Expert Systems with Applications 42 (2015) 156–164

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

An efficient and effective algorithm for mining top-rank-*k* frequent patterns

Quyen Huynh-Thi-Le[a], Tuong Le[b,c], Bay Vo[b,c,*], Bac Le[a]

[a] Department of Computer Science, University of Science, VNU-HCM, Viet Nam
[b] Division of Data Science, Ton Duc Thang University, Ho Chi Minh, Viet Nam
[c] Faculty of Information Technology, Ton Duc Thang University, Ho Chi Minh, Viet Nam

ARTICLE INFO

Article history:
Available online 7 August 2014

Keywords:
Data mining
Pattern mining
Top-rank-*k* frequent patterns
N-list

ABSTRACT

Frequent pattern mining generates a lot of candidates, which requires a lot of memory usage and mining time. In real applications, a small number of frequent patterns are used. Therefore, the mining of top-rank-*k* frequent patterns, which limits the number of mined frequent patterns by ranking them in frequency, has received increasing interest. This paper proposes the *i*NTK algorithm, which is an improved version of the NTK algorithm, for mining top-rank-*k* frequent patterns. This algorithm employs an N-list structure to represent patterns. The subsume concept is used to speed up the process of mining top-rank-*k* patterns. The experiments are conducted to evaluate *i*NTK and NTK in terms of mining time and memory usage for eight datasets. The experimental results show that *i*NTK is more efficient and faster than NTK.

© 2014 Elsevier Ltd. All rights reserved.

1. Introduction

An expert system is an intelligent system that solves the complex problems based on knowledge throughout inference procedures. Generally, there are three components in an expert system including knowledge base, inference engine and user interface (Jackson, 1999). The central of expert systems is the knowledge base, because it contains the problem solving knowledge of the particular application (Ahmed, 2008). Therefore, the reduction of this knowledge space plays a big role in the implemented performance of expert systems. Association rules are important of the knowledge (Daniel & Viorel, 2004; Guil, Bosch, Túnez, & Marín, 2003) which represent the relationships between items in a dataset. To generate association rules, traditional approaches first mine frequent patterns which are itemsets, subsequences, and substructures that appear in large transactions or relational datasets with a frequency no less than a given threshold. After that, the system uses these frequent patterns and the minimum confidence to find all rules. Two above phrases require a lot of memory usage and mining time. Therefore, the reduction of time to mine frequent patterns is very useful to enhance expert systems.

Currently, there are many forms of patterns such as frequent, subsequences, and substructure patterns. Mining frequent patterns is an indispensable component in many data mining tasks such as association rule mining (Agrawal, Imielinski, & Swami, 1993; Vo, Hong, & Le, 2012, 2013; Vo, Coenen, Le, & Hong, 2013; Vo, Le, Coenen, & Hong, 2014; Vo, Le, Hong, & Le, 2014a,b), sequential pattern mining (Agrawal & Srikant, 1995; Pham, Luo, Hong, & Vo, 2014), and classification (Liu, Hsu, & Ma, 1998; Nguyen, Vo, Hong, & Thanh, 2012; Nguyen, Vo, Hong, & Thanh, 2013). Since the introduction of frequent pattern mining (Agrawal et al., 1993), various algorithms (Agrawal & Srikant, 1994; Han, Dong, & Yin, 1999; Han, Pei, & Yin, 1999; Zaki, 2000; Zaki & Gouda, 2003) have been proposed for efficiently performing the task. These algorithms can be partitioned into two main categories: using the traditional horizontal dataset format such as two important algorithms, Apriori and FP-growth (Agrawal & Srikant, 1994; Han, Dong et al., 1999; Han, Pei et al., 1999) and using the vertical dataset format such as Eclat (Zaki, 2000).

In general, mining frequent patterns uses a minimum support threshold (*min_sup*) to generate correctly and completely frequent patterns. However, setting this threshold is an interesting problem. Whether this threshold is too large or too small, it also influences the number of generated frequent patterns in a dataset. In addition, the number of produced frequent patterns is very large, while applications such as expert systems, recommendation systems and so on, only use a small number of frequent patterns. From

* Corresponding author at: Division of Data Science, Ton Duc Thang University, Ho Chi Minh, Viet Nam.

E-mail addresses: lequyenk41@gmail.com (Q. Huynh-Thi-Le), lecungtuong@tdt.edu.vn (T. Le), vodinhbay@tdt.edu.vn (B. Vo), lhbac@fit.hcmus.edu.vn (B. Le).

http://dx.doi.org/10.1016/j.eswa.2014.07.045
0957-4174/© 2014 Elsevier Ltd. All rights reserved.

above problems, Han, Wang, Lu, and Tzvetkov (2002) proposed top-k frequent closed pattern mining, where k is the number of frequent closed patterns to be mined. Then, the authors proposed the TFP algorithm to solve this task. Unlike frequent patterns, frequent closed patterns have length no less than the minimal length of each pattern (*min_l*). Although TFP implements effectively its mission, but like *min_sup*, set the value *min_l* is not a simple problem for users. Therefore, a new direction of research was proposed, that is the problem of top-rank-k frequent pattern mining. To solve this problem, FAE (Deng & Fang, 2007) and VTK algorithms (Fang & Deng, 2008) are proposed. A top-rank-k of frequent patterns is selected based on rank order of frequency. Recently, Deng (2014) proposed NTK algorithm for mining top-rank-k frequent patterns based on the idea of PPC-tree (the Pre-order and Post-order Code tree). NTK is efficient due to its patterns presentation based on Node-list structure. The experimental results show that NTK is more effective than FAE and VTK.

Considering carefully Node-list structure, we found that N-list (Deng, Wang, & Jiang, 2012) better than Node-list because the length of the Node-list of a pattern is greater than the length of its N-list. Hence, the time required to join two Node-lists is longer than that of N-lists. In addition, NTK must generate and test all candidates in each loop of the algorithm. Therefore, this paper presents an efficient method for mining top-rank-k frequent patterns called *i*NTK. Unlike NTK, *i*NTK uses N-list structure with an improved N-list intersection function to reduce the run-time and memory-consuming. Moreover, *i*NTK employs the subsume index concept to directly mine frequent patterns without generating candidates in a number of cases.

The rest of this paper is organized as follows. Section 2 presents the related work for mining top-rank-k frequent patterns. Section 3 introduces the basic concepts. The *i*NTK algorithm for mining top-rank-k frequent patterns is described in Section 4. Section 5 compares the performance of the *i*NTK and NTK algorithms. Section 6 summarizes the study and gives some topics for future research.

2. Related work

Since mining top-rank-k frequent patterns is proposed, a number of algorithms such as FAE, VTK and NTK were built to solve this problem. Besides, mining top-rank-k erasable itemsets is also proposed (Deng, 2013; Nguyen, Le, Vo, & Le, 2014).

FAE is the first algorithm (Deng & Fang, 2007) to solve the problem of mining top-rank-k frequent patterns. FAE is an acronym for "Filtering and Extending"; it uses heuristic rules to reduce the search space, filters undesired patterns and selects useful patterns to generate the next patterns. Next, VTK (Fang & Deng, 2008) (Vertical Mining of top-rank-k frequent patterns) is more efficient than FAE because it does not need to scan the entire dataset to calculate the support of frequent patterns.

Recently, NTK algorithm was built for mining top-rank-k frequent patterns (Deng, 2014). This algorithm was proven to be more effective than FAE and VTK because it uses Node-list, a data structure that has been effectively used in frequent pattern mining (Deng & Wang, 2010). In NTK, first a tree construction algorithm is used to build a PPC-tree. Then, Node-list structure associated with frequent 1-patterns is generated. Unlike FP-tree-based approaches, this approach does not build additional trees repeatedly; it mines frequent patterns directly using Node-list.

In 2010, Node-list is first proposed (Deng & Wang, 2010). After that, N-list, like Node-list structure, has also been proposed (Deng et al., 2012) to mine frequent patterns. Both of them are generated from a PPC-tree and a list of nodes sorted in pre-order ascending order. Besides, the Node-list and N-list of a pattern contains t items can be produced from two patterns contains $(t-1)$ items. The difference between them is that Node-list is constructed by the suffix nodes while N-list is constructed by prefix nodes, and the length of Node-list of a pattern is greater than the length of N-list of a pattern. Therefore, Node-list used in NTK requires a lot of time and memory. In Vo, Coenen et al., 2013; Vo, Hong et al., 2013; Vo, Le, Coenen et al., 2014; Vo, Le, Hong et al., 2014a,b, N-list and subsume index (Song, Yang, & Xu, 2008) of frequent 1-pattern was used for mining frequent itemsets effectively. NSFI algorithm was proven more outperforms than the PrePost. In this paper, *i*NTK, an improvement algorithm of NTK, is proposed. This algorithm uses N-list structure and subsume index of 1-patterns to enhance the mining time and the memory usage.

3. Problem definition

3.1. Frequent patterns

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of items, and $DB = \{T_1, T_2, \ldots, T_n\}$ be a set of transactions, where T_i $(1 \leqslant i \leqslant n)$ is a transaction that has a unique identifier and contains a set of items. Given a pattern P and a transaction T, it is said that T contains P if and only if $P \subseteq T$.

Definition 1 (*support of a pattern*). Given a *DB* and a pattern P $(\subseteq I)$, the support of pattern P (SUP_P) in *DB* is the number of transactions containing P.

A pattern P is a frequent pattern if support of P is no less than a given *min_sup*.

3.2. Problem of mining top-rank-k frequent patterns

Deng and Fang (2007) described the problem of mining top-rank-k patterns as follows.

Definition 2 (*rank of a pattern*). Given a *DB* and a pattern $X\,(\subseteq I)$, the rank of X (R_X) is defined as $R_X = |\{SUP_Y | Y \subseteq I \text{ and } SUP_Y \geqslant SUP_X\}|$, where $|Y|$ is the number of items in Y.

Definition 3 (*top-rank-k frequent patterns*). Given a *DB* and a threshold k, a pattern $P\,(\subseteq I)$ belongs to a top-rank-k frequent pattern (TR_k) if and only if $R_P \leqslant k$.

Given a *DB* and a threshold k, top-rank-k frequent pattern mining is the task of finding the set of frequent patterns whose ranks are no greater than k. That means that $TR_k = \{P | P \subseteq I \text{ and } R_P \leqslant k\}$.

Example 1. Dataset DB_E in Table 1 is used throughout the article. According to Definition 1, $SUP_{\{c\}} = 5$ because five transactions, namely 2, 3, 4, 5, and 6, contain c. Table 2 shows the ranks and supports of all patterns in DB_E. According to Table 2, $SUP_{\{c\}}$ is the largest, and therefore $R_{\{c\}} = 1$.

3.3. N-list structure

Deng et al. (2012) presented the PPC-tree, an FP-tree-like structure (Han, Dong et al., 1999; Han, Pei et al., 1999), the PPC-tree construction algorithm, and the N-list structure as follows.

Table 1
Example dataset (DB_E).

TID	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*

Table 2
Ranks and supports of all patterns for DB_E.

Rank	Support	Patterns
1	5	{*c*}
2	4	{*a*}
3	3	{*ca*}, {*ce*}, {*d*}, {*cd*}, {*b*}, {*ab*}, {*e*}
4	2	{*cea*}, {*ae*}, {*cb*}, {*cba*}
5	1	{*cdba*}, {*ceba*}, {*cda*}, {*cfed*}, {*ceb*}, {*cfe*}, {*dfe*}, {*ced*}, {*df*}, {*adb*}, {*cdb*}, {*cfd*}, {*de*}, {*ef*}, {*aeb*}, {*ad*}, {*bd*}, {*f*}, {*be*}, {*cf*}

Table 3
DB_E after being sorted in descending order of frequency.

TID	Sorted items
1	*a*, *b*
2	*c*, *a*, *b*, *d*
3	*c*, *a*, *e*
4	*c*, *a*, *b*, *e*
5	*c*, *d*, *e*, *f*
6	*c*, *d*

Definition 4. PPC-tree is a tree structure where includes one root and set of nodes. Each node *N* composed of five values: *N.name*, *N.child*, *N.count*, *N.preorder* and *N.postorder* corresponding to name of item in dataset, set of children node of *N*, frequency of *N* and order when visiting PPC-tree by pre-order and post-order, respectively. PPC-tree's root names *R* has *R.name* = *null* and *R.count* = 0.

The PPC-tree construction algorithm (Deng & Wang, 2010; Deng, 2014) is given in Fig. 1.

Example 2. First, items in transactions are sorted in descending order of frequency. The results are shown in Table 3.

Fig. 2 shows the PPC-tree generated for DB_E. Each rectangle represents a node. A pairs of letter and number in each rectangle is the name of the item and its support. The *preorder* and *postorder* of the corresponding node are represented by a pair of numbers in each bracket. For example, the node {*c*,5} has *preorder* = 3, *postorder* = 10, *name* = *c*, and *count* = 5.

Definition 5 (*PP-code*). In a PPC-tree, each node N_i has PP-codes, $PP_i = \langle(N_i.preorder,\ N_i.postorder)\!: N_i.count\rangle$.

Property 1 (*ancestor-descendant relationship of PP-codes*). Given PP_i and PP_j are two PP-codes, PP_i is an ancestor of PP_j if and only if $PP_i.preorder < PP_j.preorder$ and $PP_i.postorder > PP_j.postorder$.

Example 3. Let $PP_1 = \langle(4,6)\!:3\rangle$ and $PP_2 = \langle(7,3)\!:2\rangle$. Based on Property 1, PP_1 is an ancestor of PP_2 because $PP_1.preorder = 4 < PP_2.preorder = 7$ and $PP_1.postorder = 6 > PP_2.postorder = 3$.

Definition 6 (*N-list of a 1-pattern*). Given a PPC-tree, N-list of a frequent 1-pattern, *A*, is a sequence of all the PP-codes of nodes in the PPC-tree whose *name* is *A*. In one N-list, PP-codes are arranged in *preorder* ascending order.

Each PP-code in N-list is denoted by $PP = \langle(preorder,\ postorder)\!:count\rangle$. N-list of a frequent pattern is denoted by $\{PP_1, PP_2, \ldots, PP_l\}$, where $PP_1.preorder < PP_2.preorder < \cdots < PP_l.preorder$.

```
Function Construct-PPC-tree (DB)
1. Scanning DB, inserting all items and their supports to I1.
2. Sort I1 in support descending order. If the supports of some items are equal, the orders
   among them can be assigned arbitrarily.
3. Create the root of a PPC-tree, R, and name it as "null".
4. For each transaction Tr in DB do
5.     Sort all items in support descending order.
6.     Call Insert_Tree(Tr, R).
7. Visit the PPC-tree to generate the preorder and the postorder values of each node by pre-
   order traverse and post-order traverse, respectively. To traverse the PPC-tree in pre-order,
   perform the following three operations: visit the root node, traverse all left sub-trees, and
   then traverse all right sub-trees. To traverse the PPC-tree in post-order, perform the
   following three operations: traverse all left sub-trees, traverse all right sub-trees, and then
   visit the root node.

Function Insert_Tree(Tr, R)
1. t ← the first element in Tr, Tr = Tr \ t.
2. If R has a child node N such that N.name = t then N.count ++.
3. Else create a new node N with N.count = 1 and N.name = t, R.child = N
4. If R not null then call Insert_Tree(Tr, N).
```

Fig. 1. PPC-tree construction algorithm.

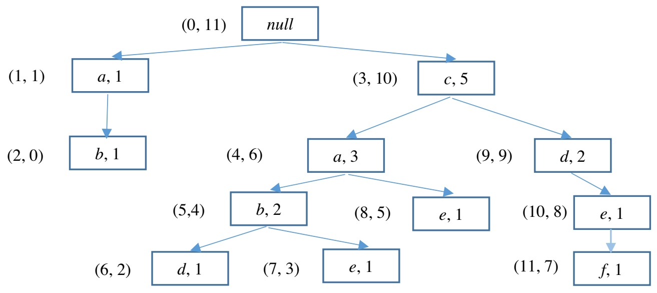

Fig. 2. PPC-tree for DB_E.

Example 4. N-list of item *b* includes two PP-codes, namely $\langle(2,0)\!:1\rangle$ and $\langle(5,4)\!:2\rangle$. Fig. 3 shows the N-lists of all frequent items in Example 1.

Definition 7 (*N-list of a t-pattern*). Let two *t*-patterns P_1X and P_2X and their N-list $NL_1 = \{PP_{11}, PP_{12}, \ldots, PP_{1m}\}$ and $NL_2 = \{PP_{21}, PP_{22}, \ldots, PP_{2n}\}$ respectively. The N-list of P_1P_2X is generated by the following rules:

(i) $\forall PP_i \in NL_1$ $(1 \leqslant i \leqslant m)$ and $PP_j \in NL_2$ $(1 \leqslant j \leqslant n)$, if PP_i is the ancestor of PP_j then add the PP-code $\langle(PP_i.preorder,\ PP_i.postorder)\!:PP_j.count\rangle$ to N-list of P_1P_2X.
(ii) Check all PP-codes of N-list of P_1P_2X, merge the PP-codes has same *preorder* and *postorder* values.

As shown by Fig. 3, $NL_{\{c\}} = \{\langle(3,10)\!:5\rangle\}$ and $NL_{\{f\}} = \{(11,7)\!:1\rangle\}$. According to Definition 7, $NL_{\{cf\}}$ can be built as follows. $\langle(3,10)\!:5\rangle$ is an ancestor of $\langle(11,7)\!:1\rangle$; therefore, PP-code $\langle(3,10)\!:1\rangle$ is added to $NL_{\{cf\}}$. Because there are no other elements in $NL_{\{c\}}$, the processing is stopped. The final result is $NL_{\{cf\}} = \langle(3,10)\!:1\rangle$ (Fig. 4).

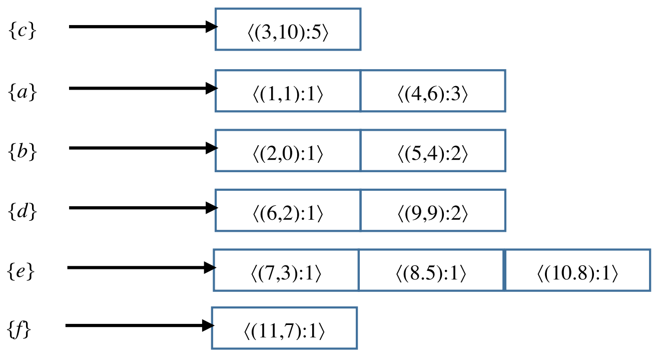

Fig. 3. N-lists of all 1-patterns in Example 1.

$$\begin{tikzcd}
\{c\} \arrow[r] & \langle(3,10):5\rangle \arrow[d] \\
\{f\} \arrow[r] & \langle(11,7):1\rangle \arrow[d, "3<11 \text{ and } 10>7 \text{ Ok}"'] \\
\{cf\} \arrow[r] & \langle(3,10):1\rangle
\end{tikzcd}$$

Fig. 4. N-list of {*cf*} in Example 1.

Property 2. Let P is a t-pattern and its N-list $NL_P = \{PP_1, PP_2, \ldots, PP_n\}$. The support of P is determined by $SUP_P = PP_1.count + PP_2.count + \cdots + PP_n.count$.

Example 5. The N-list of a is $NL_{\{a\}} = \{\langle(1,1):1\rangle, \langle(4,6):3\rangle\}$. Hence, $SUP_{\{a\}} = 1 + 3 = 4$. To verify the support of a, scanning DB_E can be found that there are three transactions that contain a.

3.4. Subsume index of frequent 1-patterns

To reduce the search space, subsume index concept was proposed by Song et al. (2008). It is based on the following function:

$$G_X = \{T.ID \in DB | X \subseteq T\}$$

where $T.ID$ is the ID of transaction T, and G_X is the set of IDs of the transactions which include all items $i \in X$.

Example 6. For the example dataset DB_E, $G_{\{c\}} = \{2,3,4,5,6\}$ because c exists in transactions 2, 3, 4, 5, and 6.

Definition 8 Song et al., 2008. Subsume index of a frequent 1-pattern, A, denoted by SS_A is defined as follows:

$$SS_A = \{B \in I_1 | G_A \subseteq G_B\}$$

Example 7. $G_{\{e\}} = \{3,4,5\}$ and $G_{\{c\}} = \{2,3,4,5,6\}$. $c \in SS_{\{e\}}$ because $G_{\{e\}} \subseteq G_{\{c\}}$.

Song et al. (2008) also presented the following property concerning subsume index, which can be used to speed up the frequent pattern mining process.

Property 3 Song et al., 2008. Let subsume index of pattern X be $\{a_1, a_2, \ldots, a_m\}$. The support of the patterns generated by combining X with each of the $2^m - 1$ nonempty subsets of $\{a_1, a_2, \ldots, a_m\}$ is equal to SUP_X.

Example 8. According to Example 6, $SS_{\{e\}} = \{c\}$. Therefore, the only $2^m - 1$ nonempty subset of $SS_{\{e\}}$ is $\{c\}$. Based on Property 3, the support of $2^m - 1$ patterns, which are combined $2^m - 1$ nonempty subsets of $SS_{\{e\}}$ with e, is equal to $SUP_{\{e\}}$. In this case, $SUP_{\{ec\}} = SUP_{\{e\}} = 3$. Besides, $\forall X$, $SUP_{X \cup e}$ is also equal to $SUP_{X \cup ec}$. Therefore, $\{ae\}$ is a frequent pattern with $SUP_{\{ae\}} = 2$ and $\{aec\}$ is also a frequent pattern and $SUP_{\{aec\}} = 2$.

```
Function NL_intersection(NL1, NL2)
1. NL3 ← ∅
2. Let i = 0, j = 0.
3. While i < |NL1| and j < |NL2| do
4.    If NL1[i].preorder < NL2[j].preorder then
5.       If NL1[i].postorder > NL2[j].postorder then
6.          If |NL3| > 0 and the last element of NL3 has a preorder value that is equal to NL1[i].preorder then
7.             Increase the count value of the last element of NL3 by NL2[j].count
8.             j++
9.          Else
10.            Add the tuple ⟨NL1[i].preorder, NL1[i].postorder, NL2[j].count⟩ to NL3
11.            i++
12.      Else j++
13. Return NL3
```

Fig. 5. Improved N-list intersection function.

```
Procedure Find_Subsume(I1)
1. For i ← 1 to |I1| - 1 do
2.    For j ← i - 1 to 0 do
3.       If j ∈ I1[i].subsume then continue
4.       If checkSubsume( I1[i].N-list, I1[j].N-list) = true then // using Property 4
5.          Add I1[j].name and its index, j, to I1[i].subsume
6.          Add all elements in I1[j].subsume to I1[i].subsume // using Property 5

Function checkSubsume(N-list a, N-list b)
1. Let i=0 and j=0
2. While j < |a| and i < |b| do
3.    If b[i].preorder < a[j].preorder and b[i].postorder > a[j].postorder then
4.       j++
5.    Else
6.       i++
7. If j = |a| then
8.    Return true
9. Return false
```

Fig. 6. Subsume index generation procedure.

```
Procedure iNTK(DB)
1. Call Construct-PPC-tree (DB) to build PPC-tree
2. Determine the N-list of $I_1$ (frequent 1-patterns)
3. Call Find_Subsume($I_1$) to find the subsume index value of each item in $I_1$
4. Find 1-patterns with rank not larger than $k$ in all 1-patterns; denote the set of these 1-patterns by $TR_1$
5. For each 1-pattern ($X$) in the 1-patterns, insert it and patterns combined $X$ with $2^m$-1 subset from the subsume index into $Tab_k$.
6. For ($j$=2; $TR_{j-1} \neq \emptyset$; $j$++)
7.     $CR_j \leftarrow$ Candidate_gen($TR_{j-1}$)
8.     For each $C \in CR_j$, $C$ is generated by $P_1(\in TR_{j-1})$ and $P_2$ $(\in TR_{j-1})$
9.         $C.N\text{-}list$ = NL_intersection($P_1.N\text{-}list$, $P_2.N\text{-}list$)
10.        If $SUP_C$ is larger than or equal to the support of patterns in any entry of $Tab_k$ and
           the number of entries is less than $k$, insert $C$ with its support into $Tab_k$
11.        If $C.subsume$ is not null, let $S$ be the set of subsets generated from all elements of
           $C.subsume$
12.            For each subset $s$ in $S$, the algorithm combines $C$ with $s$ and inserts it to $Tab_k$
13.  If $C \in Tab_k$ then $TR_j \leftarrow C$

Procedure Candidate_gen($TR_{j-1}$)
1.  $CR_j \leftarrow \emptyset$
2.  For each $C_u \in TP_{j-1}$
3.      For each $C_v \in TP_{j-1}$ $(C_v \neq C_u)$
4.          If ($C_v[1] \notin C_u.subsume$) and ($C_u[1] \neq C_v[1]$ ^ $C_u[2] = C_v[2]$ ^... $C_u[j-2] = C_v[j-2]$ ^
$C_u[j-1] = C_v[j-1]$) then
5.              $C \leftarrow C_u[1]C_v[1]C_u[2]...C_u[j-2]C_u[j-1]$
6.              $C.subsume$.add($C_u.subsume$)
7.              If each ($j$-1)-subset of $C$ belongs to $TR_{j-1}$ then
8.                  $CR_j \leftarrow CR_j \cup \{C\}$
9.  Return $CR_j$
```

Fig. 7. *i*NTK algorithm.

Table 4
Set of 1-patterns and their subsume indexes.

1-Pattern (X)	{c}	{a}	{b}	{d}	{e}	{f}
SS_X			{a}	{c}	{c}	{d}, {e}, {c}

Table 5
Results of Tab_k after *Step 2*.

k	SUP_X	1-Pattern	1-Pattern combined with subset of subsume
1	5	{c}	
2	4	{a}	
3	3	{b}, {d}, {e}	{ba}, {dc}, {ec}
4	1	{f}	{df}, {ef}, {cf}

Table 6
Set of 2-patterns candidates and their subsume indexes.

2-Pattern (X)	{ae}	{cb}	{ca}
SS_X	{c}	{c}	

Table 7
Results of Tab_k after step 3.

k	SUP_X	Patterns
1	5	{c}
2	4	{a}
3	3	{b}, {d}, {e}, {ba}, {dc}, {ec}, {ca}
4	2	{ae}, {cb}, {cae}, {cab}

Table 8
Final results for Tab_k.

K	SUP_X	Patterns
1	5	{c}
2	4	{a}
3	3	{b}, {d}, {e}, {ba}, {dc}, {ec}, {ca}
4	2	{ae}, {cb}, {cae}, {cab}

4. *i*NTK algorithm

4.1. N-list intersection function

Vo, Coenen et al. (2013), Vo, Hong et al. (2013), Vo, Le, Coenen et al. (2014) and Vo, Le, Hong et al. (2014a,b) proposed an improved N-list intersection function for determining the intersection process of two N-lists. Its complexity is $O(n + m)$, where n and m are the lengths of the first and second N-lists, respectively (the function does not traverse the resulting N-list). The improved N-list intersection function is presented in Fig. 5.

4.2. Subsume index associated with each frequent 1-pattern

Vo, Coenen et al. (2013), Vo, Hong et al. (2013), Vo, Le, Coenen et al. (2014) and Vo, Le, Hong et al. (2014a,b) also presented properties of the subsume index associated with each frequent 1-pattern based on N-list concept. These properties are summarized as follows.

Property 4. Let $A \in I_1$ (the frequent 1-patterns). The subsume index of A, $SS_A = \{B \in I_1 | \forall PP_i \in NL_A, PP_j \in NL_B$ and PP_j is an ancestor of $PP_i\}$.

Proof. This property can be proven as follows: all PP-codes in NL_A have a PP-code ancestor in NL_B, which means that all transactions that contain A also contain B. $G_A \subseteq G_B$, which implies that $B \in SS_A$. Therefore, this property is proven. □

Example 9. We have $NL_{\{c\}} = \{\langle(3,10):5\rangle\}$ and $NL_{\{e\}} = \{\langle(7,3):1\rangle, \langle(8,5):1\rangle, \langle(10,8):1\rangle\}$. According to Property 4, $\langle(7,3):1\rangle$, $\langle(8,5):1\rangle$ and $\langle(10,8):1\rangle \in NL_{\{e\}}$ are descendants of $\langle(3,10):5\rangle \in NL_{\{c\}}$. Therefore, $c \in SS_{\{e\}}$.

Property 5. Let A, B, and $C \in I_1$. If $A \in SS_B$ and $B \in SS_C$, then $A \in SS_C$.

Proof. $A \in SS_B$ and $B \in SS_C$; therefore, $G_B \subseteq G_A$ and $G_C \subseteq G_B \Rightarrow G_C \subseteq G_A$. This property is proven. □

The method generating the subsume indexes associated with 1-patterns is presented in Fig. 6.

4.3. The proposed algorithm

Subsume index is used to speed up the mining time of *i*NTK (see Fig. 7). Besides, this concept also reduces memory usage because *i*NTK does not determine and store the N-lists associated with a number of frequent patterns to determine their supports.

*i*NTK employs t-patterns to explore $(t+1)$-patterns. By using N-lists, *i*NTK does not need to scan datasets repeatedly to get the supports of $(t+1)$-patterns. Furthermore, using subsume index concept reduces the runtime of the candidate generation function because fewer candidates' information is determined. In addition, using the subsume index concept also reduces the required storage space of the N-lists of patterns that are in the top-rank-k patterns and not used to create the next candidates. The processing procedure is as follows.

(1) Scan the PPC-tree and generate N-lists of all 1-patterns.
(2) Find subsume indexes of all 1-patterns.
(3) Find the top-rank-k frequent 1-patterns and insert them into the top-rank-k table. The top-rank-k table contains patterns and their supports. Patterns with the same support are stored in the same entry. The number of entries in the top-rank-k table is not more than threshold k. For each 1-pattern (X) inserted into the top-rank-k table, find subsets from the subsume index of X, generate new patterns by associating each with X and insert into the entry of X in top-rank-k.
(4) For each 1-pattern (X) in the top-rank-k table, the algorithm finds all 2-pattern candidates by combining X with the other 1-patterns in the top-rank-k table. Note that only use the 1-patterns which not belong to the subsume index of X. All the 2-pattern candidates which has their support is not less than the smallest support of the top-rank-k table and the number of entries in top-rank-k table is not more than k will be inserted into the top-rank-k table.
(5) For each 2-pattern (Y) inserted into the top-rank-k table, the algorithm uses Property 5 to find $2^m - 1$ subsets from its subsume index and generates new patterns by combining the $2^m - 1$ subsets with Y. This new patterns will be inserted into same entry of Y in top-rank-k table because their supports are equal to the support of Y. After each insertion, the top-rank-k table is checked to ensure that the number of entries is not more than k. If the number of entries is larger than k, the entries whose support is less than the k-th minimum support are deleted from the top-rank-k table.
(6) Repeat steps 4 and 5 using t-patterns (produced by the *Candidate_gen* function) in the top-rank-k table to generate candidate $(t+1)$-patterns until no new candidate patterns can be generated.

4.4. An illustrative example

Given $k = 4$, process of mining top-rank-k frequent patterns from the dataset in Table 1 is as follow.

Table 9
Characteristics of experimental datasets.

Dataset	# Of Trans	# Of Items
Chess	3196	75
Mushroom	8124	119
Connect	67,557	129
T10I4D100K	10,000	870
Test990.99KD1	99,822	990
Test2K50KD1	50,000	2000
Pumsb	49,096	2113
Retail	88,162	16,470

Step 1. Find 1-patterns and their subsume indexes (Table 4).
Step 2. Insert 1-patterns and patterns generated from their subsume indexes into Tab_k (Table 5).
Step 3. *i*NTK finds 2-pattern candidates which are $\{de\}$, $\{be\}$, $\{ae\}$, $\{bd\}$, $\{ad\}$, $\{cb\}$, $\{ca\}$ with their N-list $\{\langle(9,9):1\rangle\}$, $\{\langle(5,4):1\rangle\}$, $\{\langle(4,6):2\rangle\}$, $\{\langle(5,4):1\rangle\}$, $\{\langle(4,6):1\rangle\}$, $\{\langle(3,10):2\rangle\}$, $\{\langle(3,10):3\rangle\}$ repeatedly. There are three candidates with their subsume indexes (Table 6) and two patterns generated from their subsume indexes including $\{ca\}$, $\{ae\}$, $\{cb\}$, $\{cae\}$, $\{cab\}$ is inserted into Tab_k. Table 7 shows the results. The patterns with the support equal 1 are deleted from Tab_k.
Step 4. Find 3-pattern candidates and no 3-pattern candidates are generated. The process is stopped. The final Tab_k is shown in Table 8.

5. Experimental results

This section compares *i*NTK to NTK algorithms in terms of mining time and memory usage for six datasets[1] such as *Chess*, *Connect*, *Mushroom*, *Pumsb*, *Retail*, *T10I4D100K* and two synthetic datasets (generated by the LUCS-KDD data generator[2]). To generate *Test990.99KD1*, the number of items and transactions are set to 990 and 99,822 respectively and to generate *Test2K50KD1*, the number of items and transactions are set to 2000 and 50,000, respectively. Table 9 shows the characteristics of these datasets. All the experiments were performed on a personal computer with an Intel Core2 Duo 2.66-GHz CPU and 2 GB of RAM. The operating system was Microsoft Windows 7. All the programs were coded in MS/Visual C#.

5.1. Mining time

Input data for NTK and *i*NTK are slightly different. Input data for NTK are Node-lists converted from the original datasets. Input data for *i*NTK includes N-list and subsume indexes of 1-patterns. Although the time required to create input data for *i*NTK is longer than that for NTK, this does not significantly affect the efficiency of *i*NTK because this procedure done only once. Table 10 shows the time required for converting the datasets.

Figs. 8–15 show mining times of *i*NTK and NTK for the experimental datasets with various values of k. The results show that *i*NTK outperforms NTK for large values of k or dense data. This is explained as follows. Generating subsume index requires a cost of time and memory usage. In the case of sparse datasets, such as *Retail*, the number of subsume indexes of a frequent 1-pattern is usually small. Besides, if value of k is small, the 1-patterns with small support usually have a little chance to insert into the top-rank-k table. However, these patterns have many elements in their

[1] Downloaded from http://fimi.ua.ac.be/data/.
[2] Downloaded from http://cgi.csc.liv.ac.uk/~frans/KDD/Software/LUCS-KDD-DataGen/generator.html.

Table 10
Dataset conversion times.

Dataset	Time required for creating Node-list (s)	Time required for creating N-list and finding subsume index values (s)
Chess	0.274	0.391
Mushroom	0.275	0.392
Connect	3.439	5.748
T10I4D100K	83.011	114.236
Test990.99KD1	71.509	133.023
Test2K50KD1	158.524	280.208
Pumsb	313.710	521.534
Retail	2715.031	3893.403

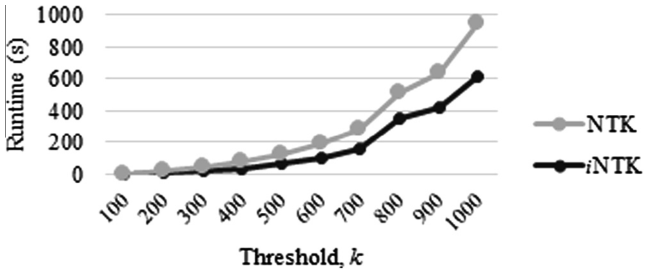

Fig. 11. Mining time of *i*NTK and NTK for *T10I4D100K* dataset.

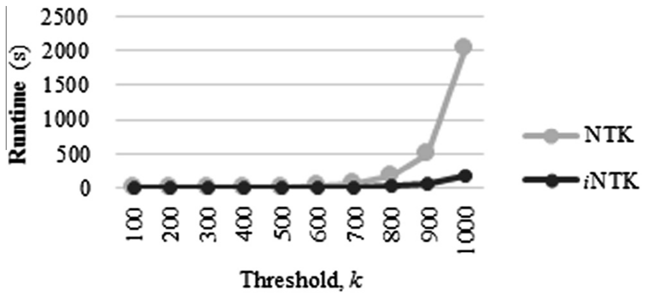

Fig. 8. Mining time of *i*NTK and NTK for *Chess* dataset.

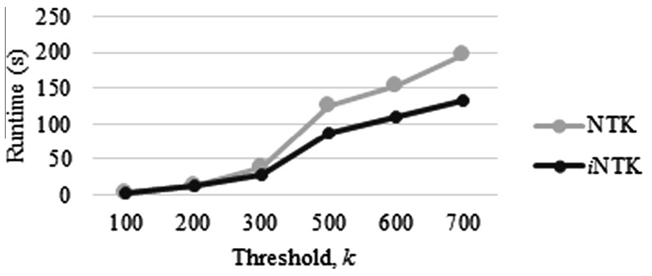

Fig. 12. Mining time of *i*NTK and NTK for *Test990.99KD1* dataset.

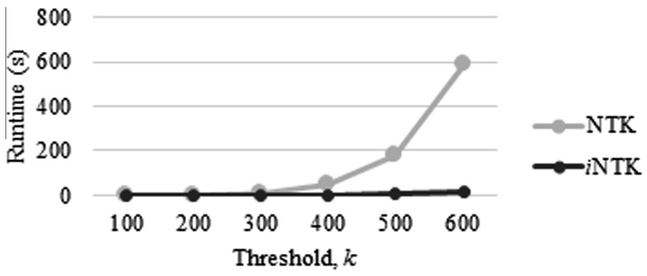

Fig. 9. Mining time of *i*NTK and NTK for *Mushroom*.

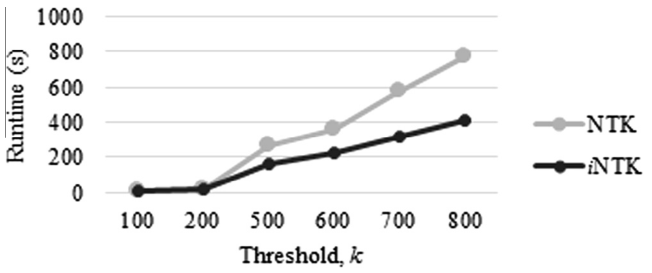

Fig. 13. Mining time of *i*NTK and NTK for *T2K50K1D* dataset.

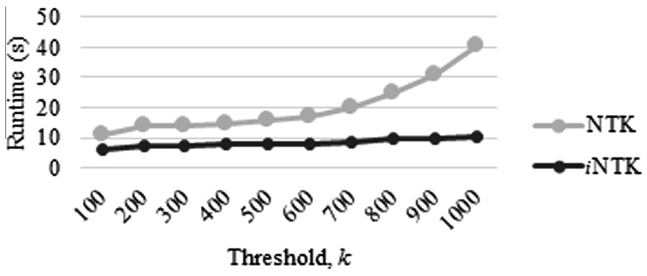

Fig. 10. Mining time of *i*NTK and NTK for *Connect* dataset.

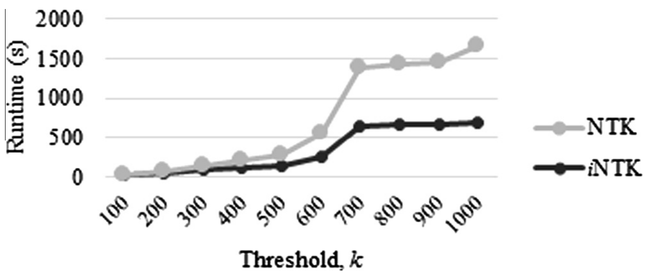

Fig. 14. Mining time of *i*NTK and NTK for *Pumsb* dataset.

subsume indexes. Therefore, *i*NTK is not effective for the sparse datasets or the small value of *k*.

5.2. Memory usage

Using subsume index does not only reduce the mining time but also reduce the memory usage. Table 11 shows the memory required to store the user input data of the NTK and *i*NTK algorithms. The memory usage of *i*NTK is slightly greater than that of NTK.

Figs. 16–23 show maximum amounts of memory used by NTK and *i*NTK for various *k* values. According to these charts, *i*NTK uses less memory. For dense datasets such as *Mushroom* and the large *k* values, subsume index significantly reduces memory usage due to a large number of candidates has not determine its information by using subsume index concept. Therefore, *i*NTK do not stored N-list of these candidates. Besides, *i*NTK uses N-list instead of Node-list for reducing memory usage because N-list saves the shorter sequence PP-codes.

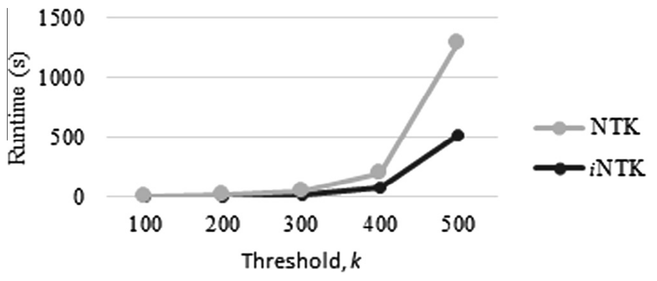

Fig. 15. Mining time of *i*NTK and NTK for *Retail* dataset.

Table 11
Memory usage of *i*NTK and NTK.

Dataset	NTK (kB)	*i*NTK (kB)
Chess	1972.105	1982.203
Mushroom	1.412	1.434
Connect	18.446	16.469
T10I4D100K	83.011	114.236
Test990.99KD1	41249.050	41304.150
Test2K50KD1	46579.808	46666.322
Pumsb	57.429	58.089
Retail	39.289	40.788

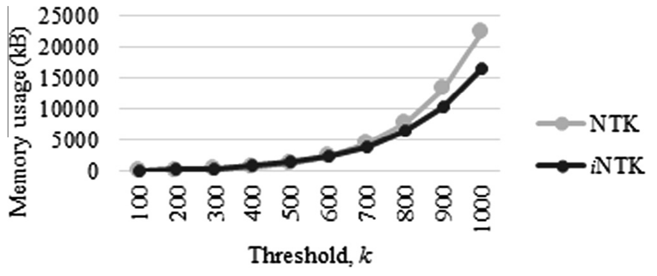

Fig. 16. Memory usage of *i*NTK and NTK for *Chess* dataset.

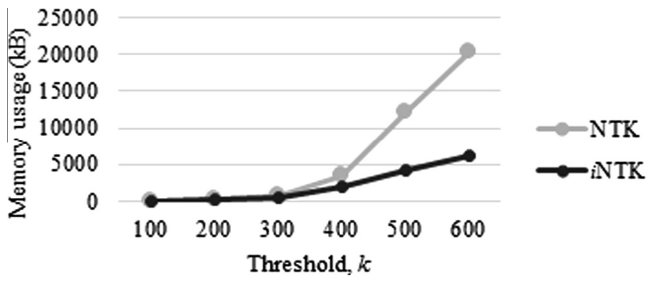

Fig. 17. Memory usage *i*NTK and NTK for *Mushroom* dataset.

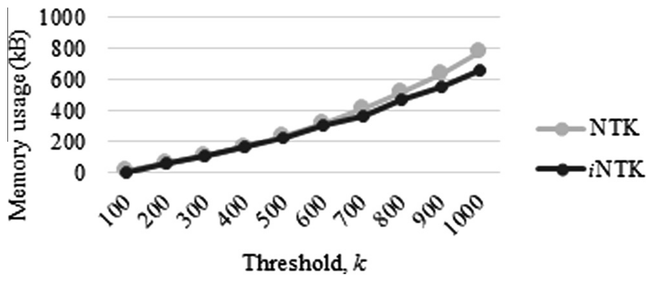

Fig. 18. Memory usage of *i*NTK and NTK for *Connect* dataset.

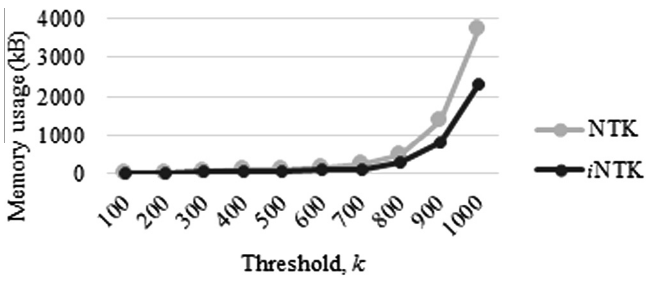

Fig. 19. Memory usage *i*NTK and NTK for *T10I4D100K* dataset.

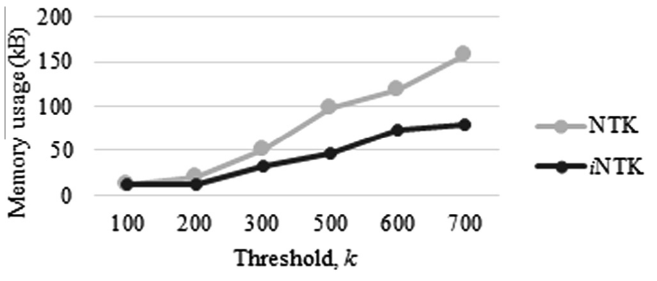

Fig. 20. Memory usage *i*NTK and NTK for *T990.99822.1* dataset.

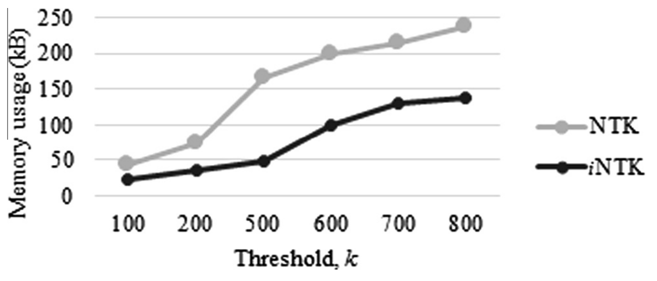

Fig. 21. Memory usage *i*NTK and NTK for *T2K50K1D* dataset.

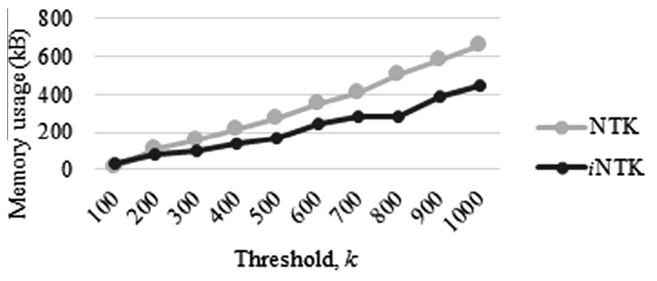

Fig. 22. Memory usage *i*NTK and NTK for *Pumsb* dataset.

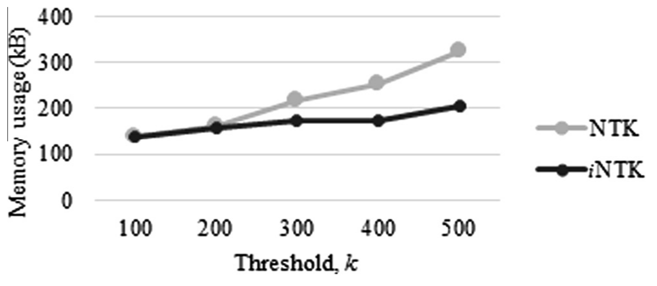

Fig. 23. Memory usage *i*NTK and NTK for *Retail* dataset.

6. Conclusion and future work

This paper presents an efficient improvement algorithm called *i*NTK to mine top-rank-*k* frequent patterns. The advantage of *i*NTK lies in that it uses N-list and subsume index of 1-patterns. N-list store information shorter than Node-list and subsume index help *i*NTK directly mining in case of patterns belonged to top-rank-*k* table contain other 1-patterns in their subsume set. This causes that *i*NTK consume less memory and runtime. Extensive experiments show that *i*NTK outperforms NTK for various datasets.

The proposed method may still generate a huge number of patterns for top-rank-*k* frequent pattern mining. Therefore, the extension of *i*NTK to mine top-rank-*k* compressed frequent patterns, such as maximal frequent patterns (Bayardo, 1998; Burdick, Calimlim, Flannick, Gehrke, & Yiu, 2005) or closed frequent patterns (Lee, Wang, Weng, Chen, & Wu, 2008; Wang, Han, & Pei, 2003) is an interesting topic for future research. Moreover, as big data become more and more popular in practice, the parallel/distributed implementation of *i*NTK to mine frequent patterns from huge dataset is also an interesting work.

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

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. In *VLDB'94* (pp. 487–499).

Agrawal, R., & Srikant, R. (1995). Mining sequential patterns. In *ICDE'95* (pp. 3–14).

Agrawal, R., Imielinski, T., & Swami, A. (1993). Mining association rules between set of items in large databases. In *SIGMOD'93* (pp. 207–216).

Ahmed, T. S. (2008). Premises reduction of rule based expert systems using association rules technique. *International Journal of Soft Computing, 3*, 195–200.

Bayardo, R. J. (1998). Efficiently mining long patterns from databases. In *SIGMOD'98* (pp 85–93).

Burdick, D., Calimlim, M., Flannick, J., Gehrke, J., & Yiu, T. (2005). Mafia: A maximal frequent itemset algorithm. *IEEE TKDE Journal, 17*(11), 1490–1504.

Daniel, P., & Viorel, N. (2004), *Debugging and verification of expert systems*. IeTA Report Series.

Deng, Z. (2013). Mining top-rank-*k* erasable itemsets by PID_lists. *International Journal of Intelligent Systems, 28*, 366–379.

Deng, Z. H. (2014). Fast mining top-rank-k frequent patterns by using Node-lists. *Expert Systems with Applications, 41*(4), 1763–1768.

Deng, Z., & Fang, G. (2007). Mining top-rank-*k* frequent patterns. *ICMLC*, 851–856.

Deng, Z., & Wang, Z. (2010). A new fast vertical method for mining frequent patterns. *International Journal of Computational Intelligence Systems, 3*(6), 733–744.

Deng, Z. H., Wang, Z., & Jiang, J. J. (2012). A new algorithm for fast mining frequent itemsets using N-lists. *Science China Information Sciences, 55*(9), 2008–2030.

Fang, G., & Deng, Z. H. (2008). VTK: Vertical mining of top-rank-*k* frequent patterns. In *FSKD'08* (pp. 620–624).

Guil, F., Bosch, A., Túnez, S., & Marín, R. (2003). Temporal approaches in data mining. A case study in agricultural environment. In *EUROCAST'03* (pp. 185–195).

Han, J., Pei, J., & Yin, Y. (1999). Mining frequent patterns without candidate generation. In *SIGMOD'00* (pp. 1–12).

Han, J., Dong, G., & Yin, Y. (1999). Efficient mining of partial periodic patterns in time series database. In *ICDE'99* (pp. 106–115).

Han, J., Wang, J., Lu, Y., & Tzvetkov, P. (2002). Mining top-k frequent closed patterns without minimum support. In *ICDM'02* (pp. 211–218).

Jackson, P. (1999). *Introduction to expert systems* (3rd ed.). England, UK: Addison Wesley Longman Limited.

Lee, A. J. T., Wang, C. S., Weng, W. Y., Chen, Y. A., & Wu, H. W. (2008). An efficient algorithm for mining closed inter-transaction itemsets. *Data and Knowledge Engineering, 66*(1), 68–91.

Liu, B., Hsu, W., & Ma, Y. (1998). Integrating classification and association rule mining. In *KDD'98* (pp. 80–86).

Nguyen, G., Le, T., Vo, B., & Le, B. (2014). A new approach for mining top-rank-*k* erasable itemsets. In *ACIIDS'14* (pp. 73–82).

Nguyen, T. T. L., Vo, B., Hong, T. P., & Thanh, H. C. (2012). Classification based on association rules: A lattice-based approach. *Expert Systems with Applications, 39*(13), 11357–11366.

Nguyen, T. T. L., Vo, B., Hong, T. P., & Thanh, H. C. (2013). CAR-Miner: An efficient algorithm for mining class association rules. *Expert Systems with Applications, 40*(6), 2305–2311.

Pham, T. T., Luo, J., Hong, T. P., & Vo, B. (2014). An efficient method for mining non-redundant sequential rules using attributed prefix-trees. *Engineering Applications of Artificial Intelligence, 32*, 88–99.

Song, W., Yang, B., & Xu, Z. (2008). Index-BitTableFI: An improved algorithm for mining frequent itemsets. *Knowledge-Based Systems, 21*(6), 507–513.

Vo, B., Coenen, F., Le, T., & Hong, T.-P. (2013). A hybrid approach for mining frequent itemsets. In *SMC'13* (pp. 4647–4651).

Vo, B., Hong, T. P., & Le, B. (2012). DBV-Miner: A dynamic bit-vector approach for fast mining frequent closed itemsets. *Expert Systems with Applications, 39*(8), 7196–7206.

Vo, B., Hong, T. P., & Le, B. (2013). A lattice-based approach for mining most generalization association rules. *Knowledge-Based Systems, 45*, 20–30.

Vo, B., Le, T., Coenen, F., & Hong, T.-P. (2014). Mining frequent itemsets using N-list and subsume concepts. *International Journal of Machine Learning and Cybernetics*. http://dx.doi.org/10.1007/s13042-014-0252-2.

Vo, B., Le, T., Hong, T.-P., & Le, B. (2014a). Maintenance of a frequent-itemset lattice based on pre-large concept. In *KSE'13* (pp. 295–305).

Vo, B., Le, T., Hong, T.-P., & Le, B. (2014b). An effective approach for maintenance of pre-large-based frequent-itemset lattice in incremental mining. *Applied Intelligence*. http://dx.doi.org/10.1007/s10489-014-0551-z.

Wang, J. Y., Han, J., & Pei, J. (2003). CLOSET+: Searching for the best strategies for mining frequent closed itemsets. In *SIGKDD'03* (pp. 236–245).

Zaki, M. J. (2000). Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering, 12*(3), 372–390.

Zaki, M., & Gouda, K. (2003). Fast vertical mining using diffsets. In *SIGKDD'03* (pp. 326–335).