-7970)
Volume-4 Number-3 Issue-16 September-2014

An Efficient Lattice-Based Approach for Generator Mining

Pham Quang Huy[1], Truong Chi Tin[2]

Abstract

Mining frequent closed itemsets and theirs corresponding generators seem to be the most effective way to mine frequent itemsets and association rules from large datasets since it helps reduce the risks of low performance, big storage and redundancy. However, generator mining has not been studied as much as frequent closed itemsets mining and it has not reached the ultra-optimization yet. In this paper, we consider the problem of enumerating generators from the lattice of frequent closed itemsets as the problem of "distributing M machines to solve N jobs" in order to introduce a close and legible point of view. From this, it is easy to infer some interesting mathematical results to solve the problem easily. Our proposed algorithm, GDP, can efficiently find all generators in very low complexity without duplicated or useless consideration. Experiments show that our approach is reasonable and effective.

Keywords

Generator, minimal generator, generator mining, lattice of closed frequent itemsets, lattice-based algorithm, dynamic programming algorithm, parallel algorithm.

1. Introduction

Association rule (AR) is known to play an important role in data mining and to have many applications in reality. Association rule mining is usually divided into two sub problems: mining all frequent itemsets (FIs) from data and deriving association rules from those frequent mined ones [1]. However, the number of FIs is often numerous since they grow in exponent of the number of itemsets, therefore, algorithms that directly mine FIs or ARs from data usually face the challenges of performance and storage, as well as the problem of generating duplicated candidates.

Manuscript received August 24, 2014.
Pham Quang Huy, Department of Mathematics and Informatics, DaLat University, Vietnam.
Truong Chi Tin, Department of Mathematics and Informatics, DaLat University, Vietnam.

A more effective approach is to mine only the class of all frequent closed itemsets (FCIs) because they are commonly much fewer than the FIs and they are essential information for deriving all FIs as well as all ARs.

Indeed, a closed itemset (also called a closure) is the largest itemset among the ones contained in the same set of transactions. Based on FCIs we can partition all FIs or ARs into equivalent classes. Then, together with their corresponding generators, it is possible to non-repeatedly derive all FIs and ARs, without the loss of their support and their confidence [2, 3, 4, 5, 6]. As stated in [7], among the best and well-known FCI mining algorithms, there are Charm [6] and FPClose [8]. Charm's search space is an IT-tree, in which each node is a pair of itemset and tidset (a list of transaction identifiers containing that itemset). Whereas, the search space of FPClose is the space of FP-trees, with each tree is a compression of a conditional dataset. FCIs can also be mined by analyzing the lattice of concepts (e.g., Titanic algorithm [9]). There are also parallel algorithms for FCI mining, such as PLCM_{QS} [10], AFOPT-close [11].

On the other hand, generators are the minimal itemsets in each class [4, 12]. This definition is equivalent to the term of "minimal generator" in [6]. FCI and its generators are keys to induce all other FIs in their class. For instance, the authors in [2] proposed a structure of the FIs in each class via its closure and generators, allowing generating them quickly without replication. They also help to divide ARs into equivalent classes such that in each class, it is only necessary to mine only the basic rules and the consequent ones can be easily derived along with their support and confidence. For example, in [4], Pasquier et al proposed the basic rules in the form of $G \rightarrow C \backslash G$, where C is a closure and G is a generator ($G \subset C$). Zaki [6] mentioned the concept of the most general rule in the form of $G \rightarrow \{m\}$, in which G is a generator and m is an item. If G and $G \cup \{m\}$ have the same closure, they are exact rules; otherwise they are approximate rules. In [2, 5], based on FCIs and their generators, the authors partitioned the class of all ARs into equivalent classes, where each one is presented by a pair of FCIs, [L, S] (with $L \subseteq S$). Then,

the basic ARs (which are usually quite little) are also in form G $\rightarrow$ S\G, where G is a generator of L. The consequent ones can quickly be induced, without any duplication that happens in Pasquier approach in [4].
In addition, lattice of FCIs and their generators play also an important role in extracting FIs and ARs with constraints (that satisfy certain user needs). However, this is out of the scope of this paper. We refer readers to [3] for more information.

Related works. While there have been a lot of researches on FCI mining, studies on generator mining are still limited and they have their own drawbacks. Generators can be mined from a given lattice of FCIs, such as MinimalGenerators[6, 13], Compute_hs_mingen [13] or directly from data as algorithm Touch [14] does. MinimalGenerators and Compute_hs_mingen base on the fact that "the minimal generators of a closed itemset C are the minimal itemsets that are subsets of C but not a subset of any of C's (immediate) closed subsets" [p. 239, 6]. Since MinimalGenerators is an Apriori-like algorithm [1], it considers too many useless candidates and runs very slow. Whereas, Compute_hs_mingen, a procedure coded in C/C++, gradually generates each candidate G that has the same closure as C then check for G to be minimal. Unfortunately, it requires many set computations and generates some duplicated candidates. Once it can not avoid all duplicated candidates during the main process, it has to perform a final check to eliminate duplicated generators at the end. The three main disadvantages in these two algorithms are that they use many set computations, their current steps cannot make use of previous steps and there are a lot of redundant candidates. Meanwhile, Touch [14] mines generators directly from data, based on an IT-tree. Firstly, it uses Charm for mining all FCIs. Then generators are mined and combined accordingly to its closure. Its idea is that "a new candidate G is a generator if there are no already found generators that are subsets of G having the same support". A hash function is used to gather the closest related generators for this testing G and to combine G with its closure. This algorithm worked well on most of our experimental cases, except that it ran out of memory and failed to run for some cases when datasets are quite large (see Table 1, Section 5) or the minimum support threshold is not small enough. Touch might not be able to deal with very large datasets because it is not a parallel algorithm and it relies on IT-tree in which is hard to fix in memory when the tidset nodes are too long. In addition, in these algorithms, the time for mining generators are quite high as compared to those for mining FCIs.

Recently, Tran et al introduced the algorithm GenClose [12] that mines FCIs and generators simultaneously based on IT-like-tree, in breadth-first search manner. They pointed out the necessary and sufficient condition to generate a generator of (k+1)-items from its sub generators of k-attributes.

Contributions. From the important roles mentioned of generators and the drawbacks of those cited algorithms, it is worthy to develop a more effective algorithm to mine generators. Our approach is also to enumerate all generators based on the lattice of FCIs by the following two reasons. First, the lattice of FCIs for input can be considered to be always available even for very large dataset as it does not depend on the number of transactions of dataset and there have been parallel algorithms for FCI. Second, for enumerating all generators of each FCI C, only its immediate frequent closed subsets, a piece of the lattice, are needed. Hence, it can be parallelized.
By transforming the problem of finding all generators of a FCI based on its immediate closed subsets into a problem of distributing M machines to N jobs, we develop an efficient algorithm, called GDP which can find all generators of FCIs in a low complexity, without any duplicated or useless considerations. In addition, costly set computations are avoided by the idea of "finding the solution for the current step is based on the previous steps" and by recurrent expressions that compute on the cardinality of sets instead of on set. Experiment shows that GDP significantly outperforms MinimalGenerators, Compute_hs_mingen and Touch in every experimental case. Especially, its time for finding generators is much smaller than the time for mining FCIs (by Charm).

The remainder of the paper is organized as follows. Section 2 provides some basic concepts and an important necessary and sufficient condition to find generators of a FCI based on its immediate closed subsets. In Sections 3, we transform this problem into a problem of distributing M machines to N jobs and then, present an effective way to solve this problem. The algorithm GDP is introduced in Section 4. All experimental results are listed in Section 5 and Section 6 is the conclusion.

2. Basic concepts

Given a context (O, A, R) where O, A, R respectively are attribute set (or set of items), object set (or set of transactions) and the binary relation in $O \times A$. Two Galois connections $\lambda: 2^O \rightarrow 2^A$, $\rho: 2^A \rightarrow 2^O$ are defined as follows: $\forall C, O: \varnothing \neq C \subseteq A, \varnothing \neq O \subseteq O$, $\lambda(O) = \{a \in A : (o, a) \in R, \forall o \in O\}$, $\lambda(\varnothing) = A$ and $\rho(C) = \{o \in O: (o, a) \in R, \forall a \in C\}$, $\rho(\varnothing) = O$. Then, the operator $h = \lambda o \rho$ in 2^A is a closure operator, and h(C) is the closure of C. The set $C \subseteq A$ is closed iff[1] h(C) = C. Let $[C] = \{X \subseteq A: h(X) = h(C)\}$ be the class of all itemsets having the same closure as C.
Given a set $C \subseteq A$ and a minimum threshold: $0 <$ minsupp ≤ 1. Let $supp(C) = |\rho(C)|/|A|$, denotes the support of C, then C is frequent if $supp(C) \geq$ minsupp. Denote FCS as the class of all FCIs and $\preccurlyeq_A$ the order relation based on the inclusion relation on subsets of A. Then, $L_A \equiv (FCS, \preccurlyeq_A)$ is the lattice of FCIs.

From now on, for convenience, we always assume that: $C \in FCS$, $\varnothing \neq G \subseteq C \subseteq A$, and $G_g \equiv G \backslash \{g\}$, $\forall g \in G$. The sign "," substitute for operator "$\wedge$" in some mathematical expressions.

Definition 1. (Generator). *G is a generator of C iff h(G) = h(C) and ($\forall G': G' \subset G \Rightarrow h(G') \subset h(G)$). For simplicity, we omit the case where $\varnothing$ is a generator of C ($h(\varnothing) = C$). Thus, we can denote $Gen(C) = \{G \in [C]: \forall g \in G, h(G_g) \subset h(G)\}$ as the class of all generators of C.*
Denote $S_C \equiv \{Y \in FCS: (Y \subset C) \wedge (\nexists Z \in FCS: Y \subset Z \subset C\}$, the class of all FCIs that are immediate subsets of C. We have the following proposition:

Proposition 1. *$G \in [C] \Leftrightarrow \forall Y \in S_C, G \not\subset Y \Leftrightarrow \forall Y \in S_C, \exists g \in G: g \notin Y$.*

Proof. "$\Rightarrow$": Assume that $G \in [C]$. If $\exists Y \in S_C: G \subseteq Y \subset C$ then $h(G) \subseteq Y \subset C = h(G)$. This is a contradiction!
"$\Leftarrow$": Assume that $\forall Y \in S_C$, $G \not\subset Y$ but $G \notin [C]$, i.e. $h(G) \subset C$, then $\exists Y \in S_C$: $G \subseteq h(G) \subseteq Y$: Contradiction! Thus, $G \in [C]$.

Then, we have the following simpler criterion to check for an itemset G to be a generator of a closure C based on S_C.

[1] Iff: if and only if

Theorem 1. *$\forall C \in FCS$, $\forall G: \varnothing \neq G \subseteq C \subseteq A$,*

$$G \in Gen(C) \Leftrightarrow \begin{cases} \forall Y \in S_C, \exists g \in G: g \notin Y & (a_1) \\ \forall g \in G, \exists Y \in S_C: (g \notin Y, G_g \subseteq Y) & (b_1) \end{cases}$$

Proof. By Proposition 1, $(a_1) \Leftrightarrow G \in [C]$. Now, under condition (a_1) is satisfied, we just need to prove $(b_1) \Leftrightarrow \forall g \in G, h(G_g) \subset h(G)$.
"$\Leftarrow$": $\forall g \in G$, $h(G_g) \subset h(G) = C$, then there exists $Y \in S_C$: $G_g \subseteq h(G_g) \subseteq Y \subset C$. Assume that, $g \in Y$, then $G \subseteq Y$ and $C = h(G) \subseteq Y \subset C$. This is a contradiction! Thus, $g \notin Y$.
"$\Rightarrow$": $\forall g \in G, \exists Y \in S_C$: $G_g \subseteq Y \Rightarrow h(G_g) \subseteq Y \subset C = h(G)$. Thus, $(b_1) \Leftrightarrow \forall g \in G, h(G_g) \subset h(G)$. □

For simplicity, in our examples, we use 123 for the set {1, 2, 3}.

R = {**o₁** = {1, 2, 3, 4, 5}, **o₂** = {1, 3, 5}, **o₃** = {1, 4, 5}, **o₄** = {2, 3, 4, 5}}

Figure 1: Dataset R.

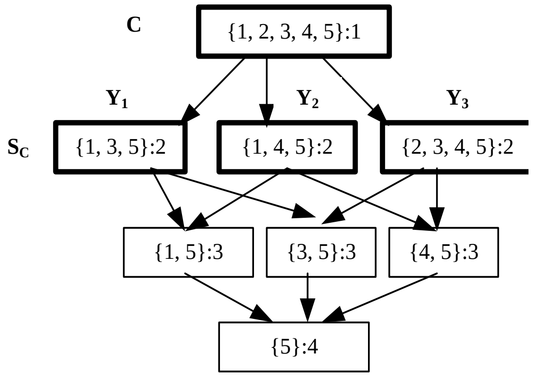

Figure 2: The lattice of FCIs corresponding to R and minsupp = 1. In which, C = {1, 2, 3, 4, 5}, S_C = {Y_1 = {1, 3, 5}, Y_2 = {1, 4, 5}, Y_3 = {2, 3, 4, 5}}.

Example 1. Given a dataset R as in Figure 1. With minsupp = 1, the corresponding lattice of FCIs is shown in Figure 2, in which each FCI is in form of itemset:support. Let find all generators of C = 12345 in this lattice. By (a_1), we can generate a candidate itemset in [C] and then check if it is minimal by (b_1).
We have $S_C = \{Y_1 = 135, Y_2 = 145, Y_3 = 2345\}$. Let $D_j = C \backslash Y_j$, □$\forall j \in 123$. Thus, $D_1 = 24$, $D_2 = 23$, $D_3 = 1$. To generate a candidate G, for each D_j we must choose an item to add into G. □$\forall g \in G$, $j \in 123$, we

say "j supports G_g" if $g \in Y_j$. If G_g is supported, for all g in G then (b_1) is satisfied.
Let's consider G = <u>21</u>. With $G_2 = G\backslash\underline{2} = \underline{1}$, there is j = 1 that supports G_2. With $G_1 = G\backslash\underline{1} = \underline{2}$, neither j = 1 nor j = 2 supports G_1 but j = 3 does. Thus, <u>21</u> is a generator of C.

Let's consider G = <u>231</u>. With G_2 = <u>13</u>, there is 1 that supports G_2. With G_3 = <u>12</u>, there is no j supporting it. Thus, <u>231</u> is not a generator of C.
Let's consider G = <u>421</u>. With G_4 = <u>12</u>, there is no j supporting it. Thus, <u>421</u> is not a generator of C.
Let's consider G = <u>431</u>. With G_4 = <u>13</u>, there is 1 that supports G_4. With G_3 = <u>14</u>, 1 doesn't support G_3 but 2 does. With G_1 = <u>34</u>, there is 3 supporting it. Thus, <u>431</u> is another generator of C.
Finally Gen(C) = {<u>12</u>, <u>431</u>}.
One can see in this example that for each candidate G, many set computations are required. Moreover, there are duplicated considerations, such as G_2 = <u>13</u> when G = <u>231</u> and G_4 = <u>13</u> when G = <u>431</u> or G_3 = <u>12</u> when G = <u>231</u> and G_4 = <u>12</u> when G = <u>421</u>. These duplications should be eliminated.

3. Our new approach

In this section, we restate the problem of finding Gen(C) based on S_C (denote as Gen(C, S_C)) in the language of a problem of distributing M machines to N jobs which is closer to readers' point of view. Then, we propose some theoretical results that turn costly conditions computing on sets in Theorem 1 to simpler ones to solve this problem effectively in a dynamic programming fashion.

3.1. Transform Gen(C, SC) to the problem of distributing M machines to N Jobs

Definition 2 (A problem of distributing M machines to N Jobs). *Given a set $C = \{m_1, m_2, ..., m_M\}$ containing M machines, $S_C = \{Y_1, Y_2, ..., Y_N\}$ being a class of N subsets of C, and a set $J = J(S_C) = \{1, 2, ..., N\}$ containing N jobs. For each $j \in J$, $m \in C$, $\varnothing \neq G \subseteq C$, machine m can solve the job j iff $m \notin Y_j$.*
Let $M_j = C\backslash Y_j$ be the set of all machines that can solve the job j, $T_m = \{j \in J: m \in M_j\}$ be the set of all jobs that machine m can solve and $T(G)= \bigcup_{m\in G} T_m = \{j: \exists m \in G, j \in T_m\}$ be the set of all jobs that machines in G together can solve.
The job j is solved (by G) if $j \in T(G)$, otherwise it is unsolved (by G).
$\forall m \in G$, machine m is redundant (in G) iff $T_m \subseteq T(G_m)$. G is called minimal iff none of its machines is redundant. G is called a solution iff T(G) = J.
Let DP(C, J) be the problem of finding all minimal solutions and [C, J] be the class of all minimal solutions, i.e.,
$[C, J] = \{G \subseteq C: T(G) = J \wedge (T_m \not\subset T(G_m), \forall m \in G)\}$.
From now on we always assume that $\varnothing \neq G \subseteq C$, $C \in$ FCS and the problem DP(C, J) are given.

Proposition 2 (A criterion for G to be a generator of C based on T(G)). *$G \in Gen(C)$ iff G is a minimal solution, i.e.,*

$$G \in Gen(C) \Leftrightarrow \begin{cases} |T(G)| = N & (a_2) \\ \forall m \in G, T_m \not\subset T(G_m) & (b_2) \end{cases}$$

Proof. We just need to prove that $a_1 \Leftrightarrow a_2$ and $b_1 \Leftrightarrow b_2$. Given $G \in$ Gen(C) by Theorem 1 we have:
$\forall Y \in S_C, \exists g \in G: g \notin Y \Leftrightarrow \forall j \in J, \exists g \in G: g \notin Y_j \Leftrightarrow \forall j \in J, \exists g \in G: j \in T_g \Leftrightarrow J \subseteq T(G) \Leftrightarrow J = T(G)$ (since $T(G) \subseteq J$) $\Leftrightarrow T(G) = |J| = N$. Thus, $a_1 \Leftrightarrow a_2$.
$\forall g \in G, \exists Y \in S_C: g \notin Y \Leftrightarrow \forall g \in G, \exists j \in J: g \notin Y_j \Leftrightarrow \forall g \in G, \exists j \in J: (j \in T_g, j \notin T(G_g))$
$\Leftrightarrow \forall m \in G, T_m \not\subset T(G_m)$. Thus, $b_1 \Leftrightarrow b_2$. □

Theorem 2. *Gen(C, S_C) is equivalent to DP(C, J(C)), where each item is a machine and Gen(C) = [C, J(C)].*

Proof. It is a consequence of and Proposition 2. □
Example 2. Given C and S_C as in Figure 2. Let's check some candidate to be generator of C.
Here, J = <u>123</u>. We have $M_1 = C\backslash Y_1$= <u>24</u>, M_2 = <u>23</u>, M_3 = <u>1</u>. T_1 = <u>3</u>, T_2 = <u>12</u>, T_3 = <u>2</u>, T_4 = <u>1</u>, $T_5 = \varnothing$; T(<u>12</u>) = $T_1 \cup T_2$ = <u>123</u>, T(<u>23</u>) = <u>12</u>, T(<u>124</u>) = <u>123</u>, T(<u>134</u>) = <u>123</u>, ….
Let's consider G = <u>23</u>. By (a_2), it is not a generator (of C) since T(<u>23</u>) $\subset$ J.
Let's consider G = <u>124</u>. Since T(G) = J, by (a_2), G is a solution. Let's check (b_2). 1 is not redundant in G because T_1 = <u>3</u> $\not\subset$ T(G_1) = T(<u>24</u>) = <u>12</u>. By the same way, 2 is not redundant. However, 4 is redundant in <u>124</u> because T_4 = <u>1</u> $\subset$ T(<u>12</u>) = <u>123</u>. Thus, <u>124</u> is not a generator.
Let's consider G = <u>12</u>. Since T(G) = J, G is a solution. Furthermore, 1 is not redundant in G because T_1 = <u>3</u> $\not\subset$ T(G_1) = T(<u>2</u>) = <u>12</u>. By the same way, 2 is not redundant. Thus, by (b_2), <u>12</u> is minimal. Then it is a generator.
…
Eventually, Gen(C) = {<u>12</u>, <u>134</u>}.

Remark 1. Note that Proposition 2 is simpler than Theorem 1 since (a_2) is simpler than (a_1). Now, the main task is to test for G to be minimal. This test can take about |G| × (|G| - 1) times computing the union of two sets (for each m ∈ G, compute $T(G_m)$). Each union takes a complexity of O(|T.|), where |T.| is the average number of jobs that a machine can solve. Thus, the complexity of this test is O(|T.| × $|G|^2$).

3.2. Effectively solving DP(C, J)

In our approach, all minimal solutions can be found by the following backtracking manner: initialize G as an empty set, we gradually add new machine into it so that the minimal property of the updated G is always preserved. This means each G in the underlying search tree is minimal and supersets of any redundant candidates will never be explored. The more machine is added, the more tasks are solved. As a result, a minimal solution will be found when no more jobs are left.

As the principles of algorithm GDP in the next Section also allow to keep away from considering duplicated candidates and useless ones that are surely not in [C], this section is to present an efficient approach to check for a candidate G to be minimal.
Recall that G is minimal iff no m in G is redundant, i.e., $T_m \not\subset T(G_m)$ or $|T_m \backslash T(G_m)| > 0$. However, it is not necessary to perform this check for all machines in G since there are machines irrelevant to the newly added one, i.e., this check on such machines will also returns true for the new G. Thus it is better to maintain the values of $|T_m \backslash T(G_m)|$, for each m in G and try to update just the ones needed, then check only on them. By this way, the current step can make use of the computations on previous steps. The following U, F functions and the recurrent expression in Proposition 4 allow realizing this idea.

From now on we always assume that the given itemset G (⊆ C ∈ FCS) is minimal and each machine is a unique positive integer.

Definition 3. *Denote $2^{C^*} \equiv 2^C \backslash\{\emptyset\}$. Given a function U: C x $2^{C^*} \rightarrow$ Z such that, ∀m ∈ C, if m ∈ G then U(m, G) = $U_{mG} = |T_m \backslash T(G_m)| = |\{j \in T_m: (\forall b \in G_m: j \notin T_b|$; otherwise U_{mG} = -1.*
If m ∈ G then U_{mG} is the number of jobs uniquely solved by m, among the machines in G.

Definition 4. *Given a function F: J x $2^{C^*} \rightarrow$ Z such that ∀j ∈ J,*

$$F(j,G) = F_{jG} = \begin{cases} 0, & if\ j \notin T(G) \\ m, & if\ \exists\,!m \in G : j \in T_m \\ -2, & if\ (\exists m, b \in G : (m, b \in M_j) \wedge (m \neq b) \end{cases}$$

If F_{jG} = 0 then the job j is unsolved (by G); if F_{jG} = m > 0 then m is the unique machine in G that can solve the job j; if F_{jG} = -2 < 0 then there are at least two machines in G can solve the job j.
Let $F_G = \{F_{jG}, \forall j \in J\}$, $U_G = \{U_{mG}, \forall m \in C\}$.

Remark 2. $U_{mG} = 0 \Leftrightarrow T_m \backslash T(G_m) = \emptyset \Leftrightarrow T_m \subseteq T(G_m) \Leftrightarrow$ m is redundant (in G).

From Proposition 2 and Remark 2 we have the following proposition:

Proposition 3. (A criterion for G to be a generator of C based on F_G and U_G). *∀C ∈ FCS, ∀G: ∅ ≠ G ⊆ C,*

$$G \in Gen(C) \Leftrightarrow \begin{cases} T(G) = N \\ \forall m \in G, U_{mG} \neq 0 \qquad (b_3) \end{cases}$$

Proof. We just need to prove (b_2) ⇔ (b_3). Given G ∈ Gen©. ∀m∈ G, we have: $T_m \not\subset T(G_m) \Leftrightarrow T_m \backslash T(G_m) \neq \emptyset \Leftrightarrow U_{mG} \neq 0$. Thus, ($b_2$) ⇔ ($b_3$) □
After adding a new machine b into a given minimal candidate G, all new values of U_G and F_G must be available to check if G is minimal by (b_3). However, instead of recalculating all these values by Definition 3, we just need to recurrently update those element values related to T_b as the following proposition.

Proposition 4 (recurrently calculate U_G and F_G). *∀C ∈ FCS, G ⊆ C, b∈ C\G, NG = G ∪ {b}, m ∈ C, j ∈ J, we have:*
F_{jNG} = b if F_{jG} = 0 and j ∈ T_b; *(a_4)*
F_{jNG} = -2 if F_{jG} = m > 0 and j ∈ T_b; *(b_4)*
F_{jNG} = F_{jG} for other cases (F_{jG} = < 0 or j ∉ T_b). *(c_4)*
$U_{bNG} = |\{j \in T_b: F_{jG} = 0\}| = |\{j \in T_b: F_{jNG} = b\}|$; *$d_4$)*
∀m ∈ G: (∃j ∈ T_b: F_{jG} = m), $U_{mNG} = U_{mG} - |\{j \in T_b: F_{jG} = m\}|$; *($e_4$)*
$U_{mNG} = U_{mG}$ for other cases. *(f_4)*

Proof. It is obvious that ∀j ∉ T_b, F_{jNG} = F_{jG} (i).
∀j ∈ T_b, we have:
$F_{jG} = 0 \Rightarrow j \notin T(G) \Rightarrow \exists!b \in NG: j \in T_b \Rightarrow F_{jNG}$ = b. We have (a_4);
$F_{jG} = m > 0 \Rightarrow \exists!m \in G: j \in T_m$ and $m \prec_{NG} b \Rightarrow F_{jNG}$ = -m. We have (b_4);
$F_{jG} = -2 \Rightarrow F_{jNG} = -2 = F_{jG}$ (ii). From (i) and (ii) we have (c_4).
$U_{bNG} = |T_b \backslash T(G)| = |\{j \in T_b: j \notin T(G)\}| = |\{j \in T_b: F_{jG} = 0\}| = |\{j \in T_b: F_{jNG} = b\}|$. We have ($d_4$).

$\forall m \in G, m \neq b$, $T_m \backslash T(NG_m) = T_m \backslash T(G_m \cup \{b\}) = T_m \backslash T(G_m) \backslash T_b$. Thus, $U_{mNG} = |T_m \backslash T(NG_m)| = |T_m \backslash T(G_m)| - |T_b \cap [T_m \backslash T(G_m)]| = U_{mG} - |\{j \in T_b: F_{jG} = m\}|$ (iv). We have (e_4);
$\forall m \notin NG$, $U_{mNG} = U_{mG} = -1$. By (iv) we have $\forall m \in G$, $(\{j \in T_b: F_{jG} = m\} = \emptyset) \Rightarrow U_{mNG} = U_{mG})$. Thus, we have ($f_4$).

Remark 3. Proposition 4 points out a less costly way to update the values of U_G and F_G as well as which are needed to be updated. Updating the values of F_{NG} is very straightforward. For the values of U_{NG}, we just need to care about the cases (d_4) and (e_4). Especially, in case (e_4), we only update U_{mNG} when there is j in T_b such that $F_{jG} = m$. In other words, we just update U_{mNG} if machine m is relevant to the new added machine b. Such machines are the only ones that might become redundant. Thus, only in these cases, if $U_{mNG} = 0$ then m is redundant. For other cases U_{mNG} is equal to U_{mG}, thus, we do nothing.

In reality, Proposition 4 can be implemented without any expensive computations on sets (such as union, intersection, difference or inclusion) but just a loop through T_b to compute on scalar values. The complexity of this test is just $O(|T_b|)$, (see procedure Update in Figure 5), which is significantly reduced as compared to the analysis in Remark 1. Thus, Proposition 4 simplifies Proposition 3.

Example 3. Given C and $S_C = \{Y_1, Y_2, Y_3\}$ in Figure 2. Let's illustrate the way a generator of C is generated.
With $G = \underline{1}$, $\exists!1 \in G$ and $T_1 = \underline{3} \Rightarrow U_{1G} = 1$, $U_{2G} = U_{3G} = U_{4G} = U_{5G} = -1$, $F_{1G} = F_{2G} = 0$, $F_{3G} = 1$. Since there exists such $F_{1G} = 0$, by (a_3), $G \notin$ Gen©.
Add machine b = 2 into G we have: $NG = G \cup \underline{2} = \underline{12}$ and $T_b = \underline{12}$. By (a_4), $F_{1G} = F_{2G} = 0$, then $F_{1NG} = F_{2NG} = 2$. By (c_4), $3 \notin T_2$, then $F_{3NG} = F_{3G} = 1$. By (f_4), $\{j \in T_2: F_{jG} = 1\} = \emptyset$, then $U_{1NG} = U_{1G} = 1$. By (d_4), $F_{1G} = F_{2G} = 0$, then $U_{2NG} = 2$. The machines 3, 4, 5 are not in NG, then, by (f_4), $U_{3NG} = U_{4NG} = U_{5NG} = -1$. By ($a_3$) and ($b_3$), all jobs are solved and $NG = \underline{12}$ is minimal, thus $NG \in$ Gen©.
Assumed that we continue to add machine 3 into NG. We have $G' = NG \cup \{3\} = \underline{123}$, $T_3 = \underline{2}$. By (c_4), 1 and $3 \notin T_3$, then $F_{1G'} = 2$, $F_{3G'} = 1$. By (b_4), $2 \in T_3$ and $F_{2NG} = 2$, then $F_{2G'} = -2$. By (f_4), $\{j \in T_3: F_{jNG} = 1\} = \emptyset$, then $U_{1G'} = 1$. By (f_4), $U_{2NG} - |\{j \in T_3: F_{jNG} = 2\}| = U_{2NG} - |\underline{2}| = 1$. By ($d_4$), $T_3 = \underline{2}$ and $F_{2G'} = -2 \neq 2$, then $U_{3G'} = 0$ or machine 3 is redundant in G'. By (b_3), $G' = \underline{123} \notin$ Gen©.

4. The proposed algorithm GDP

In this section we introduce the dynamic programming algorithm GDP which applies Proposition 3 and Proposition 4 to find all generators for each FCI in a lattice of FCIs. It only skips any of the following useless considerations: supersets of non-minimal candidate, duplicated candidates and the ones that are surely not in [C] based on the following principles:

a. Candidate G is generated by backtracking and depth-first manner. At the beginning, G is initialized as an empty set, U_G and F_G are accordingly initialized as {-1, ..., -1} and {0, ..., 0}. When one by one adding new machine b into G, new candidate NG ($NG = G \cup \{b\}$) must also be minimal to be extended further; otherwise, NG and its supersets are pruned. Concurrently, U_G and F values are updated by Proposition 4.
b. Newly added machine b must be to solve a new job j that hasn't yet been solved by G; otherwise b will be redundant in NG. Thus, b must be selected from such an M_j that $F_{jG} = 0$. By this way b will never be collapsed to any machine in G ($\forall j \in J$: $F_{jG} = 0 \Leftrightarrow M_j \cap G = \emptyset \Rightarrow \forall b \in M_j$, $j \in T_b \backslash T(G)$, i.e., b is not redundant in NG). If there is no more such job j then G is a generator since in this case it is a minimal solution. Therefore, the search for such j is also the test for G to be a solution.
c. When new machine b is added, by Proposition 4, it is only necessary to follow cases (a_4), (b_4), (d_4) and especially case (e_4) to update the related values of U_G and F_G. Because all these cases require the condition "$j \in T_b$" is satisfied, it is better to loop through T_b to perform them. Furthermore, for each j in T_b, such machine m in case (e_4) is extracted by expression "$F_{jG} = m$, where $m > 0$". Then, $U_{mNG} = U_{mNG} - 1$ and just only at this time, it is necessary to test $U_{mNG} = 0$ to verify if b causes m redundant in NG. If so, we restore these values.
d. To avoid duplication, no superset of G is allowed to contain b, except NG and supersets of NG. Similarly, to avoid useless steps, b shouldn't be added again into any superset of NG. Thus, before processing NG, b is temporary removed from M_k, for all unsolved jobs k that b can solve. (Every such M_k is restored after all machines in M_j

are tried, so no generator is missed). This is also to reduce the sub search spaces as any M_k can shrink. As a result, no candidate is considered more than once.

By level by level applying these principles, not only all duplicated candidates and the surely non-minimal ones are eliminated from consideration but also all candidates that are not solutions are implicitly skipped. The reason is that after all machines in an M_j (of the current unsolved job j) are tried to add into G, they are removed from M_k (of any unsolved job k). Thus, no superset of G generated after this time can solve job j, i.e., they could not be solutions. By adding at least one machine from M_j into G, GDP automatically pruned these unuseful sets.

Our approach is presented via the algorithms GDP, Try and Update in Figure 3, Figure 4 and Figure 5 respectively.

4.1. The algorithm GDP

In GDP, we first calculate M_j for each job j and T_m for each machine m (lines 3, 4). Then, a set composed of a machine m will be a generator if m can solve all jobs and it will be removed permanently to reduce the search space (line 5 to line 7). After that, we initialize G, F_G, U_G and call Try to begin the backtracking process (line 8, 9).

```
Input: $L_A$ – lattice of FCIs.
Output: Gen(C), $\forall C \in L_A$.
Method: GDP($L_A$).
1.  $\forall C \in L_A$
2.      Gen(C) = $\varnothing$;
3.      $\forall j \in J$, $M_j = C \backslash Y_j$;
4.      $\forall m \in C$, $T_m = \{j \in J: m \in M_j\}$;
5.      $\forall m \in C$: if ($|T_m| = |S_C|$) then
6.          Gen(C). Add({m});
7.          $\forall j \in T_m$, $M_j$.Remove(m);
8.      G = $\varnothing$; F = {0,…,0}; U = {-1,…,-1};
9.      Try (C, G, F, U, 0);
```

Figure 3: Algorithm GDP.

4.2. The algorithm Try

```
Input: C – a FCI,
       G - current minimal candidate,
       U - $U_G$,
       F - $F_G$,
       i - starting index to find the first unsolved
       job.
Output: Gen(C).
Method: (C, G, F, U, i).
```

```
1.  If ($\exists j \in J$: $(j > i) \wedge F_{jG} = 0$) then
2.      MList = {k:($\exists b \in M_j$: $k \in T_b \wedge F_{kG} = 0$)};
3.      $\forall k \in$ MList, backup $M_k$;
4.      $\forall b \in M_j$,
5.          G.Add(b);
6.          Backup those values of $F_G$ and $U_G$
            related to $T_b$;
7.          Minimal = Update(U, F, G, b, j);
8.          $\forall k \in T_b$: $F_{kG} = 0$,
9.              $M_k$.Remove(b);
10.         if(Minimal = true) then
11.             Try(C, G, F, U, j);
12.         G. Remove (b);
13.         Restore those values of $F_G$ and $U_G$
            related to $T_b$;
14.     $\forall k \in$ MList, Restore $M_k$;
15. else: Gen(C).Add(G);
```

Figure 4: Algorithm Try.

The procedure Try tries to extend the current minimal candidate G in backtracking and depth-first manner until G can solve all jobs. It first searches for the next unsolved job j. If not so, the current candidate G is a generator (line 15); otherwise, each machine in M_j is one by one add into G and G is extend further if new machine does not cause redundancy (lines 11). Then, every set M_k of every unsolved job k that can be solved by any machine in M_j is saved (lines 3). The reason is that each time a machine b is added into G, it is temporarily removed from such M_k (lines 8, 9) to reduce the search space and to avoid replicated candidates. However, after trying all machines in M_j, these M_k must be restored (line 14) for backtracking purpose. Similarly, before provoking the function Update to update those values of F_G and U_G relevant to b and to check for $G \cup \{b\}$ to be redundant, these values are saved (line 6). After considering all candidates containing $G \cup \{b\}$ or finding out that $G \cup \{b\}$ is redundant, G, F_G and U_G are restored (line 12, 13).

4.3. The algorithm Update

```
Input:  U - $U_G$,
        F - $F_G$,
        G - the current minimal candidate,
        b - the new added machine,
        j - the first unsolved job.
Output: U – updated $U_G$ ,
        F- updated $F_G$,
        Returns true if $G \cup \{b\}$ is minimal;
        otherwise returns false.
Method: Update(U, F, G, b, j).
```

```
1.  $U_{bG} = 0;$
2.  $\forall k \in T_b,$
3.      if ($F_{kG} = 0$)
4.          $F_{kG} = b;$
5.          $U_{bG} = U_{bG} + 1;$
6.      else If ($F_{kG} > 0$)
7.          $m = F_{kG};$
8.          $U_{mG} = U_{mG} - 1;$
9.          If ($U_{mG} = 0$) return false;
10.         $F_{kG} = -2;$
11. return true;
```

Figure 5: Algorithm Update.

The algorithm Update applies Proposition 4 to update the values of F_G and of U_G that are related to T_b, where b is the new added machine and to verify if the updated G (i.e., G ∪ {b}) is minimal. It first initializes U_{bG} as 0 (line 1) and then loop through each job k that b can solve. In case $F_{kG} = 0$, b will be the first machine to solve job k, then U_{bG} is increased by 1 (line 4, 5). In case $F_{kG} = m > 0$, k will no longer be the job uniquely solved by m because b will be the second machine to solve it, then U_{mG} is decreased by 1 (line 8) and F_{kG} is set to -2 (line 10). At this time, if U_{mG} is zero, the function will return false to indicate that b causes a machine in G redundant (line 9). If no machine is redundant in G, the function will return true. The complexity this function is $O(|T_b|)$.

4.4. The complexity of GDP

Because Try(0) is the main operation in GDP, the complexity of GDP can be estimated to that of Try(0) × |FCS|. Moreover, Update is the main operation of the function Try and its complexity is O(|T.|), where |T.| is the average estimated number of jobs that a machine can solve. In the worst case, we must loop through all jobs and try choosing machines for each job. Thus, the complexity of Try(0) is evaluated to a linear degree of O(|J| × |M.| × |T.|), where |M.| is the average estimated number of machines that can solve a job. Denote A_J as the average size of J, with regard to a given FCI C (i.e., the average number of intermediate closed subsets of C). Then, the complexity of GDP can be estimated to O(A_J × |M.| × |T.| × |FCS|). Since A_J × |M.| × |T.| is too small in comparison to |FCS|, GDP can be considered as linear to |FCS|.

Example 4. Given C and S_C as in Figure 2. Let's compute Gen(C).

First we initialize G = ∅, F_G = {0, …, 0}, U_G = {-1, …,-1}. Since $M_3 = \underline{1}$ (M_3 contains only one machine), machine 1 is added into G first to solve job 3 and 1 is removed from M_3. Now, $F_{3G} = 1$, $U_{1G} = 1$. Let's try machine for the first unsolved job. With j = 1 we have $M_1 = \underline{24}$. After adding machine 2 into G, we have $U_{1G} = 1$, $U_{2G} = 2$, $U_{3G} = U_{4G} = U_{5G} = -1$, $F_{1G} = F_{2G} = 2$, $F_{3G} = 1$. G = $\underline{12}$ is a generator since no more job is left. After removing machine 2 from M_1 and M_2, we have $M_1 = \underline{4}$, $M_2 = \underline{3}$. Then, we restore G into $\underline{1}$ and try another machine for job 1. Since M_1 and M_2 each contains only a machine, we just add 4 and 3 into G without the need to check for redundancy. By the same reason, G = $\underline{143}$ is another generator. Thus, Gen(C) = {$\underline{12}$, $\underline{134}$}.

As one can see in this example, the process to generate Gen(C) is much simpler than the previous examples and there are no computations on set such as union, intersection, and difference… are needed.

5. Experiments

In this section we compare the time for mining all generators of GDP to those of MinimalGenerators (MG), Compute_hs_mingen (HS), Touch (TG) and to the time for mining all FCIs by Charm on several experimental datasets, for different minimum supports. Dense datasets (DB) include MushRoom (M), Connect (Co), C73d10k (C73), C20d10k (C20), T40i10d100K (T40) and sparse ones are Retail (R), T20i6d100K (T20) [15, 16]) thresholds (MS). All algorithms are tested on a laptop HP Compaq 6520s Intel(R) Core 2 Duo CPU T7250 @2GHz 1GB of RAM, running in Windows XP. The running time of these algorithms on all testing datasets are shown in Table 1.

Table 1: Running time of GDP, MG, HS, TG and Charm on our test cases.

DB _MS (%)	#G	MCT (s)	GT (s)			
			GDP	MG	HS	TG
M_1	103516	3.5	0	oT	28	*2.7*
M_0.5	164525	5.3	0.3	oT	43	*4.2*
M_0.1	360165	8.9	1.7	oT	97	*8.7*
Co_70	35875	7.2	0	oT	7.8	*5.9*
Co_60	68349	16.7	0.3	oT	17	*8.5*
Co_50	130101	29.2	1	oT	33.8	*13.3*
Co_40	239372	51.7	2.1	oT	66.5	*21.2*
C2_0.5	170259	4.7	0.9	oT	38	*3.5*
C2_0.1	449352	11	3	oT	97.8	*9.6*

C2_0.05	604013	17.4	4	oT	533	*15.4*
C2_0.001	823633	23.2	5	oT	oT	*29.5*
C7_70	29007	1.8	0	oT	8.3	*2.7*
C7_60	166917	4.2	1.2	oT	60.7	*6.6*
C7_50	765448	122	8	oT	327	*oM*
T4_1	66278	31.3	0	18.4	14.4	oM
T4_0.75	498785	252	6	oT	253	oM
T4_0.5	1280246	548	52	oT	614	oM
R_0.05	19698	3	0	0.7	1.4	fR
R_0.01	191265	26.8	2	10.4	17.9	fR
R_0.005	801352	161	9	oT	117	fR
T2_0.15	249051	58.5	2	118	61.1	88.7
T2_0.1	357200	120	3	176	90.4	193
T2_0.075	457305	180	4	289	111	368
Abbreviation. DB_MS: dataset_minsup, #CS: number of FCIs, #G: number of generators, MCT: time for mining FCIs by Charm, GT: time for mining all generators, oT: out of time (more than 650 seconds), oM: out of memory, fR: failed to run.						

In experiments, we use the code of MG and HS downloaded at [13]. All MG, HS and GDP use CharmL [17], extended version of Charm, to mine the lattice of FCIs for input. Executive version of Touch, can be downloaded at [16]. Touch doesn't need to create the lattice of FCIs but it must take a minus amount of time to combine each FCI with its generators. This additional time can be considered equal to the time to create the lattice from all FCIs in MG, HS and GDP. The version of Charm in Touch runs a little bit faster than CharmL code in [17] except the cases it ran out of memory (denoted by oM) or failed to run (denoted by fR) as in Table 1. Due to this reason, the minimum time for mining FCIs, shown in column MCT in Table 1, is usually the one of Charm implemented in execute version of Touch. Every test case ran more than 650 seconds is considered to be out of time, denoted by oT. Comparisons on the running time of these algorithms on several datasets are shown on Figure 6, Figure 7, Figure 8 and Figure 9.

We just tested the algorithm MinimalGenerators for a few cases because it ran very slow on dense datasets and it was almost unresponsive to even highest testing minimum support thresholds, while others algorithms can finish in a few minutes.

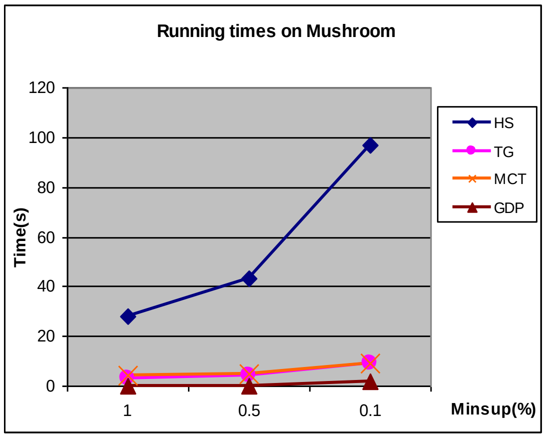

Figure 6: Running time on Mushroom.

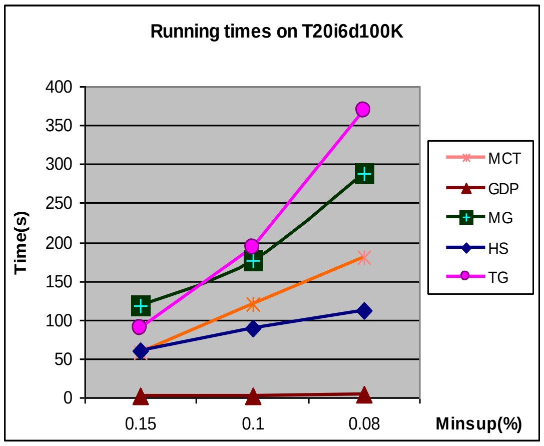

Figure 7: Running time on T20i6d100k.

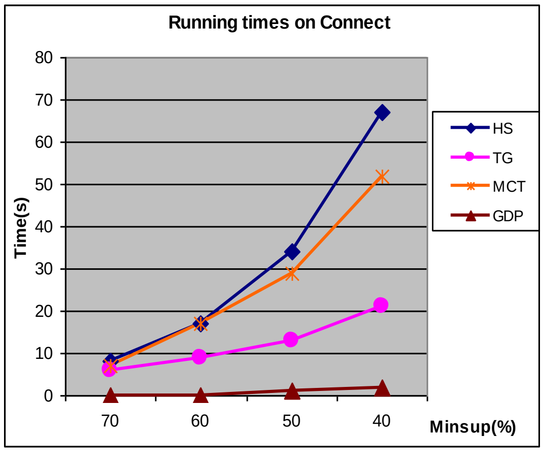

Figure 8: Running time on Connect.

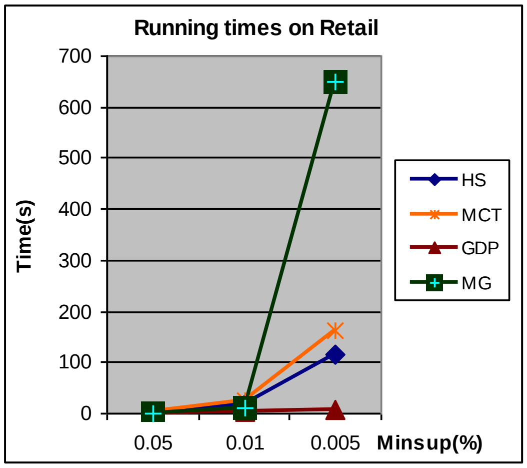

Figure 9: Running time on Retail.

Compute_hs_mingen one by one generates an itemset in a subspace of [C], with C is a FCI and check for that itemset to be minimal by the similar idea (b_3). It generates quite many redundant candidates since redundancy already happened when generating their subsets. Let |M.| be the average estimated number of machines that can process a job and A_G be the average estimated size of a candidate belonging to [C] then, the complexity of Compute_hs_mingen is $O(A_J \times |M.| \times A_G^4 \times |FCS|)$. Thus, the ratio between Compute_hs_mingen and GDP is $A_G^4/|T.|$. Moreover, as this procedure can not eliminate duplicate candidates during the search, it must perform an extra step to remove duplicate generators. It was sometimes slower than MinimalGenerators on sparse datasets. Touch worked well on most of test cases. It ran usually faster than MinimalGenerators and Compute_hs_mingen.

Experiments showed that GDP is the fastest algorithm in all test cases. When the minimum support threshold gets low, it is usually faster than Touch more than 4 times and significantly outperforms the others, especially for dense datasets. It is noteworthy that the time for mining generators and for combining them with their closure of Touch takes from about 50% to a little bit more than 100% of the minimum time for mining FCIs. In contrast, the corresponding rate of GDP is much smaller.

6. Conclusion

Since FCIs and theirs generators are essential information, they are keys to mine FIs and ARs. Therefore, it is worth to study effective approaches to mine all generators. We found out that finding generators based on lattice of FCIs, which is easy to be available, is more reasonable than mining them directly from datasets by certain mentioned reasons. By interpreting the problem of finding all generators of a FCI based on its immediate closed subsets into a problem of distributing M machines to N jobs, we derived some interesting theoretical results that turn complex criteria computing on sets to the much less costly ones that do not require any set computations. The proposed algorithm GDP can efficiently find enough generators in a low complexity without duplication and useless consideration. Experiments showed that our algorithm is more effective than the compared competitors. Especially, its time for finding all generators is very minus as compare to the time for mining all FCIs.

As GDP can be implemented in parallel, in the future, we will integrate it with a parallel or distributed FCI mining algorithm and experiment them on big data.

References

[1] Agrawal, R., Imielinski, T., and Swami, N. “Mining association rules between sets of items in large databases”. Proceedings of the 1993 ACM SIGMOID Conference, Washington DC, USA, pp. 207-216. 1993.
[2] Anh, T., Tin, T., and Bac, L. “Structures of Association Rule Set”. ACIIDS 2012, LNAI 7197, Part II, pp.361–370. 2012.
[3] Hai, D., Tin, T., Bay, V. “Efficient method for mining frequent itemsets with double constraints”. Eng. Appl. of AI 27, pp. 148-154. 2014.
[4] Pasquier, N., Taouil, R., Bastide, Y., Stumme, G., and Lakhal, L. “Generating a condensed representation for association rules”. J. of Intelligent Information Systems, Vol. 24, No. 1, pp. 29-60. 2005.
[5] Tin, T., and Anh, T. “Structure of set of association rules based on concept lattice”. ACIIDS 2010, Advances in Intelligent Information and Database Systems, SCI, Vol. 283, pp. 217-227. 2010.
[6] Zaki, M.J. “Mining non-redundant association rules”. Data mining and knowledge discovery, Vol. 9, No. 3, pp. 223-248, Kluwer Academic Publishers. 2004.
[7] Goethals, Bart, and Mohammed J. Zaki. "FIMI’03: Workshop on frequent itemset mining implementations." In Third IEEE International Conference on Data Mining Workshop on

Frequent Itemset Mining Implementations, pp. 1-13. 2003.

[8] Grahne, G., and Zhu, J."Efficiently Using Prefix-trees in Mining Frequent Itemsets". In 1st Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations FIMI. 2003.

[9] Lakhal, L., and Stumme, G. "Efficient mining of association rules based on formal concept analysis". Formal concept analysis, Lecture Notes in Computer Science, Vol. 3626, pp. 180-195. Springer Berlin Heidelberg. 2005.

[10] Negrevergne, B., Termier, A., Méhaut, J., Uno, T. "Discovering Closed Frequent Itemsets on Multicore: Parallelizing Computations and Optimizing Memory Accesses". 2010 International Conference on High Performance Computing and Simulation (HPCS), pp. 521 - 528. 2010.

[11] Wang, S. Q., Yang, Y. B., Chen, G. P., Gao, Y., and Zhang, Y. "MapReduce-based Closed Frequent Itemset Mining with Efficient Redundancy Filtering". In Data Mining Workshops (ICDMW), 2012 IEEE 12th International Conference, pp. 449-453. 2012.

[12] Anh, T., Tin, T. and Bac, L. "An approach for mining concurrently closed itemsets and generators", ICCSAMA 2013, Advanced Computational Methods for Knowledge Engineering, SCI, Vol. 479, pp.355–366. 2013.

[13] http://www.cs.rpi.edu/~zaki/www-new/pmwiki.php/Software/Software#toc4, 2012.

[14] Szathmary, L., Valtchev, P., Napoli, A., Godin, R. "Efficient vertical mining of frequent closures and generators". Proceedings of the 8th international Symposium on intelligent Data Analysis: Advances in intelligent Data Analysis VIII, Vol. 5772, pp. 393-404. Springer Berlin Heidelberg. 2009.

[15] http://fimi.ua.ac.be/data/.

[16] http://coron.loria.fr/site/downloads.php.

[17] Zaki, M.J., and Hsiao, C-J. "Efficient algorithms for mining closed itemsets and their lattice structure". IEEE Trans. Knowledge and data engineering, Vol. 17, No. 4, pp. 462-478. 2005.

Pham Quang Huy, born in Nghe An, Vietnam in 1961. Educational background: B.Sc of Computer Sciences from Dalat University in 2000 and M.Sc of Computer Sciences from University of Natural Science Ho Chi Minh in 2005. Current position: lecturer at Dalat University. Research interest: Data Mining, Rough set theory.

Truong Chi Tin, born in Da Lat, Vietnam in 1961. Educational background: B.Sc of Mathematics from Dalat University in 1983 and Ph.D of Mathematics from Vietnam National University in 1990. Current position: professor at Dalat University. Research interest: Stochastic Calculus, AI, Data Mining.