

## **An algorithm for fast mining top-rank-k frequent patterns based on Node-list data structure**

**Qian Wang<sup>1,2,3</sup>, Jiadong Ren<sup>1,2,\*</sup>, Darryl N Davis<sup>3</sup>, Yongqiang Cheng<sup>3</sup>**

<sup>1</sup>College of Information Science and Engineering, Yanshan University, China

<sup>2</sup>Computer Virtual Technology and System Integration Laboratory of Hebei Province, China

<sup>3</sup>Department of Computer Science, University of Hull, UK

### **Abstract**

Frequent pattern mining usually requires much run time and memory usage. In some applications, only the patterns with top frequency rank are needed. Because of the limited pattern number, quality of the results are even more important than time and memory consumption. A Frequent Pattern algorithm for mining **Top-rank-K** patterns, FP\_TopK, is proposed. It is based on Node-list data structure extracted from FTPP-tree. Each node is with one or more triple sets, which contain supports, preorder and postorder transversal orders for candidate pattern generation and top-rank-k frequent pattern mining. FP\_TopK uses the minimal support threshold for pruning strategy to guarantee that each pattern in the top-rank-k table is really frequent and this further improves the efficiency. Experiments are conducted to compare FP\_TopK with iNTK and BTK on four datasets. The results show that FP\_TopK can achieve a better performance.

**Keywords:** data mining, frequent pattern, top-rank-k frequent pattern, FTPP-tree, Node-list

### **1. Introduction**

The task of frequent pattern mining is to discover the relationships between items in a dataset. It is important to build knowledge base, which is the basic components of expert systems (Sadik, 2008) or decision support systems (Chae et al., 2003). These systems can make contribution to

---

\* Corresponding author at: College of Information Science and Engineering, Yanshan University, Qianhuangdao, Hebei, China, 066000. Tel.:+8603358051850. E-mail addresses: [jdren@ysu.edu.cn](mailto:jdren@ysu.edu.cn). (Jiadong Ren)

an intelligent life for people by providing concise and accurate results. Previous frequent pattern mining algorithms are usually based on Apriori (Agrawal et al., 1994) and FP-growth (Han et al., 2000). Apriori algorithm employs candidate generation and test strategy to discover frequent patterns. It is expensive for repeatedly scanning the database and checking a large set of candidates. There are many improvements of Apriori (Shenoy et al., 2000; Zaki et al., 2003) that achieve good performance by reducing the candidate number and the database scanning times. FP-growth algorithm mines frequent patterns by divide-and-conquer approach without candidate generation. It achieves better efficiency by adopting a condensed FP-tree data structure. More improved algorithms (Liu et al., 2004; Liu et al., 2007; Tanbeer et al., 2008) of FP-growth are followed. However, when the datasets are sparse, building FP-tree and conditional pattern bases recurrently make the methods inefficient. There are also other various data structures, like lattice-based algorithms (Vo et al., 2013; Vo et al., 2014) and node based algorithm (Le et al., 2014; Deng et al., 2016). Frequent pattern mining is still an active topic in data mining, ranging from various extended mining tasks (Deng et al., 2016; Vo et al., 2017; Le et al., 2016) and a variety of new applications (Bieshoy et al., 2010; Wei et al., 2015). In general, frequent pattern mining needs a minimal support threshold to generate real frequent patterns. Whether the threshold is large or small, there will be too many frequent patterns for those applications such as the expert systems and so on. Because only a small scale of the frequent patterns are used in the final results. To this point, TFP algorithm (Wang et al., 2005) is proposed for mining top-k frequent closed patterns, and k is the desired number of frequent closed patterns to be mined. It does not use a minimal support threshold, and a threshold  $min\_l$  is set as the minimal length of each pattern. However, it is not easy to decide the value of  $min\_l$  and new algorithms for top-rank-k pattern mining are proposed to solve the problem. FAE (Deng et al., 2007) and VTK (Fang et al., 2008) select patterns according to their frequency rank instead of using  $min\_l$ . FAE reduces the searching space by using heuristic rules. It filters

the undesired patterns and the useful patterns are selected for pattern extension. VTK performs better than FAE because it gets the desired results by intersecting the Tid-lists of candidate frequent patterns without scanning the entire dataset. NTK algorithm (Deng et al., 2014) is built for mining top-rank-k frequent patterns using a Node-list structure extracted from a PPC-tree, which is helpful for reducing the run time and memory consumption, but NTK must always generate and test all the candidates. iNTK (Huynh-Thi-Le et al., 2015) uses an improved N-list structure and employs the subsume index without candidate generation to achieve higher efficiency, however, it costs lots of time for finding subsume index especially when the dataset is sparse. These algorithms mine top-rank-k frequent patterns without minimal support threshold. BTK (Dam et al., 2016) employs a TB-tree structure and a B-list structure for mining top-rank-k frequent patterns, and pruning strategy with minimal support threshold is also used, but its efficiency is also dropped if the database is sparse.

This paper presents top-rank-k frequent pattern mining algorithm using Node-list data structure, called FP\_TopK. With a minimal support threshold, a frequent pattern must satisfy the minimal support requirement. FPHP-tree is built by frequent 1-patterns instead of using all the 1-patterns like PCC-tree. The number of Node-lists are reduced because there are less nodes in the FPHP-tree. The minimal support threshold works for pruning infrequent patterns and guaranteeing the quality of the top-rank-k patterns. It can interact with value of rank k. When there is specific requirement for frequency, minimal support plays a leading role. And its value can be adjusted not to affect the top-k result when rank k is dominant.

The remainder of the paper is organized as follows. Section 2 introduces the problem definition. Section 3 develops FP\_TopK algorithm and gives some examples. Section 4 presents our performance study. Section 5 contains concluding remarks.

## **2. Problem definition**

Let  $I=\{I_1, I_2, I_3 \dots I_m\}$  be a set of items, and  $DB=\{T_1, T_2, T_3 \dots T_n\}$  be a database, where each transaction  $T_i(1<i<n)$  is a set of items such that  $T_i \subseteq I$ . Given a pattern  $P$ , it is said that  $T$  contains  $P$  if  $P \subseteq T$ . If  $P$  contains  $t$  items,  $P$  is a  $t$ -pattern. Given a database  $DB$  and a pattern  $P$ . The support of the  $P$  in  $DB$ , denoted as  $Sup(P)$ , is the number of the transactions containing  $P$ . A pattern  $P$  is a frequent pattern if  $Sup(P)$  is no less than a minimal support  $min\_sup$ . The  $min\_sup$  is calculated by  $\xi * |DB|$  where  $\xi$  is a given threshold and  $|DB|$  is the number of transaction in  $DB$ .

### 2.1. Problem of top-rank-k frequent patterns

Deng et al.[28] described the problem of mining top-rank-k frequent patterns, and here some improvements are made.

**Definition 1 (The rank of a pattern).** Given a transaction database  $DB$  and a pattern  $A(A \subseteq I)$ ,  $R_A$ , the rank of  $A$ , is defined as  $R_A = |\{Sup(X) | X \subseteq I \text{ and } Sup(X) \geq Sup(A)\}|$ , where  $|Y|$  is the number of elements in  $Y$ .

**Definition 2 (Top-rank-k frequent patterns).** Given a transaction database  $DB$ , a rank threshold  $k$ , and minimal support  $min\_sup$ , a pattern  $A(A \subseteq I)$  is called to be a top-rank-k frequent pattern if and only if  $R_A \leq k$  and  $Sup(A) \geq min\_sup$ .

**Property 1 (Anti-monotone).** If  $A$  is not a top-rank-k frequent pattern, any pattern  $B$  containing  $A$ , which is also called superset of  $A$ , cannot be a top-rank-k frequent pattern.

**Definition 3 (Top-rank-k frequent table).** A top-rank-k frequent table  $Tab_k$  records the top-rank-k frequent patterns with their rank and support.

Patterns with the same support are stored in the same entry of  $Tab_k$ . The number of entries in the top-rank-k table is no more than the threshold  $k$ .

**Example 1.** Table 1 shows a database  $DB$ .  $Sup(c)=5$ , because there are five transactions which contain  $c$ . Table 2 shows the supports and the ranks of all the 1-patterns. According to Table 2,

Sup(c) is the biggest, so  $R_c=1$ . Let rank=4, min\_sup=2. Tab<sub>4</sub> for the frequent 1-pattern can be obtained after deleting the fourth row of Table 2. Although  $R_f=4$ , it is not a frequent 1-pattern.

Table 1. A database DB.

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. Rank of the items in DB.

Rank	Support	Items
1	5	c
2	4	a
3	3	b,e,d
4	1	f

## 2.2. Node-list structure

Deng et al.[28] presented the PPC-tree structure, and Node-list is extracted from PPC-tree. FP\_TopK algorithm also uses Node-list structure, and it is created from FTTPP-tree.

**Definition 4 (FTTPP-tree).** FTTPP-tree is a tree structure which includes one root and a set of top-rank-k frequent 1-pattern nodes. Each node N is composed of five values: N.name, N.child, N.count, N.pre and N.post. N.name is the 1-pattern name, N.child is all the children of node N, N.count is Sup(N.name), N.pre and N.post are the preorder and postorder of node N, respectively. The root of FTTPP-tree which is named R has R.name=null and R.count=0.

**Example 2.** Table 3 shows the sorted database, and item f is removed. Fig.1 shows the FTTPP-tree. In each rectangle, there are name and support of the 1-pattern. Next to the rectangle, preorder and postorder of the 1-pattern are stored in pairs in a bracket.

Table 3. The sorted database DB.

TID	Sorted Items
1	a,b
2	c,a,b,d
3	c,a,e
4	c,a,b,e
5	c,d,e
6	c,d

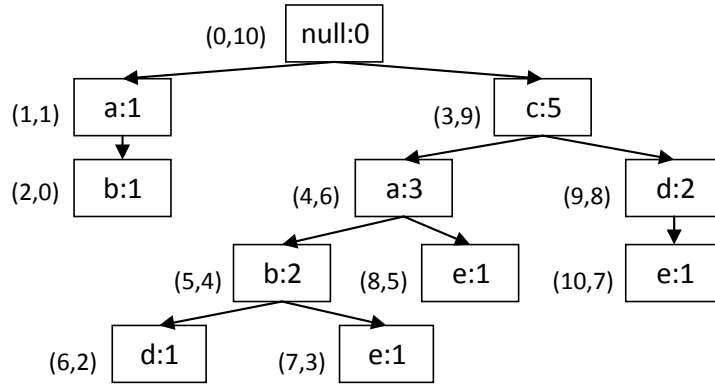


Fig.1 FPHP-tree for DB.

**Definition 5 (Node-list of a top-rank-k frequent 1-pattern).** Given a FPHP-tree, Node-list of a top-rank-k frequent 1-pattern A is a sequence of all the PP-codes of nodes in the FPHP-tree whose name is A. In an Node-list, PP-codes are arranged in preorder ascending order. Each PP-code in Node-list is denoted by  $PP = \langle (pre, post): count \rangle$ . Node-list of a top-rank-k frequent pattern is denoted by  $\{PP_1, PP_2, \dots, PP_n\}$ , where  $PP_1.pre < PP_2.pre < \dots < PP_n.pre$ .

**Property 2 (Ancestor-descendant relationship of 1-pattern PP-codes).** Given  $PP_i$  and  $PP_j$  are two PP-codes,  $PP_i$  is an ancestor of  $PP_j$  if and only if  $PP_i.pre < PP_j.pre$  and  $PP_i.post > PP_j.post$ .

**Example 3.** Node-list of 1-pattern b contains two PP-codes,  $b.PP_1 = \langle (2,0):1 \rangle$  and  $b.PP_2 = \langle (5,4):2 \rangle$ . They are arranged in preorder ascending order. Fig.2 shows the Node-lists of all the top-rank-k frequent 1-patterns. PP-code for 1-pattern c is  $c.PP_1 = \langle (3,9):5 \rangle$ , PP-codes for a is  $a.PP_1 = \langle (1,1):1 \rangle$  and  $a.PP_2 = \langle (4,6):3 \rangle$ .  $c.PP_1$  is an ancestor of  $a.PP_2 = \langle (4,6):3 \rangle$  because  $c.PP_1.pre = 3 < a.PP_2.pre = 4$ ,  $c.PP_1.post = 9 > a.PP_2.post = 6$ . But  $c.PP_1$  is not an ancestor of  $a.PP_1$ .

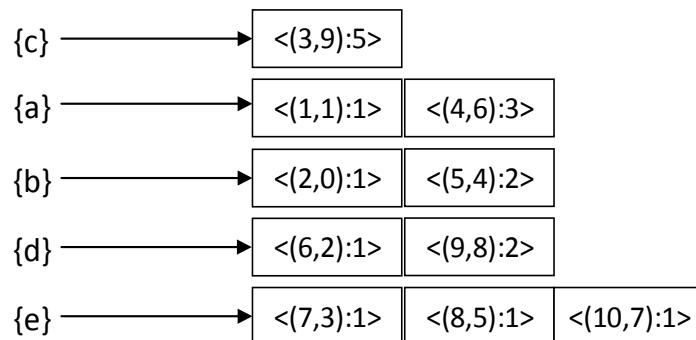


Fig.2 Node-lists of all the 1-patterns in Tab.4.

**Definition 6 (Node-list of a top-rank-k frequent t-pattern).** Let two t-patterns be  $PX_1$  and  $PX_2$ , where  $PX_1.Node-list=\{PP_{11},PP_{12},\dots,PP_{1m}\}$  and  $PX_2.Node-list=\{PP_{21},PP_{22},\dots,PP_{2n}\}$ , respectively.  $\forall PP_i \in PX_1.NL(1 \leq i \leq m)$  and  $PP_j \in PX_2.NL(1 \leq j \leq n)$ , if  $PP_i.pre < PP_j.pre$  and  $PP_i.post > PP_j.post$ , add the PP-code  $PX_2.Node-list[j]=\langle(PP_j.pre, PP_j.post):PP_j.count\rangle$  to Node-list of  $PX_1X_2$ .

**Property 3 (Support acquisition).** P is a t-pattern and its Node-list  $P.Node-list=\{PP_1,PP_2,\dots,PP_n\}$ . The support of P is determined by  $Sup(P)=PP_1.count+\dots+PP_n.count$ .

**Example 4.** As shown in Fig.3. Take 1-pattern c and a as an example.  $c.Node-list[1]=\langle(3,9):5\rangle$  and  $a.Node-list[1]=\langle(1,1):1\rangle$ ,  $a.Node-list[2]=\langle(4,6):3\rangle$ . The ca.Node-list generation of candidate 2-pattern ca is as follows.

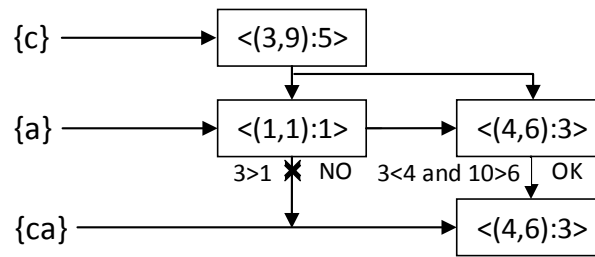


Fig.3 Node-list generation of 2-pattern ca

(1)  $c.Node-list[1].pre=3 > a.Node-list[1].pre=1$ . Connection is failed and it is then moved to  $a.Node-list[2]$ .

(2)  $c.Node-list[1].pre=3 < a.Node-list[2].pre=4$ ,  $c.Node-list[1].post=9 > a.Node-list[2].post=6$ .

Then  $ca.Node-list[1]=a.Node-list[2]=\langle(4,6):3\rangle$ ,  $Sup(ca)=3$ .

**Example 5.** As shown in Fig.4. Take 2-pattern ca and ce as an example.  $ca.Node-list[1]=\langle(4,6):3\rangle$  and  $ce.Node-list[1]=\langle(7,3):1\rangle$ ,  $ce.Node-list[2]=\langle(8,5):1\rangle$ ,  $ce.Node-list[3]=\langle(10,7):1\rangle$ . The cae.Node-list generation of candidate 3-pattern cae is as follows.

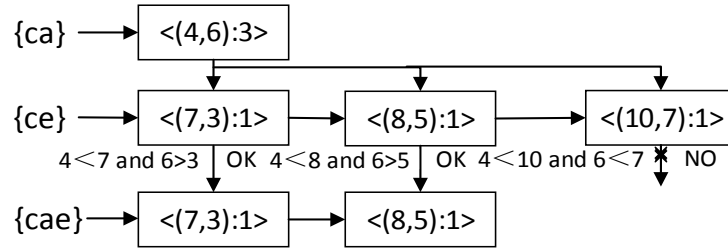


Fig.4 Node-list formation of 3-pattern cae

- (1)  $ca.Node\text{-}list[1].pre=4 < ce.Nnode\text{-}list[1].pre=7$ , and  $ca.Node\text{-}list[1].post=6 > ce.Node\text{-}list[1].post=3$ . Connection is OK.
- (2)  $ca.Node\text{-}list[1].pre=4 < ce.Node\text{-}list[2].pre=8$  and  $ca.Node\text{-}list[1].post=6 > ce.Node\text{-}list[2].post=5$ . OK.
- (3)  $ca.Node\text{-}list[1].pre=4 < ce.Node\text{-}list[3].pre=10$  and  $ca.Node\text{-}list[1].post=6 < ce.Node\text{-}list[3].post=7$ . Failed.
- (4)  $cae.Node\text{-}list[1]=ce.Node\text{-}list[1]=\langle(7,3):1\rangle$ ,  $cae.Node\text{-}list[2]=ce.Node\text{-}list[2]=\langle(8,5):1\rangle$ ,  
 $Sup(cae)=1+1=2$ .

### 3. FP\_TopK algorithm

#### 3.1. Node-list intersection function

In FP\_TopK algorithm, Node-list is extracted from FTTP-tree, so the function for Node-list intersection is presented after the FTTP-tree construction algorithm.

Function **FTTP-tree-construction**(DB, min\_sup, k)

1. Scan DB and insert all frequent items whose supports are larger than min\_sup and their supports to  $L_1$ .
2. Sort  $L_1$  in support descending order and insert the top k items into  $Tab_k$ . If the supports of some items are equal, they have the same rank and the orders among them in the transactions can be assigned by alphabetical order.
3. Create the root of a FTTP-tree, R, and name it as null.
4. For each transaction  $T_r$  in DB do
5. Sort all items in  $Tab_k$  according to the support descending order.
6. Call **Insert\_Tree**( $T_r$ , R).
7. Visit the FTTP-tree to generate the preorder and the postorder values of each node by preorder traverse and postorder traverse, respectively.

Function **Insert\_Tree**( $T_r$ , R)

1. t is the first element in  $T_r$ ,  $T_r=T_r \setminus t$ .
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  $T_r$  is not null then call **Insert\_Tree**( $T_r$ , N).

Function **NL-intersection**( $NL_1, NL_2$ )

1.  $NL_3 = \emptyset$ ;
2. Let  $i=0, j=0$ ;
3. While  $i < |NL_1|$  and  $j < |NL_2|$  do
4.     If  $NL_1[i].pre < NL_2[j].pre$
5.         If  $NL_1[i].post > NL_2[j].post$
6.             Add the tuple ( $NL_2[j].pre, NL_2[j].post$ ):  $NL_2[j].count$ ) to  $NL_3$ ;
7.              $j++$ ;
8.         Else
9.              $i++$ ;
10.     Else
11.          $j++$ ;
12. Return  $NL_3$ ;

### 3.2. The processing procedure of FP\_TopK algorithm

FP\_TopK algorithm explores (t+1)-patterns from t-patterns. By using FTTP-tree structure to extract Node-lists, FP\_TopK algorithm can get the supports of (t+1)-patterns without repeatedly scanning the database. When the PP-codes of two different Node-lists don't match each other, they can immediately know the problem and decide who should move ahead. This can reduce the comparing times and further decrease the run time. The FP\_TopK algorithm and the processing procedure are as follows.

Procedure **FP\_TopK**(DB, k)

1. Call **FTTP-tree-construction**(DB, min\_sup, k) to build FTTP-tree
2. Determine the Node-list of  $Tab_k$  (top-rank-k frequent 1-patterns)
3. Find 1-patterns in  $Tab_k$  denote the set of these 1-patterns as  $TR_1$
4. For( $j=2; TR_{j-1} \neq \emptyset; j++$ )
5.      $CR_j = \text{Candidate\_gen}(TR_{j-1})$
6.     For each  $C \in CR_j$ , C is generated by  $P_1 (\in TR_{j-1})$  and  $P_2 (\in TR_{j-1})$
7.          $C.Node\text{-list} = \text{Node-list-intersection}(P_1.Node\text{-list}, P_2.Node\text{-list})$
8.         If  $Sup(C)$  is equal to the support of the patterns in any entry of  $Tab_k$
9.             Insert C with its support into the same rank of  $Tab_k$  together with other patterns who have the same support
10.         If  $Sup(C)$  is larger than min\_sup and smaller than the support of the patterns in any entry of  $Tab_k$  and the number of the entries is less than k
11.             Insert C into the  $Tab_k$  as the last rank pattern
12.         If  $C \in Tab_k$  then  $TR_j = TR_j \cup \{C\}$

Procedure **Candidate\_gen**( $TR_{j-1}$ )

1.  $CR_j = \emptyset$ ;
2. For each  $C_u \in TR_{j-1}$
3.     For each  $C_v \in TR_{j-1} (v > u)$
4.     If  $(C_u[1]=C_v[1] \wedge C_u[2]=C_v[2] \wedge \dots \wedge C_u[j-2]=C_v[j-2] \wedge C_u[j-1] \neq C_v[j-1])$
5.      $C = C_u[1]C_u[2] \dots C_u[j-2]C_u[j-1]C_v[j-1]$  ;
6.      $CR_j = CR_j \cup \{C\}$ ;
7. Return  $CR_j$ ;

(1) Scan the database, find the top-rank-k frequent 1-patterns and insert them into top-rank-k frequent table  $Tab_k$ .

(2) Sort the transactions according to the rank of the 1-patterns in  $Tab_k$  and form FTTP-tree.

(3) Scan the FTTP-tree and generate Node-lists of all 1-patterns in  $Tab_k$ .

(4) For each t-pattern X and another t-pattern Y in  $Tab_k$ , FP\_TopK finds all candidate (t+1)-patterns by combining X with Y, satisfying  $X[1]=Y[1] \wedge X[2]=Y[2] \wedge \dots \wedge X[t-2]=Y[t-2] \wedge X[t-1] \neq Y[t-1]$ ,  $X.NL[i].pre < Y.NL[j].pre$  and  $X.NL[i].post > Y.NL[j].post$ .

(5) Each candidate t-pattern whose support is equal to any entry of  $Tab_k$  is inserted to the entry. If the support is larger than  $min\_sup$  and less than the smallest support of the  $Tab_k$ , and the number of entries in  $Tab_k$  is no more than k, then the t-pattern will be inserted into the last entry of  $Tab_k$ . The supports of the t-patterns cannot be larger than any entries of  $Tab_k$ , because the support of their supper patterns must be larger or equal to theirs.

(6) Repeat steps 4 and 5 until no new candidate patterns can be generated.

### 3.3. An illustrative example

Let rank=4,  $min\_sup=2$ . The process of mining top-rank-4 frequent patterns from the database DB in Table 1 is as follows.

(1) Find top-rank-4 frequent patterns and form the top-rank-4 frequent table  $Tab_4$ . The result is shown in Table 2 without the last row.

(2) Sort the transactions in DB and form the FTTP-tree as shown in Table 3 and Fig.1.

(3) Extract Node-list from FTTP-tree as shown in Fig.2.

(4) Candidates are generated as shown in Fig.3 and Fig.4. Candidate 2-patterns and their Node-lists are listed as shown in Fig.5.

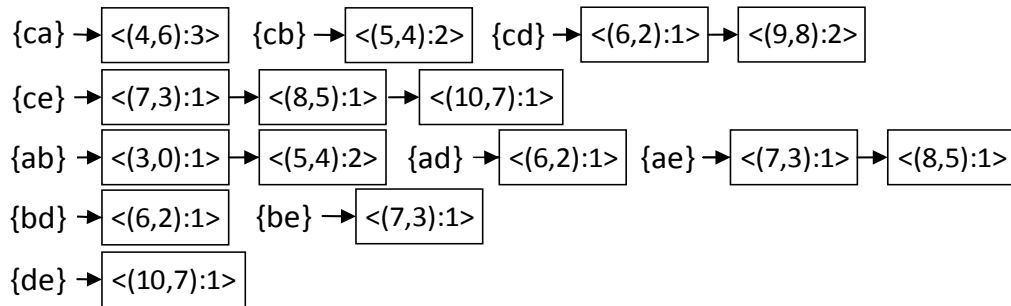


Fig.5 Node-lists for candidate 2-patterns.

Sup(ca)=3, Sup(cd)=3, Sup(ce)=3, Sup(ab)=3. Sup(cb)=2, Sup(ae)=2.

Sup(de)=1, Sup(ad)=1, Sup(bd)=1, Sup(be)=1.

2-patterns de, ad and bd are infrequent because their supports are smaller than min\_sup. 2-patterns ca, cd, ce, ab are inserted into the third rank entry of Tab<sub>4</sub>. 2-patterns cb, ae are inserted into the fourth rank entry of Tab<sub>4</sub> as shown in Table 4.

Table 4. Tab<sub>4</sub> after 2-patterns inserted.

k	Sup(X)	Patterns
1	5	c
2	4	a
3	3	b,d,e,ca,cd,ce,ab
4	2	cb,ae

Node-lists for candidate 3-patterns are shown in Fig.6.

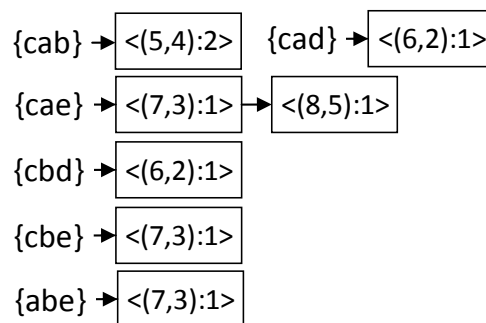


Fig.6 Node-lists for candidate 3-patterns.

Sup(cab)=2, Sup(cae)=2. Sup(cad)=1, Sup(cbd)=1, Sup(cbe)=1, Sup(abe)=1.

3-patterns cad, cbd and cbe are infrequent because their supports are smaller than min\_sup. 3-patterns cab, cae are inserted into the fourth rank entry of Tab<sub>4</sub>. Table 5 is the final results.

Table 5. Final results for Tab<sub>4</sub>.

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. Experiment results

Experiments are performed on a PC with Intel(R) Core(TM) 3.6 GHz CPU and 16G main memory, running on Windows 8. We evaluate the run time of FP\_TopK algorithm, and compare it with iNTK and BTK using different rank k thresholds. Minimal support threshold is used in FP\_TopK and BTK, but not in iNTK. There are three real datasets Connect, Mushroom and Retail downloaded from <http://fimi.ua.ac.be/data/> and a synthetic dataset T25I10D100K generated by the IBM data generator. To test the algorithms in the same coding environment, all the programs are written in C++ using Visual Studio 2013. Table 6 shows the characteristics of these datasets, including the average transaction length, the item number and the transaction number.

Table 6. Characteristics of the experimental datasets.

Dataset	Average transaction length	Item number	Transaction number
Connect	43	130	67557
Mushroom	23	119	8124
Retail	10.3	16470	88162
T25I10D100K	25	990	99822

The run time comparison of FP\_TopK with iNTK and BTK is shown in Figs.7-10. It is noted that, here run time means the total execution time, from input to output. The min\_sup is set to 5%, which is as small as possible not to affect the mining results of top-rank-k patterns(Retail dataset.is affected). Then run time of the two algorithm can be fairly compared.

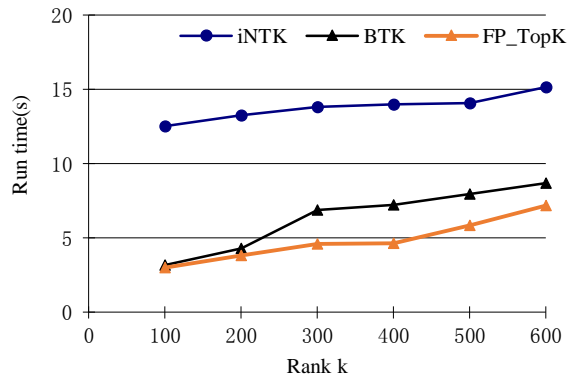


Fig.7. Run time on Connect dataset.

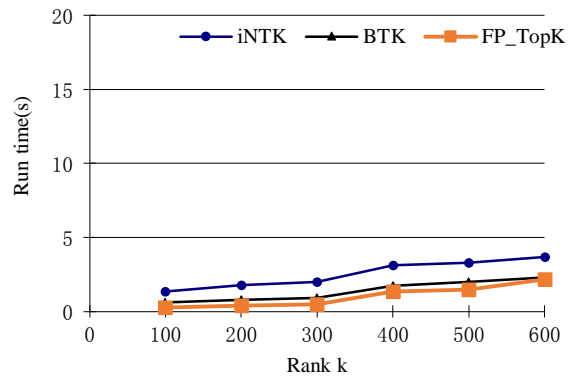


Fig.8. Run time on Mushroom dataset.

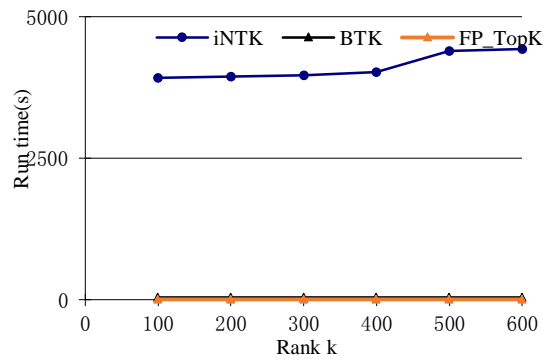


Fig.9. Run time on Retail dataset.

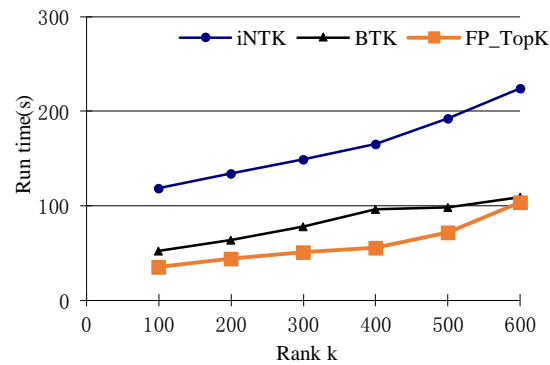


Fig.10. Run time on T25I10D100K dataset.

The experimental results show that FP\_TopK outperforms iNTK more and BTK less for all the values of k. This is because FP\_TopK can save time in every link than iNTK. At the beginning, less nodes are built in the FTPP-tree than PPC-tree. This can lead to less Node-list creation, calculation and frequent top-rank-k table insertion. Although iNTK uses subsume indexes to avoid unnecessary connection, subsume index generation requires a cost of time and the cost is even worse for small value of rank k or sparse datasets. BTK is similar to iNTK as they use the same subsume index strategy, but it also uses the min\_sup to do early pruning, so its performance is better than iNTK and close to FP\_TopK. When the value of rank k is small or the dataset is sparse, iNTK is not effective and BTK is more or less interrupted.

FP\_TopK performs well in both dense and sparse datasets. It prunes infrequent patterns in the early stage, and more infrequent patterns will be pruned in the later process for sparse datasets. Among the four datasets, the two real datasets Connect and Mushroom are very dense, another real dataset Retail and the synthetic dataset T25I10D100K are much sparser. In Fig.9, it is irregular that iNTK runs much slower than FP\_TopK. This is because the data characteristic of Retail dataset is too sparse. Even the min\_sup is small, a lot of the infrequent patterns are pruned by FP\_TopK at the beginning, however, these infrequent patterns contain many elements in the subsume indexes, and most of the run time in iNTK is for generating the subsume indexes. The numbers of frequent 1-patterns and top-rank-100 frequent patterns for each real dataset and algorithm are shown in Table 7. For FP\_TopK and BTK, the pattern contained in  $L_1$  is smaller than  $Tab_{100}$ , that mean less patterns need to be frequency-checked and less candidates need to be generated. For iNTK, lots of unwanted patterns and their growth patterns will first enter and then leave the  $Tab_{100}$ . That is why the efficiency of iNTK is the lowest.

Table 7. Number of returned item sets for each real dataset(min\_sup=5%, rank=100)

Algorithm	FP_TopK	iNTK	BTK
-----------	---------	------	-----

Dataset	(L <sub>1</sub> /Tab <sub>100</sub> )	(L <sub>1</sub> /Tab <sub>100</sub> )	(L <sub>1</sub> /Tab <sub>100</sub> )
Connect	86/100	100/100	86/100
Mushroom	73/100	100/100	73/100
Retail	6/16	100/100	6/16

It seems that, if the value of rank  $k$  is small, the patterns with small support usually have less chance to enter the top-rank- $k$  table. In fact, the support is large or small is relative. If there are too many small supports, then patterns with poor quality will insert into the table.  $\text{Min\_sup}$  is necessary to guarantee that each pattern in the top-rank- $k$  table is meaningful. Whether it affects the mining result or not, it can be effectively helpful to save run time.

## 5. Conclusion

This paper presents an algorithm called FP\_TopK for fast mining top-rank- $k$  frequent patterns based on Node-list data structure. FP\_TopK prunes the infrequent items and builds FPHP-tree with less nodes. This further avoids unnecessary Node-list creation and frequent top-rank- $k$  table insertion, which makes the run time and memory usage reduced. FP\_TopK is effective for both dense and sparse datasets. The minimal support threshold can satisfy the frequency requirement and guarantee the quality of the patterns in the top-rank- $k$  table. Even if it is small enough not to affect the top-rank- $k$  patterns, it is helpful for efficiency improvement. After analyzing the experimental results, FP\_TopK is proved to be efficient.

## Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant No.61472341, No.61572420, and the Natural Science Foundation of Hebei Province P.R.China under Grant No.F2014203152, No.F2016203330. It is also supported by China Scholarship Council. The authors are grateful to valuable comments and suggestions of the reviewers.

## Disclosure Statement

No potential conflict of interest was reported by the authors.

## Reference

- Agrawal, R., & Srikant, R. (1994, September). Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB* (Vol. 1215, pp. 487-499).
- Bieshoy Awad, Janaka Ekanayake, & Nick Jenkins. (2010). Intelligent load control for frequency regulation in microgrids. *Intelligent Automation & Soft Computing*, 16(2), 303-318.
- Chae, Y. M., Kim, H. S., Tark, K. C., Park, H. J., & Ho, S. H. (2003). Analysis of healthcare quality indicator using data mining and decision support system. *Expert Systems with Applications*, 24(2), 167-172.
- Dam, T. L., Li, K., Fournier-Viger, P., & Duong, Q. H. (2016). An efficient algorithm for mining top-rank- k, frequent patterns. *Applied Intelligence*, 45(1), 96-111.
- Deng, Z. H., & Fang, G. D. (2007, August). Mining top-rank-K frequent patterns. In *Machine Learning and Cybernetics, 2007 International Conference on* (Vol. 2, pp. 851-856). IEEE.
- 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. H. (2016). Diffnodesets: an efficient structure for fast mining frequent itemsets. *Computer Science*, 41, 214-223.
- Fang, G. D., & Deng, Z. H. (2008, October). VTK: Vertical mining of top-rank-k frequent patterns. In *Fuzzy Systems and Knowledge Discovery, 2008. FSKD'08. Fifth International Conference on* (Vol. 2, pp. 620-624). IEEE.
- Grahne, G., & Zhu, J. (2005). Fast algorithms for frequent itemset mining using fp-trees. *Knowledge and Data Engineering, IEEE Transactions on*, 17(10), 1347-1362.
- Han, J., Pei, J., & Yin, Y. (2000, May). Mining frequent patterns without candidate generation. In *ACM SIGMOD Record* (Vol. 29, No. 2, pp. 1-12). ACM.
- Huynh-Thi-Le, Q., Le, T., Vo, B., & Le, B. (2015). An efficient and effective algorithm for mining top-rank-k frequent patterns. *Expert Systems with Applications*, 42(1), 156-164.
- Le, T., & Vo, B. (2014). MEI: an efficient algorithm for mining erasable itemsets. *Engineering Applications of Artificial Intelligence*, 27, 155-166.
- Lee, G., & Yun, U. (2016). A new efficient approach for mining uncertain frequent patterns using minimum data structure without false positives. *Future Generation Computer Systems*, 68, 89-110.
- Liu, G., Lu, H., Lou, W., Xu, Y., & Yu, J. X. (2004). Efficient mining of frequent patterns using ascending frequency ordered prefix-tree. *Data Mining and Knowledge Discovery*, 9(2), 249-274.
- Liu, G., Lu, H., & Yu, J. X. (2007). CFP-tree: A compact disk-based structure for storing and querying frequent itemsets. *Information Systems*, 32(2), 295-319.
- Shenoy, P., Haritsa, J. R., Sudarshan, S., Bhalotia, G., Bawa, M., & Shah, D. (2000, May). Turbo-charging vertical mining of large databases. In *ACM SIGMOD Record* (Vol. 29, No. 2, pp. 22-33). ACM.
- Sadik, A. T. (2008). Premises reduction of rule based expert systems using association rules technique. *International Journal of Soft Computing*, 3(3), 195-200.
- Tanbeer, S. K., Ahmed, C. F., Jeong, B. S., & Lee, Y. K. (2008, October). Efficient frequent pattern mining over data streams. In *Proceedings of the 17th ACM conference on Information and knowledge management* (pp. 1447-1448). ACM.
- 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., Hong, T. P., & Le, B. (2014). An effective approach for maintenance of pre-large-based frequent-itemset lattice in incremental mining. *Applied Intelligence*, 41(3), 759-775.
- Vo, B., Sang, P., Le, T., & Deng, Z. H. (2017). A novel approach for mining maximal frequent patterns. *Expert Systems with Applications*, 73, 178-186.

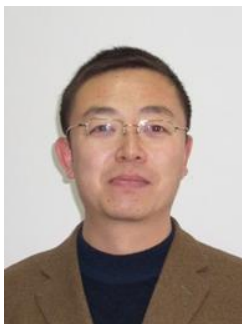


- Wang, J., Han, J., Lu, Y., & Tzvetkov, P. (2005). TFP: An efficient algorithm for mining top-k frequent closed itemsets. *Knowledge and Data Engineering, IEEE Transactions on*, 17(5), 652-663.
- Wei, X., Wang, Y., Li, Z., Zou, T., & Yang, G. (2015). Mining users interest navigation patterns using improved ant colony optimization. *Intelligent Automation & Soft Computing*, 21(3), 445-454.
- Zaki, M. J., & Gouda, K. (2003, August). Fast vertical mining using diffsets. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 326-335). ACM.

## Notes and contributions



Qian Wang received a Ph. D. degree in Computer Software and Theory from Yanshan University, China, in June 2016. Since 2016, she has been a lecturer at the College of Information Science and Engineering, Yanshan University, China. Her current research interests include data mining, software security.



Jiadong Ren received a Ph. D. degree in Computer Application Technology from Harbin Institute of Technology, China, in December 1999. Since 2005, he has been a professor at the College of Information Science and Engineering, Yanshan University, China. His current research interests include data mining, software security.



Darryl N Davis received a Ph. D. degree in Wolfson Image Analysis Unit from Victoria University of Manchester, United Kingdom, in June 1991. Since 1999, he has been a senior lecturer at the School of Engineering and Computer Science, University of Hull, United Kingdom. His current research interests include data mining, cognition and affect, agents and artificial life.



Yongqiang Cheng received a Ph. D. degree in Design and Technology from University of Bradford, United Kingdom, in June 2010. Since 2014, he has been a lecturer at the School of Engineering and Computer Science, University of Hull, United Kingdom. His current research interests include data mining, AI, embedded systems.