Mining frequent itemsets using the N-list and subsume concepts

Article *in* International Journal of Machine Learning and Cybernetics · April 2016

DOI: 10.1007/s13042-014-0252-2

4 authors:


Bay Vo
Ho Chi Minh City University of Technology (H…


Tuong Le
Ton Duc Thang University


Frans Coenen
University of Liverpool


Tzung-Pei Hong
National University of Kaohsiung

Int. J. Mach. Learn. & Cyber. (2016) 7:253–265
DOI 10.1007/s13042-014-0252-2

ORIGINAL ARTICLE

Mining frequent itemsets using the N-list and subsume concepts

Bay Vo · Tuong Le · Frans Coenen · Tzung-Pei Hong

Received: 27 December 2013 / Accepted: 2 April 2014 / Published online: 26 April 2014

Abstract Frequent itemset mining is a fundamental element with respect to many data mining problems directed at finding interesting patterns in data. Recently the PrePost algorithm, a new algorithm for mining frequent itemsets based on the idea of N-lists, which in most cases outperforms other current state-of-the-art algorithms, has been presented. This paper proposes an improved version of PrePost, the N-list and Subsume-based algorithm for mining Frequent Itemsets (NSFI) algorithm that uses a hash table to enhance the process of creating the N-lists associated with 1-itemsets and an improved N-list intersection algorithm. Furthermore, two new theorems are proposed for determining the "subsume index" of frequent 1-itemsets based on the N-list concept. Using the subsume index, NSFI can identify groups of frequent itemsets without determining the N-list associated with them. The experimental results show that NSFI outperforms PrePost in terms of runtime and memory usage and outperforms dEclat in terms of runtime.

Keywords Data mining · Pattern mining · Frequent itemset · N-list · Subsume

This paper is a expanded version of the paper "A hybrid approach for mining frequent itemsets" [32] presented in IEEE International Conference on Systems, Man, and Cybernetics 2013.

B. Vo · T. Le (✉)
Faculty of Information Technology, Ton Duc Thang University, Ho Chi Minh City, Vietnam
e-mail: lctuong@it.tdt.edu.vn

B. Vo
e-mail: vdbay@it.tdt.edu.vn

F. Coenen
Department of Computer Science, University of Liverpool, Liverpool, UK
e-mail: coenen@liverpool.ac.uk

T.-P. Hong
Department of Computer Science and Information Engineering, National University of Kaohsiung, Kaohsiung, Taiwan
e-mail: tphong@nuk.edu.tw

1 Introduction

Itemset mining is an important problem within the field of data mining. Currently, there are many variations of itemset mining such as frequent itemset mining [1, 23], frequent closed itemset mining [25, 35], frequent weighted itemset mining [29], constrained itemset mining [5], erasable itemset mining [8, 16, 17] and so on. However, frequent itemset mining is the most popular. Frequent itemset mining plays an important role in association rule mining [2, 22, 30, 33], sequential mining [3, 4, 12, 14, 26], classification [6, 7, 18–21, 24, 28, 36, 37]. Currently, there are a large number of algorithms which effectively mine frequent itemsets. They may be divided into three main groups:

1. **Methods that use a candidate generate-and-test strategy**: These methods use a level-wise approach for mining frequent itemsets. First, they generate frequent 1-itemsets which are then used to generate candidate 2-itemsets, and so on until no more candidates can be generated. Apriori [2] and BitTableFI [11] are exemplar algorithms.
2. **Methods that adopt a divide-and-conquer strategy**: Methods that compress the dataset into a tree structure and mine frequent itemsets from this tree by using a divide-and-conquer strategy. FP-Growth [15] and FP-Growth* [13] are exemplar algorithms.

3. **Methods that use a hybrid approach**: These methods use vertical data formats to compress the database and mine frequent itemsets by using a divide-and-conquer strategy. Eclat [34], dEclat [35], Index-BitTableFI [27], DBV-FI [31], and Node-list-based method [9, 10] are some examples.

Although many solutions have been proposed in recent years, the complexity of the frequent itemset mining problem remains a challenge. Therefore more computationally efficient solutions are desirable especially in the face of ever larger datasets to be processed. Recently, Deng and Wang [9] proposed the idea of PPC-trees (Pre-order Post-order Code trees), an FP-tree like structure for holding frequent item set information. The PPC-tree outperforms FP-tree because the algorithm using PPC-tree traverses the tree one time to determine the N-List associated with frequent 1-itemset. Meanwhile the algorithm using FP-tree traverses the tree in many times. And then Deng et al. [10] proposed the PrePost algorithm for mining frequent itemsets using that structure. PrePost operates as follows. First a tree construction algorithm is used to build a PPC-tree (a data structure for holding frequent item set information). Then N-lists are generated, each associated with a 1-itemset contained in the tree. A N-list of a k-itemset is a list describing its features, it is a compact form of transaction ID list (TID list). A divide-and-conquer strategy is then used for mining frequent itemsets. Unlike FP-tree-based approaches, this approach does not build additional trees on each iteration, it mines frequent itemsets directly using the N-list concept. The efficiency of PrePost is achieved because: (i) N-lists are much more compact than previously proposed vertical structures, (ii) the support of a candidate frequent itemset can be determined through N-list intersection operations which have a time complexity of $O(m + n + k)$, where m and n are the cardinalities of the two N-lists and k is the cardinality of the resulting N-list. This process is more efficient than finding the intersection of TID lists, as used in some frequent itemset mining algorithms, because it avoids unnecessary comparisons. The experimental results in Deng et al. [10] shows that PrePost is more efficient than FP-Growth [15], FP-Growth* [13] and dEclat [35].

Song et al. [27] proposed the concept of the "subsume index". Broadly the subsume index of a frequent 1-itemset is the list of frequent 1-itemsets that co-occur with it. This method is used to enhance the performance of frequent itemset mining. Therefore, this idea has also been incorporated into the proposed NSFI algorithm.

In this paper, two new theorems for effectively determining the "subsume index" of frequent 1-itemsets based on the N-list associated with frequent 1-itemsets are also presented. Using these theorems and a theorem in Song et al. [27], we propose an effective algorithm which combines N-lists and the subsume index concept to speed up the runtime and reduce the memory usage in comparison with the original PrePost algorithm. The proposed NSFI algorithm features the following improvements: (i) Use of a hash table to speed up the process of creating the N-lists associated with frequent 1-itemsets, (ii) an improved N-list intersection procedure to determine the intersection between two N-lists, (iii) the use of the subsume concept (with two supporting theorems) to quickly identify frequent itemsets without needing to determine the N-lists associated with them.

The rest of the paper is organized as follows. Section 2 presents the basic concepts. The NSFI algorithm is presented in Sect. 3. Section 4 gives an illustration of the operation of NSFI. Then, Sect. 5 shows the results of experiments comparing the runtime and memory usage of NSFI with those of PrePost to show the effectiveness of NSFI. Finally, Sect. 6 summarizes the results and offers some future research topics.

2 Basic concepts

2.1 Notations

DB	The dataset
DB_e	An example transaction dataset
$minSup$	A given threshold
X	An itemsets
$\sigma(X)$	The support of an itemset X
k-itemset	A itemset with k items
$\mathcal{R}$	The root of the PPC-tree
N_i	A node in PPC-tree
C_i	The PP-code of node N_i
$NL(A)$	The N-list of item A
$Subsume(A)$	The subsume index of item A

2.2 Frequent itemsets

We assume a dataset DB comprised of n transactions such that each transaction contains a number of items belonging to I where I is the set of all items in DB. An example transaction dataset, DB_e, is presented in Table 1 which will

Table 1 An example transaction dataset DB_e

Transaction	Items
1	a, b
2	a, b, c, d
3	a, c, e
4	a, b, c, e
5	c, d, e, f
6	c, d

Fig. 1 The PPC-tree construction

```
procedure Construct_PPC_tree(DB, minSup)
1.scan DB to find I1 and their frequency
2.sort I1 in frequency descending order
3.create H1, the hash table of I1
4.create the root of a PP-tree, ℛ, and label it as 'null'
5.let threshold = ⌈minSup × n⌉
6.for each transaction T ∈ DB do
7.    remove the items that their supports do not satisfy the threshold
8.    sort its 1-itemsets in frequency descending order
9.    Insert_Tree(T, ℛ)
10.traverse PP-tree to generate pre and post values associate with
each node
11.return ℛ, I1, H1 and threshold

procedure Insert_Tree(T, ℛ)
1.while (T is not null) do
2.    t ← the first item of T and T ← T \ t
3.    if ℛ has a child N such that N.name = t then N.frequency++
4.    else create a new node N with N.name = t, N.frequency = 1 and
ℛ.childnodes = N
5.    Insert_Tree(T, N)
```

be used for illustrative purposes throughout the remainder of this paper. The support of an itemset X, denoted by $\sigma(X)$ where $X \in I$, is the number of transactions in DB which contain all the items in X. An itemset X is a "frequent itemset" if $\sigma(X) \geq \lceil minSup \times n \rceil$, where $minSup$ is a given threshold. Note that a frequent itemset with k elements is called a frequent k-itemset, and I_1 is the set of frequent 1-itemsets sorted in frequency descending order.

2.3 PPC-tree

Deng and Wang [9] and Deng et al. [10] presented the PPC-tree and the PPC-tree construction algorithm as follows:

Definition 1 (*The PPC-tree*) A PPC-tree, $\mathcal{R}$, is a tree where each node holds five values: $N_i.name$, $N_i.frequency$, $N_i.childnodes$, $N_i.pre$ and $N_i.post$ which are: a the frequent item identifier (taken from the set I), the associated frequency count, the set of children node associated with this node, the order index of this node when traversing this tree in a pre-order manner and the order index of this node when traversing this tree in post-order manner.

The PPC-tree construction algorithm is presented in Fig. 1. DB_e will be used, with $minSup = 30\ \%$, to illustrate the operation of this algorithm. First the algorithm removes all items whose frequency does not satisfy the $minSup$ threshold and sorts the remaining items in descending order of frequency (see Table 2).

Table 2 DB_e after removing infrequent 1-itemsets and sorting in descending order of frequency

Transaction	Ordered frequent items
1	*a*, *b*
2	*c*, *a*, *b*, *d*
3	*c*, *a*, *e*
4	*c*, *a*, *b*, *e*
5	*c*, *d*, *e*
6	*c*, *d*

Then the algorithm inserts, in turn, the remaining items in each transaction into the PPC-tree with respect to DB_e. Finally the algorithm traverses the full tree (Fig. 2f) to generate the required *pre* and *post* values associated with each node. The final PPC-tree is presented in Fig. 3.

2.4 N-list

Deng et al. [10] also presented the definition of the N-list concept and three theorems associated with it. We summarize these as follows:

Definition 2 (*The PP-code*) The PP-code, C_i, of each node N_i in a PPC-tree comprises a tuple of the form:

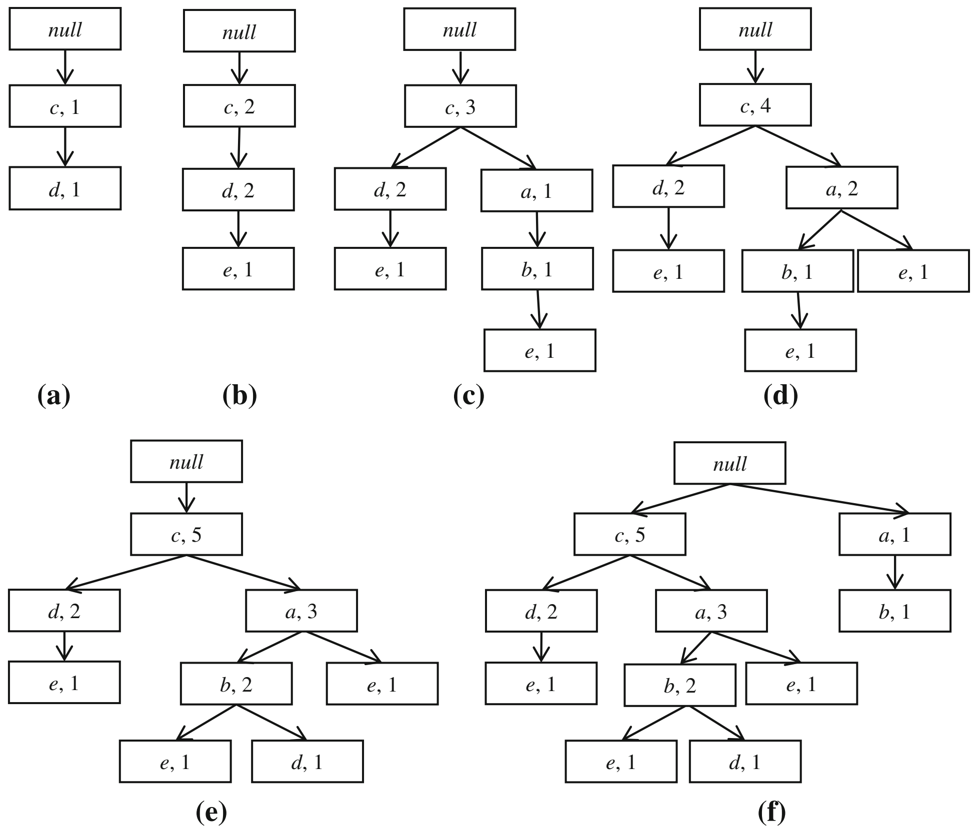

Fig. 2 Illustration of the creation of a PPC-tree using DB_e with $minSup = 30\ \%$

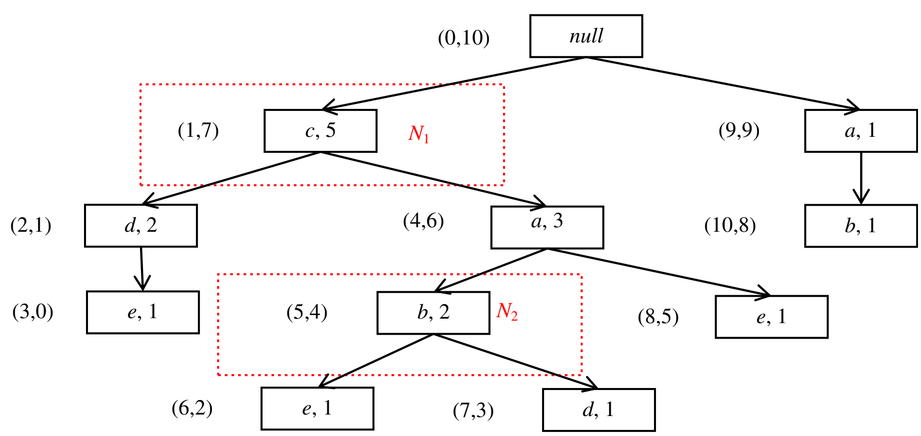

Fig. 3 The final PPC-tree created from DB_e with $minSup = 30\ \%$

$$C_i = (N_i.pre, N_i.post, N_i.frequency) \tag{1}$$

Example 1 The highlighted nodes N_1 and N_2 (for example) in Fig. 3 have the PP-codes $C_1 = \langle 1, 7, 5 \rangle$ and $C_2 = \langle 5, 4, 2 \rangle$, respectively.

Theorem 1 [10] *A PP-code C_i is an ancestor of another PP-code C_j if and only if $C_i.pre \leq C_j.pre$ and $C_i.post \geq C_j.post$.*

Example 2 According to Example 1, we have $C_1 = \langle 1, 7, 5 \rangle$ and $C_2 = \langle 5, 4, 2 \rangle$. Based on Theorem 1, C_1 is an ancestor of C_2 because $C_1.pre = 1 < C_2.pre = 5$ and $C_1.post = 7 > C_2.post = 4$.

Definition 3 (*The N-list of an item*) The N-list associated with an item *A*, denoted by *NL*(*A*), is the set of PP-codes associated with nodes in the PPC-tree whose name is equal to *A*. Thus:

$$NL(A) = \bigcup_{\{N_i \in \mathcal{R} | \mathrm{N_i.name=A}\}} C_i \tag{2}$$

where C_i is the PP-code associated with N_i.

Example 3 Let $A = \{c\}$ and $B = \{e\}$. According to the PPC-tree in Fig. 3, $NL(A) = \{\langle 1, 7, 5 \rangle\}$ and $NL(B) = \{\langle 3, 0, 1 \rangle, \langle 6, 2, 1 \rangle, \langle 8, 5, 1 \rangle\}$.

Theorem 2 [10] *Let A be an item with its N-list NL(A). The support for A, σ(A), is calculated by:*

$$\sigma(A) = \sum_{C_i \in NL(A)} C_i.frequency \quad (3)$$

Example 4 According to Example 3 we have and $NL(B) = \{\langle 3,0,1\rangle, \langle 6,2,1\rangle, \langle 8,5,1\rangle\}$. Therefore, $\sigma(A) = 5$ and $\sigma(B) = 1 + 1 + 1 = 3$.

Definition 4 (*The N-list of a k-itemset*) Let XA and XB be two $(k-1)$-itemsets with the same prefix X such that A is before B according to the I_1 ordering. $NL(XA)$ and $NL(XB)$ are two N-lists associated with XA and XB, respectively. The N-list associated with XAB is determined as follows:

1. For each PP-code $C_i \in NL(XA)$ and $C_j \in NL(XB)$, if C_i is an ancestor of C_j, the algorithm will add $\langle C_i.pre, C_i.post, C_j.frequency\rangle$ to $NL(XAB)$.
2. Traversing *NL(XAB)* to combine the PP-codes which has the same *pre* and *post* values.

Example 5 According to Example 4 we have $NL(A) = \{\langle 1,7,5\rangle\}$ and $NL(B) = \{\langle 3,0,1\rangle, \langle 6,2,1\rangle, \langle 8,5,1\rangle\}$. Therefore $NL(AB) = \{\langle 1,7,1\rangle, \langle 1,7,1\rangle, \langle 1,7,1\rangle\} = \{\langle 1,7,3\rangle\}$.

Theorem 3 [10] *Let X be an itemset and NL(X) be N-list associated with X. The support of X denoted by σ(X) is calculated as follows:*

$$\sigma(X) = \sum_{C_i \in NL(X)} C_i.frequency \quad (4)$$

Example 6 According to Example 5 we have $NL(AB) = \{\langle 1,7,3\rangle\}$, therefore $\sigma(AB) = 3$.

2.5 The subsume index of frequent 1-itemsets

To reduce the search space, the concept of a subsume index was proposed in Song et al. [27] which is based on the following function:

$$g(X) = \{T \cdot ID | \, T \in DB \text{ and } X \subseteq T\} \quad (5)$$

where $T{\cdot}ID$ is the ID of the transaction T, and $g(X)$ is the set of IDs of the transactions which include all items $i \in X$.

Example 7 Let $A = \{c\}$, we have $g(A) = \{2, 3, 4, 5, 6\}$ because A exists in the transactions 2, 3, 4, 5, 6.

Definition 5 [27] The subsume index of a frequent item A, denoted by $subsume(A)$ is defined as follows:

$$subsume(A) = \{B \in I_1 | g(A) \subseteq g(B)\} \quad (6)$$

Example 8 Let $A = \{e\}$ and $B = \{c\}$, we have $g(A) = \{3, 4, 5\}$ and $g(B) = \{2, 3, 4, 5, 6\}$. Because $g(A) \subseteq g(B)$, thus $B \in subsume(A)$.

The following theorem concerning the subsume index idea were also presented, which in turn can be used to speed up the frequent itemset mining process.

Theorem 4 [27] *Let the subsume index of an item A be {A_1, A_2,…, A_m}. The support of each of the 2^m-1 nonempty subsets of {A_1, A_2,…, A_m} combined with A is equal to the support of A.*

Example 9 Let $A = \{e\}$ and $B = \{c\}$ and according to Example 8, we have $subsume(A) = \{B\}$. Therefore 2^m-1 nonempty subsets of $subsume(A)$ is only $\{B\}$. Based on Theorem 4, the support of 2^m-1 itemset which are combined 2^m-1 nonempty subsets of $subsume(A)$ with A is equal to $\sigma(A)$. In this case, we have $\sigma(AB) = \sigma(A) = 3$. Besides, the support of the frequent itemset XA is also equal to the support of frequent itemset XAB. For example ae is a frequent itemset with $\sigma(ae) = 2$. So, aec is also a frequent itemset and $\sigma(aec) = 2$.

3 NSFI algorithm

In this section, we present the NSFI algorithm. More specifically we present the main contributions of this paper: (i) the use of a hash table to speed up the process of creating the N-lists associated with frequent items, (ii) an improved N-list intersection function, and (iii) two new theorems associated with the generation of subsume indexes, incorporated into the NSFI algorithm, that serves to both reduce the runtime and memory usage.

3.1 The N-list intersection method

Deng et al. [10] proposed the N-list intersection method, for determining the intersection of two N-lists, which has a time complexity of $O(n + m + k)$ where n, m and k is the length of the first, the second and the resulting N-lists (the method traverses the resulting N-list so as to merge the same PP-codes). In this section we present an improved N-list intersection method to give $O(n + m)$. This improved method offers the advantage that it does not traverse the resulting N-list to merge the same PP-codes.

Furthermore, we also propose an "early abandoning" strategy comprised of three steps: (i) determine the total frequency of two N-list (sF) by summing the frequencies of the first and the second N-list; (ii) for each PP-code C_i that does not belong to the result N-list, update sF = sF−C_i.*frequency*; and (iii) if sF is less than $\lceil minSup \times n \rceil$ then stop the function (the itemset currently being considered is not frequent).

Given the above the improved N-list intersection method is presented in Fig. 4.

```
function NL_intersection(PS1, PS2)
1. let PS3 ←∅, sF ← σ(PS1) + σ(PS2), i = 0, j = 0 and f = 0
2. while i < |PS1| and j < |PS2| do
3.    if PS1[i].pre < PS2[j].pre then
4.       if PS1[i].post > PS2[j].post then
5.          if |PS3| > 0 and pre value of the last element in PS3 equal
to PS1[i].pre then
6.             increase the frequency value of the last element in PS3
by  PS2[j].frequency
7.          else
8.             add the tuple ⟨PS1[i].pre, PS1[i].post, PS2[j].frequency⟩
to PS3
9.          increase f by PS2[j].frequency and increase j by 1
10.      else
11.         sF = sF - PS1[i].frequency  and increase i by 1
12.   else
13.      sF = sF - PS2[j].frequency and increase j by 1
14.   if sF < threshold then
15.      return null // stop the procedure
16.   return PS3 and f
```

Fig. 4 The improved N-list intersection method

Table 3 The comparison between the N-list intersection method [10] and the improved N-list intersection method

Step No.	The N-list intersection method [10]	The improved N-list intersection method
1	$NL(AB) = \{\}$	$NL(AB) = \{\}$
2	$NL(AB) = \{\langle 1,7,1\rangle\}$	$NL(AB) = \{\langle 1,7,1\rangle\}$
3	$NL(AB) = \{\langle 1,7,1\rangle, \langle 1,7,1\rangle\}$	$NL(AB) = \{\langle 1,7,2\rangle\}$
4	$NL(AB) = \{\langle 1,7,1\rangle, \langle 1,7,1\rangle, \langle 1,7,1\rangle\}$	$NL(AB) = \{\langle 1,7,3\rangle\}$
5	$NL(AB) = \{\langle 1,7,3\rangle\}$	

Example 10 To illustrate the improved N-list intersection method, let $A = \{c\}$ and $B = \{e\}$. According to Example 4 we have $NL(A) = \{\langle 1,7,5\rangle\}$ and $NL(B) = \{\langle 3,0,1\rangle, \langle 6,2,1\rangle, \langle 8,5,1\rangle\}$. The N-List intersection method [10] comprises four steps (column 2 in Table 3). Notably step 3 in this method traverses the resulting N-list so as to merge the same PP-codes. Vice versa, the improved N-list intersection method only requires three steps and does not traverses the resulting N-list (column 3 in Table 3).

3.2 The subsume index associated with each frequent 1-itemset based on N-List concept

Theorem 5 *Let A be a frequent item. We have:*

$$subsume(A) = \{B \in I_1 | \forall C_i \in NL(A), \exists C_j \in NL(B) such \quad that\ C_j\ is\ an ancestor \quad of\ C_i\} \tag{7}$$

Proof This theorem can be proven as follows: all PP-codes in $NL(A)$ have a PP-code ancestor in $NL(B)$, this means that all transactions that contain A also contain B. This, $g(A) \subseteq g(B)$, which implies that $B \in subsume(A)$. Therefore, this theorem is proven.

Example 11 Let $A = \{e\}$, $B = \{c\}$. We have $NL(B) = \{\langle 1,7,5\rangle\}$ and $NL(A) = \{\langle 3,0,1\rangle, \langle 6,2,1\rangle, \langle 8,5,1\rangle\}$. According to Theorem 5, $\langle 3,0,1\rangle$, $\langle 6,2,1\rangle$ and $\langle 8,5,1\rangle \in NL(A)$ are descendants of $\langle 1,7,5\rangle \in NL(B)$. Therefore, $B \in subsume(A)$.

Theorem 6 *Let A, B, C* $\in I_1$ *be three frequent items. If* $A \in subsume(B)$ *and* $B \in subsume(C)$ *then* $A \in subsume(C)$.

Proof We have $A \in subsume(B)$ and $B \in subsume(C)$ therefore $g(B) \subseteq g(A)$ and $g(C) \subseteq g(B)$. So $g(C) \subseteq g(A)$ and thus this theorems is proven.

Fig. 5 The generating subsume index proceduce

```
procedure Find_Subsume(I1)
for i ← 1 to |I1| - 1 do
   for j ← i - 1 to 0 do
      if j ∈ I1[i].Subsumes then continue
      if checkSubsume(I1[i].N-list, I1[j].N-list) is true then
         add I1[j].name and its index, j, to I1[i].Subsumes
         add all elements in I1[j].Subsumes to I1[i].Subsumes

function checkSubsume(Na, Nb)
let i = 0 and j = 0
while j < |Na| and i < |Nb| do
   if Nb[i].pre < Na[j].pre and Nb[i].post > Na[j].post then
      increase j by 1
   else increase i by 1
if j = |Na| then
   return true
return false
```

To find all frequent items associated with the subsume index of each $A \in I_1$, I_1 should be sorted in ascending order of frequency. However, I_1 has already been sorted in descending order of frequency with respect to the PPC-tree constructed previously. Therefore, in the context of the generate subsume index procedure, we propose a different traversal (see Fig. 5) to avoid the cost of this re-ordering process and also so as to facilitate the use of Theorem 6.

3.3 Main algorithm

The theorem proposed in Song et al. [27], which was re-presented in Sect. 2.5, was also adopted in the NSFI algorithm so as to speed up its operation (Fig. 6). Adoption of this theorem also helped reduce the NSFI algorithm's memory usage requirements, because it is not necessary to determine and store the N-lists associated with a group of frequent itemsets to determine their supports.

First the NSFI algorithm creates the PPC-tree, then traverses this tree to generate the N-lists associated with the frequent 1-itemset. Then, a divide-and-conquer strategy, together with the subsume index concept, is used to mine frequent itemsets. For each element X, if its subsume index has elements, the set of itemsets produced by combining X with the elements in its subsume index are frequent and has the same support as X. For each of the remaining elements that are not contained in the subsume index of X, the algorithm will combine them with X to create the frequent candidate itemsets. Note that for 2-itemsets or more, the algorithm does not use the subsume index.

4 *Example*

An illustrative example is presented in this section using DB_e with $minSup = 30\ \%$. First the NSFI algorithm scans DB_e to create the PPC-tree (Fig. 3). Then the algorithm traverses the PPC-tree to generate the N-lists associated with the frequent items (Fig. 7).

Next the algorithm finds the subsume index associated with the frequent items using the generating subsume index procedure (Fig. 5). The subume index associated with the frequent items is shown in Table 4.

Then the NSFI algorithm combines, in turn, the frequent $(k-1)$-itemsets in I_1 in reverse order using a divide-and-conquer strategy to create the k-itemset candidates. For detail, e, the last frequent 1-itemset, is used to: (i) find the 2^m-1 subsets from the m frequent 1-itemsets in $subsume(\{e\})$ and combine them with $\{e\}$ to generate the 2^m-1 frequent itemsets S. In this case, $subsume(\{e\}) = \{c\}$, therefore $S = \{ec\}$; (ii) combine, in turn, with remaining frequent items $\{d, b, a\}$ (not combined with c because $c \in subsume(\{e\})$) to create candidate 2-itemsets $\{de, be, ae\}$. However, only $\{ae\}$ is frequent, thus $FIs_{next} = \{ae\}$. Next the algorithm combines the elements in FIs_{next} with the elements in S to create further frequent itemsets without calculating their support. In this case, only $\{aec\}$ is created; and (iii) use the elements in FIs_{next} to combine together to create the candidate 3-itemsets. In this case, this algorithm will stop here because FIs_{next} has only one element (see Fig. 8).

Then, using the above strategy, the remaining frequent 1-itemsets in turn continue to create the tree which contains

Fig. 6 NSFI algorithm

```
Input: A dataset DB and minSup
Output: FIs, the set of all frequent itemsets
1. Construct_PPC_tree(DB, minSup) to generate R, I1, H1 and threshold
2. Generate_NList(R, I1)
3. Find_Subsume(I1)
4. let FIs ← I1 and Subsume ← {}
5. Find_FIs(I1, Subsumes)
6. return FIs

procedure Generate_NList(R, I1)
1. let C ← ⟨R.pre,R.post, R.frequency⟩
2. add C to H1[R.name].N-list
3. increase H1[R.name].frequency by C.frequency
4. for each child in R.children do
5.    Generate_NList(child)

procedure Find_FIs(Is, S)
1. for i ← Is.size - 1 to 0 do
2.    let FIs_next ← ∅
3.    if |Is[i].Subsumes| > 0 then
4.       let S be the set of subset generated from all elements of
Is[i].Subsumes
5.       for each s in S do
6.          add ⟨s, Is[i].frequency⟩ to FIs //using theorem 4
7.    else if |Is[i]| = 1 then
8.       S ← {}
9.    indexS = |Is[i].Subsumes| - 1
10.   for j ← i - 1 to 0 do
11.      if indexS ≥ 0 and Is[i].Subsumes[indexS] = j then
12.         indexS = indexS - 1 and continue
13.      let e_first be the first item of Is[j]
14.      FI ← {e_first} + Is[i]
15.      (FI.N-list and FI.frequency) ← NL_intersection(Is[j].N-list,
Is[i].N-list)
16.      if FI.N-list = null then continue
17.      if(FI.frequency ≥ threshold) then
18.         add FI to FIs
19.         insert FI at first in FIs_next
20.         for each subsume in S do
21.            let f = FI + subsume
22.            f.frequency = FI.frequency
23.            add f to FIs // using theorem 4
24.   Find_FIs(FIs_next, S)
```

Fig. 7 The I_1 and its N-lists on DB_e ($minSup = 30\ \%$)

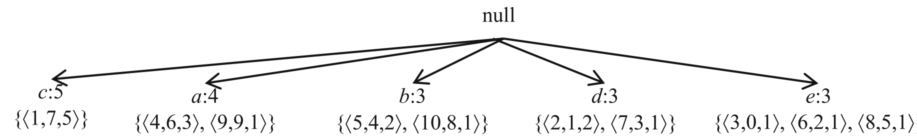

Table 4 The subsume index associated with frequent items

Frequent items	Subsume index
c	
a	
b	*a*
d	*c*
e	*c*

Table 5 Statistical summary of the experimental datasets

Dataset	#Trans	#Items
Accidents	340,183	468
Chess	3,196	76
Mushroom	8,124	120
Pumsb_star	49,046	7,117
Retail	88,162	16,470

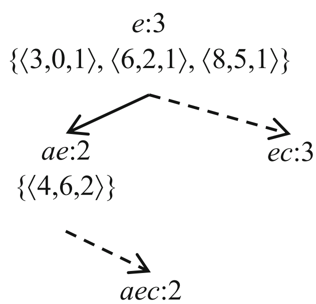

Fig. 8 The frequent itemsets generated from *e* on DB_e ($minSup = 30\ \%$)

all frequent itemsets as Fig. 9. In Fig. 9, the NSFI algorithm does not compute and store the N-lists of the nodes {*ba*, *cba*, *dc*, *ae*, *ec*, *aec*}. Therefore, using the subsume index concept it not only reduces the runtime but also reduce the memory usage.

5 Experimental results

All experiments presented in this section were performed on a laptop with Intel core i3-3110 M 2.4 GHz and 4 GBs of RAM. All the programs were coded in C# on MS/Visual studio 2012 and run on Microsoft.Net Framework Version 4.5.50709. The experiments were conducted using the following UCL datasets: Accidents, Chess, Mushroom, Pumsb_star and Retail[1]. Some statistics concerning these datasets are shown in Table 5.

We compared the proposed algorithm with PrePost and dEclat in terms of the mining time in Sect. 5.1. Then the memory usage of the proposed algorithm and PrePost were compared in Sect. 5.2. Note that:

1. Runtime in this context is the period between the start of the input to the end of the output.
2. Memory usage is determined by summing either:
   (i) the memory to store frequent itemsets, their N-lists, their frequency and the subsume index (NSFI algorithm) or
   (ii) the memory to store frequent itemsets and their N-lists and their frequency (PrePost algorithm). We assume that *pre*, *post*, *frequence* and the item identifier associated with a frequent itemset are all stored in an integer format which requires 4 bytes per integer.

5.1 The runtime

The experimental results with respect to the runtime experiments are presented in Figs. 10, 11, 12, 13 and 14. From these figures it can be observed that with datasets which have a small number of elements (or no elements) with respect to the subsume index for a frequent items such as Retail (Fig. 14), NSFI is a little slower than PrePost. This is explained as follows. Generating the subsume index involves a cost. However, the subsume index associated with each of the frequent items in a sparse datasets usually have few elements. Therefore, using the subsume concept is not effective in this case. Fortunately, this cost is usually relatively low, about 4 s for Retail dataset with $minSup = 0.1$ (0.072 % of the runtime) (see Fig. 14). Similar to Retail, Accidents (Fig. 10) also has a small number of elements with respect to the subsume index; therefore, using the subsume concept is not better. However, using a hash table to speed up the process of creating the N-lists associated with frequent items is effective; Therefore, the runtime of NSFI is much better than that of PrePost. On the other hand, given a dense datasets, the performance of NSFI

[1] Downloaded from http://fimi.cs.helsinki.fi/data/.

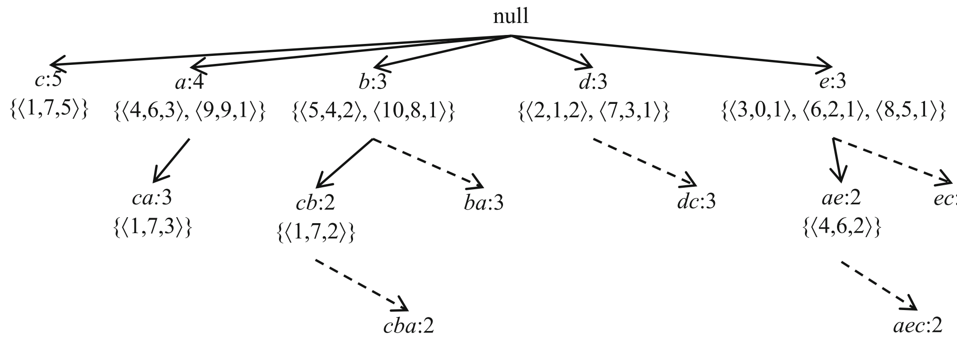

Fig. 9 All frequent itemsets on DB_e ($minSup = 30\ \%$)

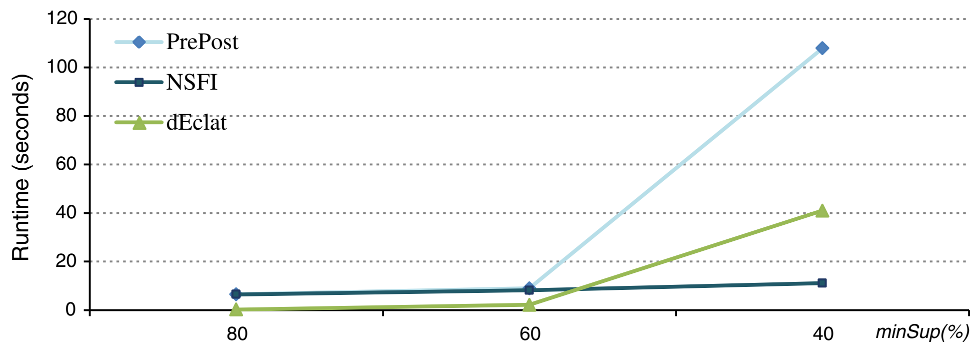

Fig. 10 The runtime using NSFI, PrePost and dEclat applied to the Accidents dataset

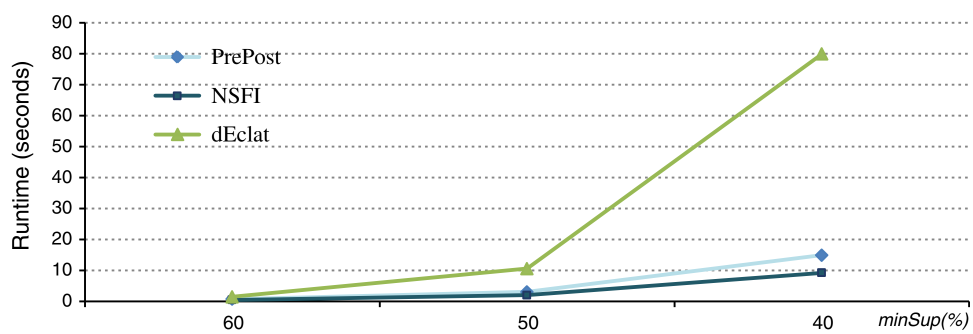

Fig. 11 The runtime using NSFI, PrePost and dEclat applied to the Chess dataset

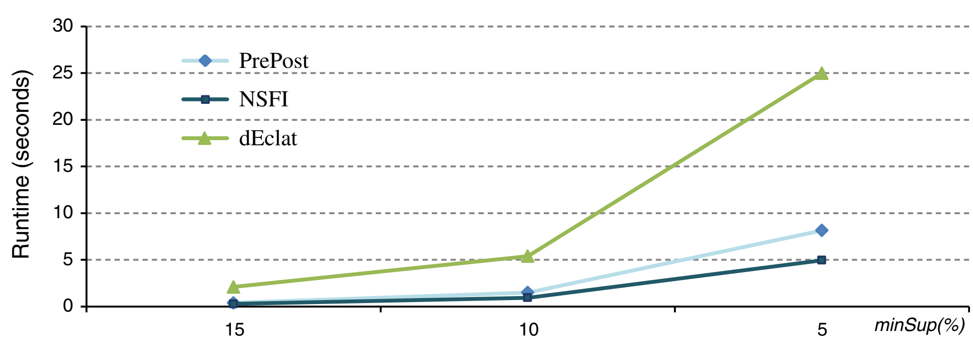

Fig. 12 The runtime using NSFI, PrePost and dEclat applied to the Mushroom dataset

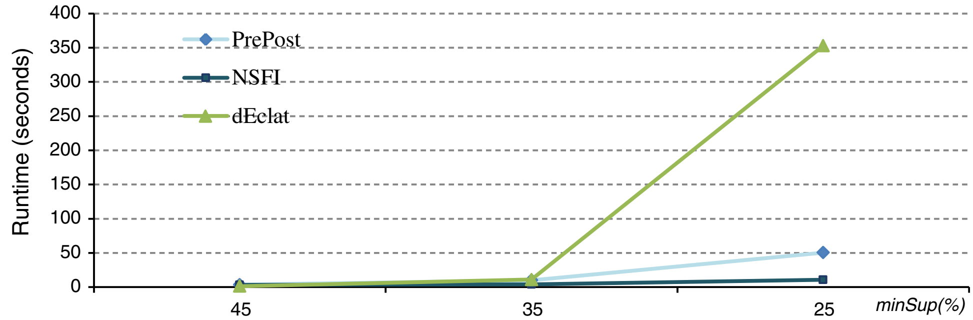

Fig. 13 The runtime using NSFI, PrePost and dEclat applied to the Pumsb_star dataset

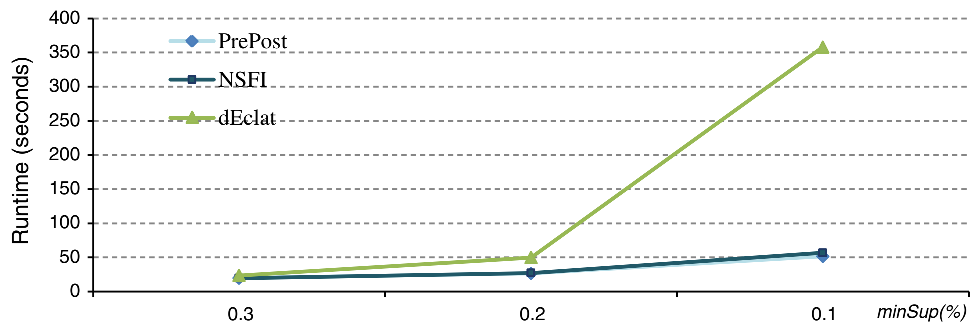

Fig. 14 The runtime using NSFI, PrePost and dEclat applied to the Retail dataset

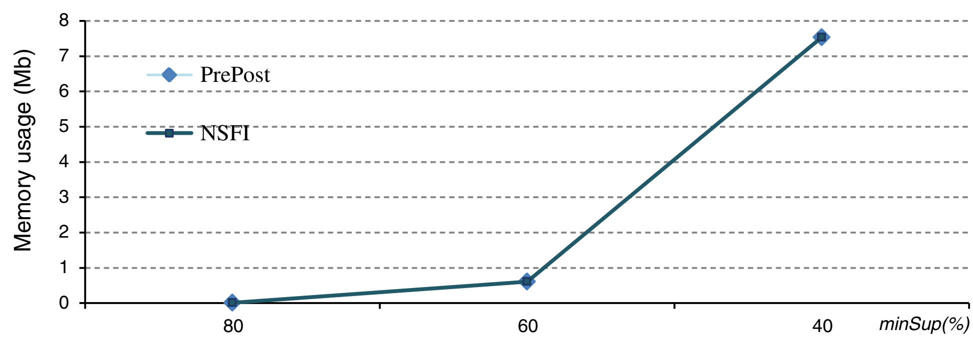

Fig. 15 The memory usage using NSFI and PrePost applied to the Accidents dataset

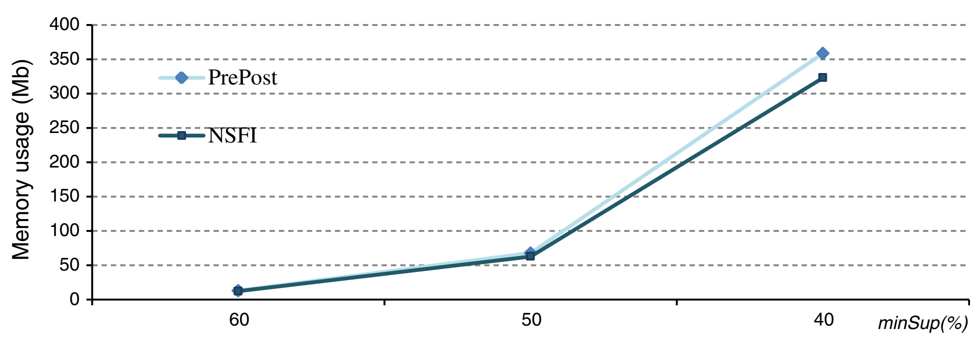

Fig. 16 The memory usage using NSFI and PrePost applied to the Chess dataset

is better than PrePost (see Figs. 11, 12, 13), especially with low thresholds. NSFI thus generally outperforms PrePost. Besides, the runtime of the dEclat algorithm is always greater than that of the NSFI algorithm. Therefore, we conclude that NSFI outperforms PrePost and dEclat in terms of the runtime.

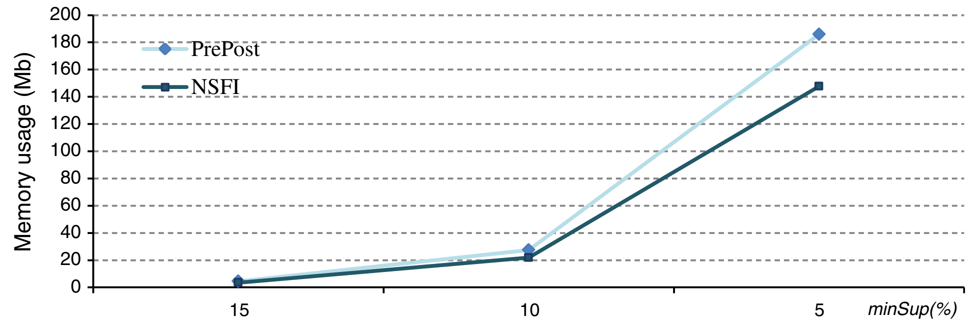

Fig. 17 The memory usage using NSFI and PrePost applied to the Mushroom dataset

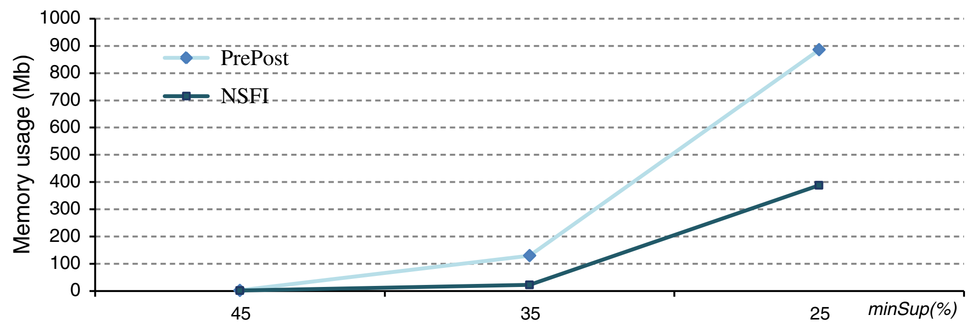

Fig. 18 The memory usage using NSFI and PrePost applied to the Pumsb_star dataset

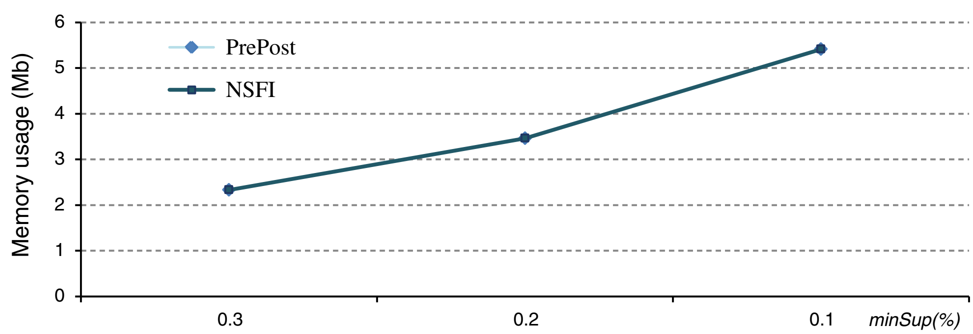

Fig. 19 The memory usage using NSFI and PrePost applied to the Retail dataset

5.2 The memory usage

The experimental results with respect to the memory usage experiments are presented in Figs. 15, 16, 17, 18 and 19. With datasets which have no elements in the context of the subsume index of the frequent items, such as Accidents and Retail, the memory usage of NSFI is equal to that of PrePost (Figs. 15, 19). However, the operation of NSFI is better than PrePost in the case of dense datasets (Figs. 16, 17, 18). Therefore, using the subsume index concept is effective with respect to dense datasets.

6 Conclusions and future work

This paper has proposed the NSFI algorithm, an effective algorithm for mining frequent itemsets using the N-list and subsume concepts. First, we proposed several improvements on the previously published PrePost algorithm: (i) use of a hash table to enhance the process of creating the N-lists associated with the frequent 1-itemsets and (ii) an improved intersection function to find the intersection between two N-lists. Then, two theorems were proposed for application with respect to the determination of the subsume index of frequent items based on the N-list concept which were incorporated into the NSFI algorithm so as to improve the runtime and memory usage. NSFI does not improve over the PrePost with respect to sparse datasets but the time gap is not significant. With respect to dense datasets NSFI is faster than PrePost. Besides, the runtime of NSFI is always faster than dEclat. We therefore conclude that NSFI generally outperforms the PrePost and dEclat.

For future work we will focus on applying the N-list concept and the hybrid approach for mining frequent closed/maximal itemsets.

Acknowledgments This research was funded by Foundation for Science and Technology Development of Ton Duc Thang University (FOSTECT), website: http://fostect.tdt.edu.vn, under Grant FOSTECT.2014.BR.07.

References

1. Agrawal R, Imielinski T, Swami AN (1993) Mining association rules between sets of items in large databases. In: Proceeding of the SIGMOD'93, pp 207–216
2. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules. In: Proceeding of the VLDB'94, pp 487–499
3. Agrawal R, Srikant R (1995) Mining sequential patterns. In: Proceeding of the 11th international conference on data engineering, Taipei, Taiwan, pp 3–14
4. Ayres J, Gehrke JE, Yiu T, Flannick J (2002) Sequential pattern mining using a bitmap representation. In: Proceeding of the SIGKDD'02, pp 429–435
5. Baralis E, Cerquitelli T, Chiusano S (2010) Constrained itemset mining on a sequence of incoming data blocks. Int J Intell Syst 25(5):389–410
6. Chen G, Liu H, Yu L, Wei Q, Zhang X (2006) A new approach to classification based on association rule mining. Decis Support Syst 42(2):674–689
7. Coenen F, Leng P, Zhang L (2007) The effect of threshold values on association rule based classification accuracy. Data Knowl Eng 60(2):345–360
8. Deng Z, Fang G, Wang Z, Xu X (2009) Mining erasable itemsets. In: Proceeding of the ICMLC'09, pp 67–73
9. Deng Z, Wang Z (2010) A new fast vertical method for mining frequent patterns. Int J Comput Intell Syst 3(6):733–744
10. Deng Z, Wang Z, Jiang JJ (2012) A new algorithm for fast mining frequent itemsets using N-lists. Sci China Inform Sci 55(9):2008–2030
11. Dong J, Han M (2007) BitTableFI: an efficient mining frequent itemsets algorithm. Knowl Based Syst 20:329–335
12. Fournier-Viger P, Faghihi U, Nkambou R, Nguifo EM (2012) CMRules: an efficient algorithm for mining sequential rules common to several sequences. Knowl Based Syst 25(1):63–76
13. Grahne G, Zhu J (2005) Fast algorithms for frequent itemset mining using FP-trees. IEEE Trans Knowl Data Eng 17:1347–1362
14. Gouda K, Hassaan M, Zaki MJ (2010) PRISM: a primal-encoding approach for frequent sequence mining. J Comput Syst Sci 76(1):88–102
15. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proceeding of the SIGMODKDD'00, pp 1–12
16. Le T, Vo B, Coenen F (2013) An efficient algorithm for mining erasable itemsets using the difference of NC-Sets. In: Proceeding of the IEEE SMC'13, pp 2270–2274
17. Le T, Vo B (2014) MEI: an efficient algorithm for mining erasable itemsets. Eng Appl Artif Intell 27(1):155–166
18. Li W, Han J, Pei J (2001) CMAR: accurate and efficient classification based on multiple class-association rules. In: Proceeding of the ICDM'01, pp 369–376
19. Lim AHL, Lee CS (2010) Processing online analytics with classification and association rule mining. Knowl Based Syst 23(3):248–255
20. Liu B, Hsu W, Ma Y (1998) Integrating classification and association rule mining. In: Proceeding of the SIGKDD'98, pp 80–86
21. Liu L, Yu Z, Guo J, Mao C, Hong X (2013) Chinese question classification based on question property kernel. Int J Mach Learn Cybern. doi:10.1007/s13042-013-0216-y
22. Lucchese B, Orlando S, Perego R (2006) Fast and memory efficient mining of frequent closed itemsets. IEEE Trans Knowl Data Eng 18(1):21–36
23. Mohamed MH, Darwieesh MM (2013) Efficient mining frequent itemsets algorithms. Int J Mach Learn Cybern. doi:10.1007/s13042-013-0172-6
24. Nguyen LTT, Vo B, Hong TP, Hoang CT (2012) Classification based on association rules: a lattice-based approach. Expert Syst Appl 39(13):11357–11366
25. Pasquier N, Bastide Y, Taouil R, Lakhal L (1999) Efficient mining of association rules using closed itemset lattices. Inform Syst 24(1):25–46
26. Pham TT, Luo JW, Hong TP, Vo B (2012) MSGPs: a novel algorithm for mining sequential generator patterns. In: Proceeding of the ICCCI'12, pp 393–402
27. Song W, Yang B, Xu Z (2008) Index-BitTableFI: an improved algorithm for mining frequent itemsets. Knowl Based Syst 21:507–513
28. Veloso A, Meira W Jr, Goncalves M, Almeida HM, Zaki MJ (2011) Calibrated lazy associative classification. Inform Sci 181(13):2656–2670
29. Vo B, Coenen F, Le B (2013) A new method for mining Frequent Weighted Itemsets based on WIT-trees. Expert Syst Appl 40(4):1256–1264
30. Vo B, Hong TP, Le B (2013) A lattice-based approach for mining most generalization association rules. Knowl Based Syst 45:20–30
31. Vo B, Le B (2011) Interestingness measures for mining association rules: combination between lattice and hash tables. Expert Syst Appl 38(9):11630–11640
32. Vo B, Le T, Coenen F, Hong TP (2013) A hybrid approach for mining frequent itemsets. In: Proceeding of the IEEE SMC'13, pp 4647–4651
33. Zaki MJ (2004) Mining non-redundant association rules. Data Min Knowl Disc 9(3):223–248
34. Zaki MJ, Parthasarathy S, Ogihara M, Li W (1997) New algorithms for fast discovery of association rules. In: Proceeding of KDD'97, pp 283–286
35. Zaki MJ, Hsiao CJ (2005) Efficient algorithms for mining closed itemsets and their lattice structure. IEEE Trans Knowl Data Eng 17(4):462–478
36. Zhang X, Chen G, Wei Q (2011) Building a highly-compact and accurate associative classifier. Appl Intell 34(1):74–86
37. Zhao S, Tsang ECC, Chen D, Wang XZ (2010) Building a rule-based classifier—a fuzzy-rough set approach. IEEE Trans Knowl Data Eng 22(5):624–638