A Hybrid Approach for Mining Frequent Itemsets

Bay Vo
Ton Duc Thang University, Ho Chi Minh City
Viet Nam
bayvodinh@gmail.com

Tuong Le
University of Food Industry, Ho Chi Minh City
Viet Nam
tuonglecung@gmail.com

Frans Coenen
Department of Computer Science, University of Liverpool, UK
coenen@liverpool.ac.uk

Tzung-Pei Hong
Department of Computer Science and Information Engineering, National University of Kaohsiung, Taiwan
tphong@nuk.edu.tw

***Abstract* — Frequent itemset mining is a fundamental element with respect to many data mining problems. Recently, the PrePost algorithm has been proposed, a new algorithm for mining frequent itemsets based on the idea of N-lists. PrePost in most cases outperforms other current state-of-the-art algorithms. In this paper, we present an improved version of PrePost 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. The experimental results show that the performance of the proposed algorithm improves on that of PrePost.**

Keywords - frequent itemset, PPC-tree, N-list, data mining

I. INTRODUCTION

Frequent itemset mining was first introduced in 1993 [1] and plays an important role in the mining of associate rules [1, 2, 7, 10]. 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 of which Apriori [2] and BitTableFI [4] are exemplar algorithms.
(2) Methods that adopt a divide-and-conquer strategy and a compressed data structure of which FP-Growth [6] and FP-Growth* [5] are exemplar algorithms.
(3) Methods that use a hybrid approach of which Eclat [11], dEclat [12] and Index-BitTableFI [8] are all examples.

Although many solutions have been proposed, the complexity of the frequent itemset mining problem remains a challenge. Therefore more computationally efficient solutions are desirable. Recently, Deng et al. [3] introduced the PrePost algorithm for mining frequent itemsets based on the idea of PPC-trees (**P**re-order **P**ost-order **C**ode trees), an FP-tree like structure. PrePost operates as follows. First a tree construction algorithm is used to build a PPC-tree. Then N-lists are generated, each associated with a 1-itemset contained in the tree. A N-list of *k*-itemset is a list describing its features, it is 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 are $O(m+n+k)$, where m, n are the cardinalities of the two N-lists and k is the cardinalities of the resulting N-list. This process is more efficient than finding the intersection of TID lists because it avoids unnecessary comparisons. The experimental results in [3] shows that the PrePost is more efficient than FP-Growth [6], FP-Growth* [5] and dEclat [12].

In this paper, we propose a hybrid algorithm based on PrePost, which features the following improvements:

(1) Use of a hash table to speed up the process of creating the N-lists associated with frequent 1-itemsets.
(2) Improving the N-list intersection procedure to determine the intersection between two N-lists.

Song et al. [8] 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 idea and the N-list concept have also been incorporated into the proposed hybrid algorithm.

The main contributions of this paper: (i) an improved N-list intersection function, (ii) two new theorems associated with the generation of subsume indexes and (iii) the usage of the two theorems proposed in [8] in the proposed algorithm to reduce the runtime and memory usage.

The rest of the paper is organized as follows. Section 2 presents the basic concepts. The proposed algorithm is proposed in Section 3 and an example of the process of this algorithm is presented in Section 4. Section 5 shows the results of experiments. Finally, the paper is concluded in Section 6 with a summary and some future research issues.

II. BASIC CONCEPTS

A. Frequent itemsets

We assume a dataset *DB* comprised of *n* transactions such that each transaction contains a number of items belong to *I* where *I* is the set of all items in *DB*. An example transaction dataset is

978-1-4799-0652-9/13/$31.00 ©2013 IEEE.

presented in Table 1 (the meaning of the third column will become clear later in this paper), this dataset will 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.

TABLE I. AN EXAMPLE TRANSACTION DATASET

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

B. PPC-tree

Deng et al. [3] presented the PPC-tree (an FP-tree like structure) 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 the frequent 1-itemset in I_1, the frequency of this node, the set of children node associated with this node, the order of this node when traversing this tree in Left-Right order and the order of this node when traversing this tree in Right-Left order respectively. Note that the root of the tree, $\mathcal{R}_{root}$, has $\mathcal{R}_{root}.name$ = "null" and $\mathcal{R}_{root}.frequency = 0$.

```
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
4.     N.frequency++
5.   else
6.     create a new node N with N.name = t , N.frequency = 1 and R.childnodes = N
7.   Insert_Tree(T, N)
```

Figure 1. The PPC-tree construction

The PPC-tree construction algorithm is presented in Figure 1. The example transaction dataset from Table 1 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 column three in Table 1). The algorithm then inserts, in turn, the remaining items in each transaction into the PPC-tree as shown in Figure 2 with respect to our example dataset.

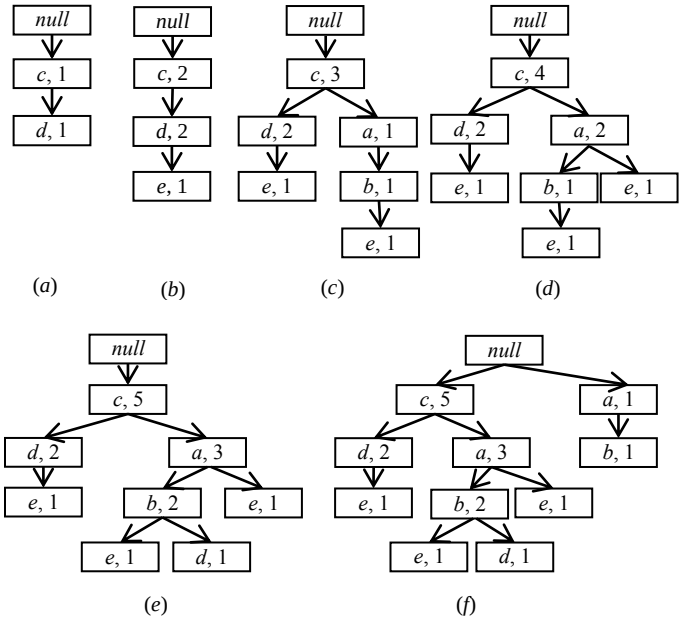

Figure 2. Illustration of the creation of a PPC-tree using the example transaction dataset with $minSup$ = 30%

Finally the algorithm traverses the full tree (Figure 2 (*f*)) to generate the required *pre* and *post* values associated with each node. The final PPC-tree is presented in Figure 3.

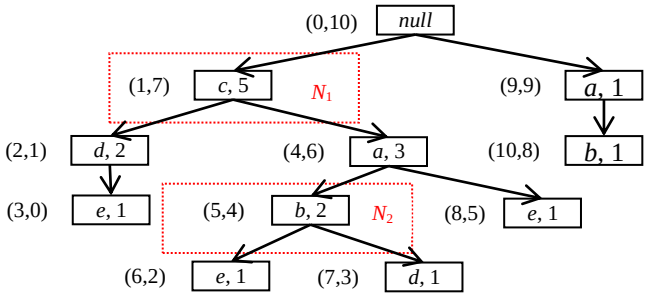

Figure 3. The final PPC-tree created from the example transaction dataset with $minSup$ = 30%

C. N-list

Deng et al. [3] 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 has a tuple as follows:

$$C_i = \langle N_i.pre, N_i.post, N_i.frequency \rangle \quad (1)$$

Example 1. The highlighted nodes N_1 and N_2 (for example) in Figure 3 have the PP-codes C_1 = ⟨1,7,5⟩ and C_2 = ⟨5,4,2⟩ respectively.

Theorem 1 [3]. 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 $C_i.post \geq C_j.post$. Note that any PP-code is also considered to be its own ancestor.

Example 2. According to *Example 1*, we have C_1= ⟨1,7,5⟩ and C_2= ⟨5,4,2⟩. 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 a frequent 1-itemset). 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:

$$NL(A) = \bigcup_{\{N_i \in \mathcal{R} \mid N_i.name=A\}} C_i \quad (2)$$

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 Figure 3, *NL*(*A*) = {⟨1,7,5⟩} and *NL*(*B*) = {⟨6,2,1⟩,⟨8,5,1⟩}.

Theorem 2 [3]. Let *A* be a 1-itemset with the associated N-list *NL*(*A*). The support for *A*, $\sigma(A)$, is calculated by:

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

Example 4. According to *Example* 3 we have *NL*(*A*) = {⟨1,7,5⟩} and *NL*(*B*) = {⟨6,2,1⟩, ⟨8,5,1⟩}. Therefore, $\sigma(A) = 5$ and $\sigma(B) = 1 + 1 = 2$.

Definition 4 (The N-list of a *k*-itemset). Let *XA* and *XB* be two (*k*-1)-itemsets with the same prefix *X* (*X* can be an empty set) 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*) = {⟨1,7,5⟩} and *NL*(*B*) = {⟨6,2,1⟩, ⟨8,5,1⟩}. Therefore *NL*(*AB*) = {⟨1,7,1⟩, ⟨1,7,1⟩} = {⟨1,7,2⟩}.

Theorem 3 (The support of a *k*-itemset) [3]. Let *X* be an itemset and *NL*(*X*) be N-list associated with *X*. The support of *X* denoted by $\sigma(X)$ 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*) = {⟨1,7,2⟩}, therefore $\sigma(AB) = 2$.

D. The subsume index of frequent 1-itemsets

To reduce the search space, the concept of the subsume index was proposed in [8] which is based on the following function:

$$g(X) = \{T.ID \in DB \mid X \subseteq T\} \quad (5)$$

where *T.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 [8]. The subsume index of a frequent 1-itemset, *A*, denoted by *subsume*(*A*) is defined as follows:

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

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 $B \in subsume(A)$. In other words, $\{c\} \in subsume(\{e\})$

In [8] the following two theorems concerning the subsume index idea were also presented, which in turn can be used to speed up the frequent itemset mining process.

Theorem 4 [8]. Let *A* be a frequent 1-itemset. If the support associated with *A* is equal to $\lceil minSup \times n \rceil$, then there exists no item *B* which has $\sigma(B) > \sigma(A)$ and $B \notin subsume(A)$ such that $A \cup B$ is a frequent itemset.

Theorem 5 [8]. Let the subsume index of an item *A* be $\{a_1, a_2, \ldots, a_m\}$. The support of each of the 2^m-1 nonempty subsets of $\{a_1, a_2, \ldots, a_m\}$ 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 5, 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 detail, *ae* is a frequent itemset with $\sigma(ae) = 2$. So, *aec* is also a frequent itemset and $\sigma(aec) = 2$.

III. THE PROPOSED ALGORITHM

A. The N-list intersection function

Deng et al. [3] proposed a N-list intersection function for determining the intersection of two N-lists which was $O(n+m+k)$ where *n*, *m* and *k* is the length of the first, the second and the resulting N-lists (the function traverses the resulting N-list so as to merge the same PP-codes). In this section we present an improved N-list intersection function to give $O(n+m)$. This improved function 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 the first and the second N-list denoted by `sF`, (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` falls below $\lceil minSup \times n \rceil$ stop (the itemset currently being considered is not frequent).

Given the above the improved N-list intersection function is presented in Figure 4.

```
function NL_intersection(PS1, PS2)
1. PS3 ← ∅
2.let sF be the sum of frequency of PS1 and PS2
3.let i = 0, j = 0 and frequency = 0
4. while i < PS1.size and j < PS2.size do
5.   if PS1[i].pre < PS2[j].pre then
6.     if PS1[i].post > PS2[j].post then
7.       if PS3.size > 0 and PS3[PS3.size-1].Pre =
PS1[i].pre then
8.          PS3[PS3.size-1].frequency +=
PS2[j].frequency
9.       else
10.         add the tuple ⟨PS1[i].pre, PS1[i].post,
PS2[j].frequency⟩ to PS3
11.      frequency += PS2[j++].frequency
12.    else
13.      sF = sF - PS1[i++].frequency
14.  else
15.    sF = sF - PS2[j++].frequency
16. if sF < threshold then // using early
abandoning strategy
17.    return null // stop the procedure
18.return PS3 and frequency
```

Figure 4. The improved N-list intersection function

B. The subsume index associated with each frequent 1-itemset

Theorem 6. Let A be a frequent 1-itemset. We have:

$$subsume(A) = \{B \in I_1 \mid \forall C_i \in NL(A), \exists C_j \in NL(B) \text{ and } C_j \text{ is an ancestor of } C_i\} \quad (7)$$

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. 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 6, $\langle 3,0,1\rangle$, $\langle 6,2,1\rangle$ 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 7. Let A, B, $C \in I_1$ be three frequent 1-itemsets. 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.

To find all frequent 1-itemset 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, with respect to the generate subsume index procedure, we propose a different traverse (see Figure 5) to avoid the cost of this reordering process and also facilitate the use of Theorem 7.

```
procedure Find_Subsume(I1)
1. for i ← 1 to I1.size - 1 do
2.   for j ← i - 1 to 0 do
3.     if j ∈ I1[i].Subsumes then continue
4.     if checkSubsume(I1[i].N-list, I1[j].N-list) =
true then // using Theorem 6
5.       add I1[j].name and its index, j, to
I1[i].Subsumes
6.       add all elements in I1[j].Subsumes to
I1[i].Subsumes // using Theorem 7

function checkSubsume(N-list a, N-list b)
1. let i=0 and j=0
2.while j < a.size and i < b.size do
3.   if b[i].pre < a[j].pre and b[i].post >
a[j].post then
4.     j++
5.   else
6.     i++
7.if j = a.size then
8.   return true
9. return false
```

Figure 5. The generating subsume index proceduce

C. Algorithm

The two theorems proposed in [8] and re-presented in section 2.4 were also adopted in the proposed algorithm to speed up the runtime (Figure 6). Besides, these theorems also helped reduce the memory usage because it is not necessary to determine and store the N-lists associated with a number of frequent itemsets to determine their supports.

```
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.FIs ← I1
5.Subsume ← {}
6.Find_FIs(I1, Subsumes)
7.return FIs
procedure Generate_NList(R, I1)
1. C ← ⟨R.pre,R.post, R.frequency⟩
2. H1[R.name].N-list.add(C)
3. H1[R.name].frequency += C.frequency
4. for each child in R.children
5.   Generate_NList(child)

procedure Find_FIs(Is, S)
1.for i ← Is.size - 1 to 0 do
2.   FIsnext ← ∅
3.   if Is[i].Subsumes.size > 0 then
4.     let S be the set of subset generated from all
elements of Is[i].Subsumes
5.     for each subset in S
6.       add ⟨subset, Is[i].frequency⟩ to FIs // using
theorem 5
7.   else if Is[i].size = 1 then
8.     S ← {}
9.   if Is[i].size = 1 and Is[i].frequency = threshold
then // using Theorem 4
10.    continue
11.  indexS = Is[i].Subsumes.size - 1
12.  for j ← i - 1 to 0 do
13.    if indexS >= 0 and the index of
Is[i].Subsumes[indexS] equals than j then
14.      indexS = indexS - 1
15.      continue
16.    let efirst be the first item of Is[j]
17.    FI ← {efirst} + Is[i]
18.    (FI.N-list and frequency) ←
NL_intersection(Is[j].N-list, Is[i].N-list)
```

```
19.     if FI.N-list = null then
20.       continue // using early abandoning strategy
21.     FI.frequency = frequency
22.     if(FI.frequency ≥ threshold) then
23.       add FI to FIs
24.       insert FI at position 0 in FIs_next
25.       for each subsume in S do
26.         let f = FI + subsume
27.         f.frequency = FI.frequency
28.         add f to FIs // using theorem 5
29.   Find_FIs(FIs_next, S)
```

Figure 6. The proposed algorithm

IV. THE ILLUSTRATION

An illustrative example is presented in this section using our example dataset. First the proposed algorithm scans the dataset to create the PPC-tree (Figure 3). Then, this algorithm traverses the PPC-tree to generate the N-lists associated with the frequent 1-itemsets in I_1 (Figure 7).

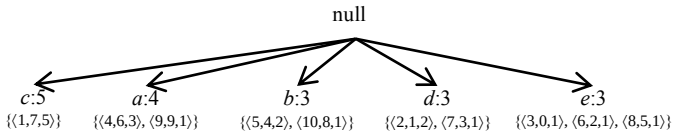

Figure 7. The I_1 and its N-lists on example dataset (*minSup*=30%)

Next the 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*($\{e\}$) = $\{c\}$, so S = $\{ec\}$; (ii) combine, in turn, with remaining frequent 1-itemsets $\{d, b, a\}$ (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 2-itemsets. In this case, this algorithm will stop here because FIs_{next} has only one element (see Figure 8).

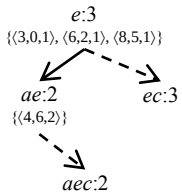

Figure 8. The frequent itemsets generated from *e* on example dataset (*minSup*=30%)

Then, using the above strategy, the other frequent 1-itemsets in turn continue to create the tree which contains all frequent itemsets as Figure 9.

In Figure 9, the proposed 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.

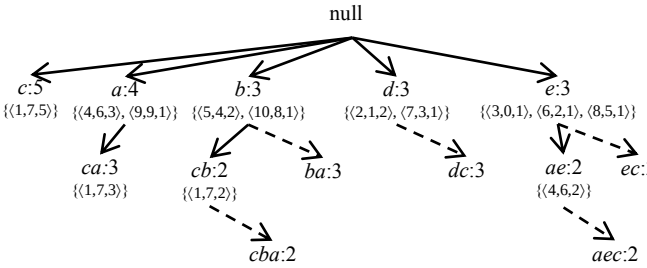

Figure 9. All frequent itemsets on example dataset (*minSup*=30%)

V. EXPERIMENTAL RESULTS

All experiments presented in this section were performed on an ASUS laptop with Intel core i3-3110M 2.4GHz and 4GBs of RAM. The operating system was Microsoft Windows 8. 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 2. We report the runtime (total execution time) of the proposed algorithm and compare it to the runtime of PrePost.

TABLE II. 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

The experimental results are presented in Figure 10. From the figure it can be observed that given a sparse datasets such as Retail, the proposed algorithm 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 1-itemsets in a sparse datasets usually have few elements. Therefore, using the subsume index concept is not effective in this case. Fortunately, this cost is usually relatively low, about 4 seconds for the Retail dataset with *minSup* = 0.1 (0.072% of the runtime) (see Figure 10(e)). However, given a dense datasets, the performance of the proposed algorithm is better than PrePost (see Figure 10(a)(b)(c) and (d)), especially with low thresholds. The proposed algorithm thus generally outperforms than the PrePost.

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

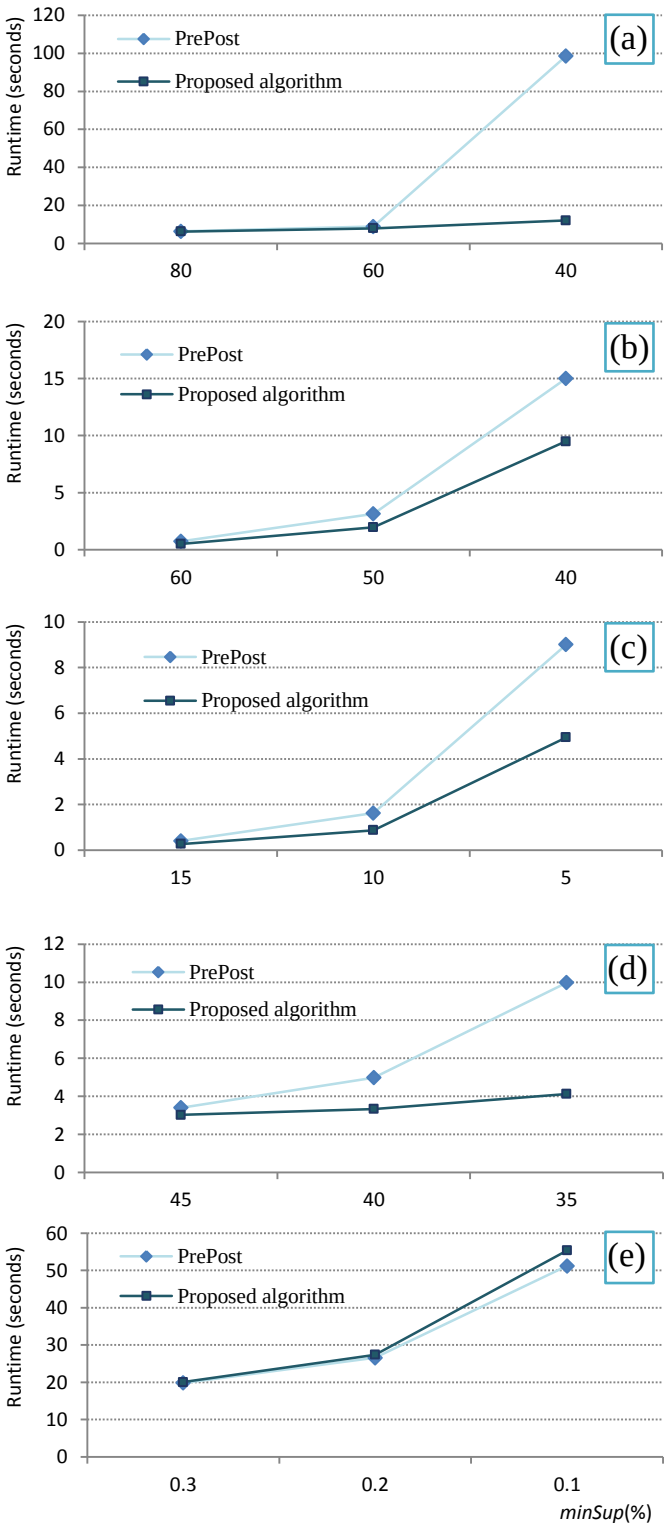

Figure 10. The runtime of the proposed and PrePost algorithms using UCL datasets: (a) Accidents, (b) Chess, (c) Mushroom, (d) Pumsb_star and (e) Retail datasets

VI. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a hybrid algorithm for mining frequent itemsets. First, we proposed several improvements on the previously published PrePost: (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 1-itemsets which were used in the proposed algorithm for improving the runtime. The proposed algorithm does not improve over the PrePost with respect to sparse datasets but the time gap is not significant. With respect to dense datasets the proposed algorithm is faster than PrePost. We therefore conclude that the proposed algorithm generally outperforms the PrePost.

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

REFERENCES

[1] Agrawal, R., Imielinski, T., Swami, A.N.: Mining association rules between sets of items in large databases. SIGMOD'93, 207-216, 1993.

[2] Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. VLDB'94, 487-499, 1994.

[3] Deng Z., Wang Z., Jiang J.J.: A new algorithm for fast mining frequent itemsets using N-lists. SCIENCE CHINA Information Sciences, 55(9), 2008-2030, 2012.

[4] Dong, J., Han, M.: BitTableFI: An efficientmining frequent itemsets algorithm. Knowledge-Based Systems, 20, 329–335, 2007.

[5] Grahne, G., Zhu, J.: Fast algorithms for frequent itemset mining using FP-trees. IEEE Transactions on Knowledge and Data Engineering, 17, 1347–1362, 2005.

[6] Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. SIGMODKDD'00, 1–12, 2000.

[7] Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Efficient Mining of Association Rules using Closed Itemset Lattices. In: Information Systems 24 (1), 25-46, 1999.

[8] Song, W., Yang, B., Xu, Z.: Index-BitTableFI: An improved algorithm for mining frequent itemsets. Knowledge-Based Systems, 21, 507-13, 2008.

[9] Vo, B., Hong, T.P., Le, B.: Dynamic bit vectors: An efficient approach for mining frequent itemsets. Scientific Research and Essays, 6(25), 5358-5368, 2011.

[10] Vo B., Hong T.P., Le B.: A Lattice-based Approach for Mining Most Generalization Association Rules. Knowledge-Based Systems, 45, 20-30, 2013.

[11] Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W.: New algorithms for fast discovery of association rules. KDD'97, 283-286, 1997.

[12] Zaki, M.J., Hsiao, C.J.: Efficient algorithms for mining closed itemsets and their lattice structure. IEEE Transactions on Knowledge and Data Engineering, 17(4), 462-478, 2005.