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 \( iNTK \) 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 \( iNTK \) and NTK in terms of mining time and memory usage for eight datasets. The experimental results show that \( iNTK \) is more efficient and faster than NTK.

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, & Marin, 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 substructure patterns 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 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 (\( \text{min}_\text{sup} \)) 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
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 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 iNTK. Unlike NTK, iNTK uses N-list structure with an improved N-list intersection function to reduce the run-time and memory-consuming. Moreover, iNTK 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 iNTK algorithm for mining top-rank-k frequent patterns is described in Section 4. Section 5 compares the performance of the iNTK 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, iNTK, 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 \( I = \{i_1, i_2, \ldots, i_n\} \) be a set of items, and \( DB = \{T_1, T_2, \ldots, T_n\} \) be a set of transactions, where \( T_i \ (1 \leq i \leq n) \) 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 \( R_X = |SUP_Y| \ (Y \subseteq I \text{ and } SUP_Y \geq SUP_X) \), 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 \( R_P \leq k \).

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 \( TR_k = |P|P \subseteq I \text{ and } R_P \leq k \).

**Example 1.** Dataset \( DB_2 \) 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 \( DB_2 \). 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 \( DB_2 \).

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a, b</td>
</tr>
<tr>
<td>2</td>
<td>a, b, c, d</td>
</tr>
<tr>
<td>3</td>
<td>a, c, e</td>
</tr>
<tr>
<td>4</td>
<td>a, b, c, e</td>
</tr>
<tr>
<td>5</td>
<td>c, d, e, f</td>
</tr>
<tr>
<td>6</td>
<td>c, d</td>
</tr>
</tbody>
</table>
Table 2

<table>
<thead>
<tr>
<th>Rank</th>
<th>Support</th>
<th>Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>{c}</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>{a}</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>{ca}, {ce}, {d}, {cd}, {b}, {ab}, {e}</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>{ce}, {ae}, {eh}, {cb}</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>{cbde}, {ceba}, {cde}, {cdef}, {ce}, {df}, {ced}, {df}, {adden}, {adb}, {df}, {f}, {be}, {cf}</td>
</tr>
</tbody>
</table>

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 DBe. Each rectangle represents a node. A pair 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 Ni has PP-codes, PPi = (⟨Ni.preorder, Ni.postorder⟩: Ni.count).

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

Example 3. Let PPi = ⟨(4, 6): 3⟩ and PPj = ⟨(7, 3): 2⟩. Based on Property 1, PPi is an ancestor of PPj because PPi.preorder = 4 < PPj.preorder = 7 and PPi.postorder = 6 > PPj.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 = ⟨⟨preorder, postorder⟩: count⟩. N-list of a frequent pattern is denoted by ⟨PP, PP, ..., PP⟩, where PP.preorder < PP2.preorder < … < PPn.preorder.

Function Construct-PPC-tree (DB)
1. Scanning DB, inserting all items and their supports to Ii.
2. Sort Ii 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 Ti in DB do
5. Sort all items in support descending order.
6. Call Insert_Tree(Ti, R).
7. Visit the PPC-tree to generate the preorder and the postorder values of each node by preorder traverse and postorder traverse, respectively. To traverse the PPC-tree in preorder, 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(Ti, R)
1. t ← the first element in Ti. Tr = Ti \ 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 null then call Insert_Tree(Ti, N).

Example 4. N-list of item b includes two PP-codes, namely ⟨(2, 0): 1⟩ and ⟨(5, 4): 2⟩. 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 PtX and PtX and their N-list NL1 = [PP1, PP2, …, PPm] and NL2 = [PP2, PP22, …, PP2n] respectively. The N-list of PtP2X is generated by the following rules:

(i) ∀PPi ∈ NL1 (1 ≤ i ≤ m) and PPj ∈ NL2 (1 ≤ j ≤ n), if PPi is the ancestor of PPj then add the PP-code ⟨⟨PPi.preorder, PPi.postorder⟩: PPj.count⟩ to N-list of PtP2X.

(ii) Check all PP-codes of N-list of PtP2X, merge the PP-codes has same preorder and postorder values.

As shown by Fig. 3, NL1(t) = [⟨(3, 10): 5⟩] and NL2(t) = [⟨(11, 7): 1⟩]. According to Definition 7, NL2(t) can be built as follows. ⟨(3, 10): 5⟩ is an ancestor of ⟨(11, 7): 1⟩; therefore, PP-code ⟨(3, 10): 1⟩ is added to NL2(t). Because there are no other elements in NL1(t), the processing is stopped. The final result is NL2(t) = [⟨(3, 10): 1⟩] (Fig. 4).
Property 2. Let \( P \) is a \( t \)-pattern and its N-list \( NLP = \{PP_1, PP_2, \ldots, PP_n\} \). The support of \( P \) is determined by \( SUP_P = PP_1.count + PP_2.count + \ldots + PP_n.count \).

Example 5. The N-list of \( a \) is \( NL = \{a\} \). Hence, \( SUP\{a\} = 1 + 3 = 4 \). To verify the support of \( a \), scanning \( DB_B \) 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:

\[
GX = \{T.ID: DB \cap X \}
\]

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

Example 6. For the example dataset \( DB_B \), \( G\{c\} = \{2,3,4,5,6\} \) 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 \( SSA \) is defined as follows:

\[
SSA = \{B \in I_1 \mid \text{SS}\{c\} \subseteq \text{SS}\{a\} \}
\]

Example 7. \( G\{c\} = \{3,4,5\} \) and \( G\{e\} = \{2,3,4,5,6\} \). \( c \in \text{SS}\{e\} \) because \( \text{SS}\{c\} \subseteq \text{SS}\{e\} \).

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, \( SUP\{e\} = SUP\{c\} = 3 \). Besides, \( X \), \( X_{\text{SS}} \) is also equal to \( SUP\{X_{\text{SS}}\} \). Therefore, \( \{ae\} \) is a frequent pattern with \( SUP\{ae\} = 2 \) and \( \{aec\} \) is also a frequent pattern and \( SUP\{aec\} = 2 \).

---

**Fig. 4.** N-list of \( \langle cf \rangle \) in Example 1.

**Fig. 5.** Improved N-list intersection function.

**Fig. 6.** Subsume index generation procedure.
4. iNTK 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 $A$, $SS_A = \{B \in I_1 | \forall PP_i \in NL_A, PP_j \in NL_B \text{ and } PP_j \text{ is an ancestor of } PP_i\}$.

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 $NL_C = \{(3, 10), 5\}$ and $NL_{\{c\}} = \{(7, 3), 1\}$, \{(8, 5), 1\}. According to Property 4, \{(7, 3), 1\}, \{(8, 5), 1\} and \{(10, 8), 1\} $\in NL_C$ are descendants of \{(3, 10), 5\} $\in NL_{\{c\}}$. Therefore, $c \in SS_{\{c\}}$.

Property 5. Let $A, B, C$ and $D \in I_1$. 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 iNTK (see Fig. 7). Besides, this concept also reduces memory usage because iNTK does not determine and store the N-lists associated with a number of frequent patterns to determine their supports.

iNTK employs $t$-patterns to explore $(t + 1)$-patterns. By using N-lists, iNTK 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 combining 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 do not belong to the subsume index of X. All the 2-pattern candidates which have 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 $2^n$ – $1$ subsets from its subsume index and generates new patterns by combining the $2^n$ – $1$ 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 the number of entries in Y. If the number of entries is larger than the number of entries in Y, 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.

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$_b$ (Table 5).

Step 3. iNTK finds 2-pattern candidates which are \{(de), \{be\}, \{ae\}, \{bd\}, \{ad\}, \{cb\}, \{ca\}\} with their N-list \{(9, 9), 1\}, \{(5, 4), 1\}, \{(4, 6), 2\}, \{(5, 4), 1\}, \{(4, 6), 1\}, \{(3, 10), 2\}, \{(3, 10), 3\} 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$_b$. Table 7 shows the results. The patterns with the support equal 1 are deleted from Tab$_b$.

Step 4. Find 3-pattern candidates and no 3-pattern candidates are generated. The process is stopped. The final Tab$_b$ is shown in Table 8.

5. Experimental results

This section compares iNTK to NTK algorithms in terms of mining time and memory usage for six datasets such as Chess, Connect, Mushroom, Pumsb, Retail, T1O4D100K and two synthetic datasets (generated by the LUCS-KDD data generator). 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 iNTK are slightly different. Input data for NTK are Node-lists converted from the original datasets. Input data for iNTK includes N-list and subsume indexes of 1-patterns. Although the time required to create input data for iNTK is longer than that for NTK, this does not significantly affect the efficiency of iNTK because this procedure done only once. Table 10 shows the time required for converting the datasets.

Figs. 8–15 show mining times of iNTK and NTK for the experimental datasets with various values of $k$. The results show that iNTK 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
subsume indexes. Therefore, iNTK 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 iNTK algorithms. The memory usage of iNTK is slightly greater than that of NTK.

Table 10
Dataset conversion times.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Time required for creating Node-list (s)</th>
<th>Time required for creating N-list and finding subsume index values (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess</td>
<td>0.274</td>
<td>0.391</td>
</tr>
<tr>
<td>Mushroom</td>
<td>0.275</td>
<td>0.392</td>
</tr>
<tr>
<td>Connect</td>
<td>3.439</td>
<td>5.748</td>
</tr>
<tr>
<td>T10I4D100K</td>
<td>83.031</td>
<td>114.236</td>
</tr>
<tr>
<td>Test990.99KD1</td>
<td>71.509</td>
<td>133.023</td>
</tr>
<tr>
<td>Test2K50KD1</td>
<td>158.524</td>
<td>280.208</td>
</tr>
<tr>
<td>Pumsb</td>
<td>313.710</td>
<td>521.534</td>
</tr>
<tr>
<td>Retail</td>
<td>2715.031</td>
<td>3893.403</td>
</tr>
</tbody>
</table>

Figs. 16–23 show maximum amounts of memory used by NTK and iNTK for various $k$ values. According to these charts, iNTK 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, iNTK do not stored N-list of these candidates. Besides, iNTK uses N-list instead of Node-list for reducing memory usage because N-list saves the shorter sequence PP-codes.
Table 11
Memory usage of iNTK and NTK.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>NTK (kB)</th>
<th>iNTK (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess</td>
<td>1972.105</td>
<td>1982.203</td>
</tr>
<tr>
<td>Mushroom</td>
<td>1.412</td>
<td>1.434</td>
</tr>
<tr>
<td>Connect</td>
<td>18.446</td>
<td>16.469</td>
</tr>
<tr>
<td>T10I4D100K</td>
<td>83.011</td>
<td>114.238</td>
</tr>
<tr>
<td>Test29099KD1</td>
<td>41249.050</td>
<td>41304.150</td>
</tr>
<tr>
<td>Test2950KD1</td>
<td>46579.808</td>
<td>46666.322</td>
</tr>
<tr>
<td>Pumsb</td>
<td>57.429</td>
<td>58.089</td>
</tr>
<tr>
<td>Retail</td>
<td>39.289</td>
<td>40.788</td>
</tr>
</tbody>
</table>

Fig. 15. Mining time of iNTK and NTK for Retail dataset.

Fig. 16. Memory usage of iNTK and NTK for Chess dataset.

Fig. 17. Memory usage of iNTK and NTK for Mushroom dataset.

Fig. 18. Memory usage of iNTK and NTK for Connect dataset.

Fig. 19. Memory usage iNTK and NTK for T10I4D100K dataset.

Fig. 20. Memory usage iNTK and NTK for T990.99K22.1 dataset.

Fig. 21. Memory usage iNTK and NTK for T2K50KD1 dataset.

Fig. 22. Memory usage iNTK and NTK for Pumsb dataset.

Fig. 23. Memory usage iNTK and NTK for Retail dataset.
6. Conclusion and future work

This paper presents an efficient improvement algorithm called nNTK to mine top-rank-k frequent patterns. The advantage of nNTK 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 nNTK directly mining in case of patterns belonged to top-rank-k table contain other 1-patterns in their subsume set. This causes nNTK consume less memory and runtime. Extensive experiments show that nNTK 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 nNTK 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 nNTK 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://fospect.tdt.edu.vn, under Grant FOSTECT.2014.BR.07.

References

Han, J., Pei, J., & Yin, Y. (1999). Mining frequent patterns without candidate generation. In SIGMOD’99 (pp. 1–12).
Han, J., Wang, J., Yu, J., & Tsiverkov, P. (2002). Mining top-k frequent closed patterns without minimum support. In ICDM’02 (pp. 211–218).