Computers, Materials & Continua CMC, vol.63, no.3, pp.1445-1469, 2020

GTK: A Hybrid-Search Algorithm of Top-Rank-*k* Frequent Patterns Based on Greedy Strategy

Yuhang Long[1], Wensheng Tang[1,*], Bo Yang[1,*], Xinyu Wang[2], Hua Ma[1], Hang Shi[1] and Xueyu Cheng[3]

Abstract: Currently, the top-rank-*k* has been widely applied to mine frequent patterns with a rank not exceeding *k*. In the existing algorithms, although a level-wise-search could fully mine the target patterns, it usually leads to the delay of high rank patterns generation, resulting in the slow growth of the support threshold and the mining efficiency. Aiming at this problem, a greedy-strategy-based top-rank-*k* frequent patterns hybrid mining algorithm (GTK) is proposed in this paper. In this algorithm, top-rank-*k* patterns are stored in a static doubly linked list called RSL, and the patterns are divided into short patterns and long patterns. The short patterns generated by a rank-first-search always joins the two patterns of the highest rank in RSL that have not yet been joined. On the basis of the short patterns satisfying specific conditions, the long patterns are extracted through level-wise-search. To reduce redundancy, GTK improves the generation method of subsume index and designs the new pruning strategies of candidates. This algorithm also takes the use of reasonable pruning strategies to reduce the amount of computation to improve the computational speed. Real datasets and synthetic datasets are adopted in experiments to evaluate the proposed algorithm. The experimental results show the obvious advantages in both time efficiency and space efficiency of GTK.

Keywords: Top-rank-*k* frequent patterns, greedy strategy, hybrid-search.

1 Introduction

The power of data mining facilitates every aspect of our lives, and some applications [Ruiz and Alisha (2019); Guo, Liu, Ren et al. (2019)] are very intuitive examples. The frequent patterns mining, one of the most popular areas of data mining research, was proposed by Agrawal et al. [Agrawal, Imieliński and Swami (1993)]. Its main task is to search for itemsets, sequences or structures with a support that is not less than the user-

[1] Hunan Provincial Key Laboratory of Intelligent Computing and Language Information Processing, Hunan Normal University, Changsha, 410081, China.

[2] College of Information Science and Engineering, Hunan University, Changsha, 410082, China.

[3] Clayton State University, Morrow, GA 30260, USA.

[*] Corresponding Authors: Wensheng Tang. Email: tangws@hunnu.edu.cn; Bo Yang. Email: yangbo@hunnu.edu.cn.

Received: 30 January 2020; Accepted: 26 February 2020.

specified minimum support threshold in the dataset, among which frequent itemsets are the most basic frequent patterns. Apriori [Agrawal and Srikant (1994)] is the first mining algorithm of frequent patterns and was followed by two other classic algorithms [Ogihara, Zaki, Parthasarathy et al. (1997); Han, Pei, Yin et al. (2004)]. Those three are representative algorithms for horizontal, vertical and trie layouts. In addition, several algorithms [Zaki and Gouda (2003); Tsay and Chiang (2005); Vo, Coenen, Le et al. (2013); Vo, Le, Coenen et al. (2016)] also effectively implemented this mining task. With more advanced researches, frequent pattern mining has spawned a variety of extended algorithms, such as:

(1) algorithms for closed frequent patterns mining [Zaki and Hsiao (2002); Wang, Han and Pei (2003); Fang, Wu, Li et al. (2015)];

(2) algorithms for maximal frequent patterns mining [Burdick, Calimlim, Flannick et al. (2005); Zeng, Pei, Wang et al. (2009); Yun and Lee (2016)];

(3) algorithms for high utility frequent patterns mining [Erwin, Gopalan and Achuthan (2007); Hu and Mojsilovic (2007); Yun and Ryang (2015)];

(4) algorithms for erasable frequent patterns mining [Hong, Lin, Lin et al. (2017); Le and Vo (2014); Nguyen, Le, Vo et al. (2015)].

Early mining algorithms of frequent patterns usually pre-set the support threshold, whose accuracy requires professional knowledge or experience and is too hard for ordinary users. If the threshold is set too high, the users' desired patterns cannot be fully detected. And conversely, many useless candidates are generated, which considerably reduces the mining efficiency and even causes the crash. To tackle this problem, Han et al. [Han, Wang, Lu et al. (2002)] proposed a top-*k* closed frequent pattern mining task, and designed TFP (top-*k* frequent closed patterns) algorithm [Wang, Han, Lu et al. (2005)]. Although no *min_sup* is not used in this algorithm, its core concept *min_l* is as difficult to predict as the traditional *min_sup*. In addition, due to the same support which may belong to different patterns, the final mining results may cause the missing of important patterns for users. Aiming at the two major defects of TFP, Deng et al. [Deng and Fang (2007)] introduced the concept of top-rank-*k* frequent patterns. In the circumstances, there's no need for *min_sup* and *min_l* to be predetermined. How to mine top-rank-*k* frequent patterns with high efficiency has received extensive attention in the industry and academia. In recent years, numerous improved algorithms [Fang and Deng (2008); Deng (2014); Huynh, Le, Vo et al. (2015); Dam, Li, Fournier et al. (2016); Wang, Ren, N Davis et al. (2017); Jia, Xiang and Liu (2018)] have been produced.

FAE (Vertical Mining of Top-rank-*k* Frequent Patterns, FAE) algorithm [Deng and Fang (2007)] adopts the horizontal data layout, uses heuristic rules and effective pruning strategies to reduce the mining space, and retains useful patterns for the expansion of long patterns. VTK (Vertical Mining of top-rank-k Frequent Patterns, VTK) algorithm [Fang and Deng (2008)] follows the level-wise-search method of FAE. In order to overcome the problem that many scans for dataset in FAE face, VTK adopts vertical data layout so that pattern information can be represented by Tid-list and the support of each pattern is obtained through calculating the length of its Tid-list. VTK achieves better results than FAE. However, when facing dense databases, VTK may experience performance degradation. To this end, Deng proposed the NTK (Fast mining top-rank-*k*

frequent patterns by using Node-lists, NTK) algorithm Deng (2014)] that compresses dataset into a PPC-tree like FP-Tree and conducts a layer-wise mining of top-rank-*k* frequent patterns by extracting 1-patterns' Node-list [Deng and Wang (2010)]. In spite of the improvement compared with FAE and VTK, NTK has not produced a strategy to narrow the search range. iNTK (top-rank-*k* frequent patterns mining algorithm based on subsume index and N-list, iNTK) [Huynh, Le, Vo et al. (2015)] is an NTK-based optimized algorithm whose core structure adopts N-list [Deng, Wang and Jiang (2012)]. N-list, an efficient structure, composed of prefix nodes, the length of which is always shorter than that of Node-list composed of suffix nodes, takes less memory than the Node-list. Moreover, iNTK reduces the scope of pattern mining and speeds up the mining process by introducing subsume index [Song, Yang and Xu (2008)]. iNTK was shown to outperform NTK and its advantages increase as the *k* value increases. Although iNTK has been optimized for lifting efficiency, it still leaves some following issues left for BTK to solve. In response to the above problems, BTK (top-rank-*k* frequent patterns mining algorithm based on TB-tree, BTK) [Dam, Li, Fournier et al. (2016)] gives some effective solutions: (1) It proposes a TB-tree and B-lists in which each node records its start-build and finish-build code to resolve the time-consuming construction of PPC-tree. (2) To avoid useless operations in iNTK, until the final candidates are found, patterns containing subsume indexes are combined. In the meantime, BTK also designs an EP (early pruning by threshold) strategy and an RSC (raising threshold by the support of candidates) strategy about B-list. An extensive experimental study has shown that BTK is superior to iNTK with a significant difference in the round [Dam, Li, Fournier et al. (2016)].

In summary, although a variety of algorithms have been proposed to achieve fast mining of top-rank-*k* frequent patterns, it is not yet efficient enough. Thus, without a high-performance method of top-rank-*k* frequent patterns mining, it is difficult to save runtime and memory usage. Consequently, this paper proposes an algorithm called GTK. Conclusively, GTK and BTK were compared through numerous experiments, and the experimental results reflect obvious advantages in both time efficiency and space efficiency of GTK.

The main contributions of this paper are as follows:

(1) Aiming at the problem that B-list takes up a lot of space and a long time for intersection function, an FPI (frequent pattern information) class is designed to represent pattern information by using vertical data structure. The FPI of a pattern includes three parts: pattern's items (Its), the subsume indexes of items (Si), and the bitset of pattern's Tids (Bs). The length of bitset is the support of the pattern. The Tids information represented by bitset is greatly compressed to save storage space. In the pattern mining, bitset performs bitwise AND operation for simplifying the intersection function of B-lists.

(2) To reduce the high maintenance cost of the top-rank-*k* table structure, a static doubly linked list structure named RSL (Static Doubly-Linked Lists of top-rank-*k*) is designed to store the top-rank-*k* frequent patterns. All nodes are listed in the RSL with a descending order of Support. As a linear table described by array, RSL only needs to modify cursor field instead of moving large numbers of elements when inserting and deleting nodes, which effectively reduces consumption and saves time cost.

(3) With regard to the defect of time consuming in level-wise-search, a hybrid mining

algorithm is designed. According to the length threshold, the pattern is divided into short patterns and long patterns. For mining short patterns, rank-first-search is proposed based on the greedy strategy. Taking all short patterns with a length equals the length threshold as the input, long patterns are generated through level-wise-search. The algorithm is so efficient that it promotes fast generation of high rank patterns and reduces massive invalid joins. At the same time, related strategies are also designed to prune candidates.

(4) For the shortcoming of inefficient use of computing resources in the generation process of subsume index, an optimization method is proposed: the process of finding subsume indexes is embedded in the rank-first-search, and the qualified subsume indexes will be searched when mining the 2-patterns, therefore, it avoids 1-patterns being scanned repeatedly so as to make full use of computing resources. This method is efficient despite facing the sparse or dense datasets.

The rest of this paper is organized as follows. The basic concepts are given in Section 2, highlights the problems associated with the current mainstream algorithms, and also conducts an analysis; Section 3 describes the construction and initialization of RSL; Section 4 details the design and implementation of the GTK algorithm, including the Ex_Short method, the Ex_Long method, the related pruning strategies and a simple example. Section 5 presents the experimental design, results and analysis; Section 6 draws a conclusion of the full text and forecasts the development tendency of future research.

2 Basic definitions and problem analysis

2.1 Problem of mining top-rank-k frequent patterns

Related definitions of frequent patterns can be found in Agrawal et al. [Agrawal and Srikant (1994)]. The relevant basic concepts and problem of the mining top-rank-k frequent patterns in the literature [7] are described below.

The Rank of a Pattern. Given a transaction database $\boldsymbol{D}$ and a pattern A ($A \subseteq I$), R_A, the rank of A, is defined below, where $|Y|$ is the number of elements in Y.

$$R_A = |\{X \square Sup \mid X \subseteq I \text{ and } X \square Sup \geq A \square Sup\}| \quad (1)$$

Top-rank-k Frequent Patterns. Given a transaction database $\boldsymbol{D}$ and a threshold k, a pattern A ($A \subseteq I$) is referred as a top-rank-k frequent pattern if and only if R_A is not greater that k. That is, $R_A \leq k$.

Top-rank-k Frequent Patterns Mining. Given a transaction database $\boldsymbol{D}$ and a threshold k, the top-k frequent patterns mining is the task of finding the complete set of frequent patterns whose ranks are not greater than k, that is, the set of top-rank-k frequent pattern is equal to $S_{top\text{-}k}$, the minimum support which is equal to the support threshold denoted as S^t in this paper:

$$S_{top\text{-}k} = |\{X \mid X \subseteq I \text{ and } R_x \leq k\}| \quad (2)$$

Example 1. Tab. 2 shows the support and rank of each pattern of $\boldsymbol{D_1}$ in Tab. 1. [c] has the largest support count, so R_c=1, and similarly, R_d=5. Assume that the rank threshold k=2, the top-rank-k frequent pattern set of the top two is {[c], [a], [e], [b], [b, c]}, then S^t equals 6.

Table 1: Transaction database D_1

Tids	List of items	Tids	List of items	Tids	List of items
1	c, a, b, d	4	e, b, c, a	7	a, e, c
2	a, b, c	5	c, e, b	8	a
3	b, c, e, d	6	c, e, a, b	9	e

Table 2: Rank and Support of patterns

Rank	Sup	Patterns
1	7	[c]
2	6	[a], [e], [b], [b, c]
3	5	[a, c], [e, c]
4	4	[b, a], [b, a, c], [b, e], [b, e, c]
5	3	[a, e], [a, e, c]
6	2	[d], [d, b], [d, b, c], [d, c], [b, a, e], [b, a, e, c]
7	1	[d, a], [d, a, c], [d, e], [d, e, c], [d, b, a], [d, b,a,c], [d, b,e], [d,b,e,c]

2.2 The subsume indexes of frequent 1-patterns

The subsume index is proposed by Song et al. [Song, Yang and Xu (2008)] to reduce the search scope in the pattern mining process, which is defined as:

$$Subsume(X)=\{Y\in I \mid X\prec Y \wedge g(X)\subseteq g(Y) \wedge g(X)=\{T\in D \mid \forall i\in X, i\in T\}\} \quad (3)$$

The subsume index of pattern (the representative item [Song, Yang and Xu (2008)]) is an itemset, which means that if $Y \in$ Subsume(X), according to some order "$\prec$"(e.g., lexicographic order), then the Tids of X are the subset of Tids of Y. Obviously, the support of the union of X with any nonvoid subset of Subsume(X) is equal to Sup(X), conversely, if the support of the union of Y one of 1-patterns with X equals Sup(X), so $Y \in$ Subsume(X).

Example 2. In Tab. 2, g([d])=Tids{1, 3}, g([b])=Tids{1, 2, 3, 4, 5, 6}, then it is easy to find that |g([d, b])|=[d].Sup=2. From the above, [b]$\in$Subsume([d]).

2.3 Problem analysis

This paper argues that: S^t is the determinant for the top-rank-k frequent patterns. S^t of the mainstream algorithm raises dynamically with the mining process until it is finally determined, crucially, the speed of this process affects the performance of the algorithm directly. So, the key to improve the mining efficiency of top-rank-k frequent patterns is to accelerate the rise of S^t.

Proceeding from this view, this paper proposes an algorithm called GTK which focuses on the crucial point of accelerating the rise of S^t to find the final S^t as early as possible. The study proves that when the k is constant and within a reasonable range, the speed of mining high rank patterns is positively correlated with the speed of raising S^t. Therefore,

to improve mining efficiency, algorithms should be made to speed up the generation of high rank patterns. Fast generation of high rank patterns is a challenging work that almost all tr-aditional algorithms avoid. To implement the mining of top-rank-*k* frequent patterns, traditional algorithms generally adopt the level-wise-search which obtains the (t+1) patterns by joining t-patterns. This method is not efficient enough.

When *k* equals to 5, a simple example is shown in Fig. 1, in which the horizontal number is pattern's rank and the gray area refers to the top-rank-*k* frequent patterns. As shown, the level-wise-search performs 8 joins in total. By 7 joins, the final S^t is determined as the support of the pattern [ABC]. {[BD], [AD], [CD]} are not top-rank-*k* frequent patterns, so these three joins are useless. Not only that, in fact, most of patterns in each level are not top-rank-*k* frequent patterns, so an equal number of joins are useless. Useless joins should be avoided as much as possible since they are time-consuming and space-consuming. While observing {[BD], [AD], [CD]}, it is not difficult to realize that these three patterns are generated by [D] and other patterns. [D] has a rank of 6, which makes it infrequent. However, due to the limitation of level-wise-search, [D] hasn't been filtered out in time for the slowly rising S^t.

Step	**S^t**	**Input dataset *D*, let *k*=5**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	0	Ectract frequent 1-patterns	A	B		C		D		
2	D.Sup	Join 1-patterns to get 2-patterns			AB	AC BC		BD	AD	CD
3	ABC.Sup	Join 2-patterns to get 3-patterns					ABC			BCD
4	ABC.Sup	While t-patterns is empty, end the mining.								

Figure 1: The process of level-wise-search in top-rank-*k*

Step	**S^t**	**Input dataset *D*, let *k*=5**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	0	Ectract frequent 1-patterns	A	B		C		D		
2	D.Sup	Join rank 2 with rank 1			AB					
3	ABC.Sup	Join rank 4 with rank 1, rank 2, rank 3				AC BC	ABC			
4	ABC.Sup	While rank.Sup < S^t(D.Sup<ABC.Sup), end the mining.								

Figure 2: The process of rank-first-search in top-rank-*k*

In order to break through the constraint of level-wise-search, based on the greedy strategy, this paper proposes the rank-first-search method just introduced. Fig. 2 shows the process of rank-first-search with the same dataset and *k*. [AB] with a rank of 3 is generated first, and thereafter, the final S^t is determined by only 3 joins. As a result, high rank patterns such as {[AB], [AC], [BC]} are generated fast, moreover, benefiting from fast rising S^t, [D] has been removed from the frequent patterns and can't join with any other pattern. Consequently, there is a reduction of 3 useless joins, which may reach thousands or even

tens of thousands on the experimental dataset. In comparison, the computational cost of rank-first-search is less than 1/2 of that of level-wise-search in the example.

Thus, the argument presented above are verified.

3 RSL: Construction and initialization

3.1 FPI: definitions and properties

3.1.1 Basic definitions and properties

Definition 1 *(FPI of a 1-pattern).* Given a transaction dataset ***D*** and a pattern X, the class consisting of (X.Its, X.Bs, X.Si) is called FPI of X and is denoted as F_X. Its means items, Bs is the bit set of Tids of X and its size equals the support of X (X.Sup), Si is the set of X's subsume indexes (described in the next chapter).

Property 1. For any F_X and F_Y, if $|X.Bs \ \& \ Y.Bs|=X.Sup$, then $Y \in Subsume(X)$.

Proof. If $|X.Bs \ \& \ Y.Bs|=X.Sup$, i.e., $|X.Bs \cap Y.Bs|=|X.Bs|$, it is equivalent to $X.Tids \subseteq Y.Tids$, i.e., $g(X) \subseteq g(Y)$, it will be found from the definition of subsume index that $Y \in Subsume(X)$. ***Q.E.D***.

Definition 2 *(FPI of a t-pattern).* Assume that pattern X, Y, both with length not greater than t($t \geq 2$), F_X=(X.Its, X.Bs, X.Si), F_Y=(Y.Its, Y.Bs, Y.Si), and the join operation of X and Y to generate A is called **Join**, F_A=**Join** (F_X, F_Y)=(A.Its, A.Bs, A.Si), it satisfies:

- $A.Its=X.Its \cup Y.Its$
- $A.Bs=X.Bs \ \& \ Y.Bs$
- $A.Si=X.Si \cup Y.Si$

Example 3. The FPIs of 1-patterns are shown in Tab. 3, $F_{[a]}$=([a], <1,1,0,1,0,1,1,1,0>, ∅), $F_{[b]}$=([b], <1,1,1,1,1,1,0,0,0>, {c}). According to definition 2, it can be found that $F_{[ab]}$=**Join**($F_{[a]}$, $F_{[b]}$)=([ab], <1,1,0,1,0,1,0,0,0>, {c}).

Table 3: The FPIs of 1-patterns

Sup	Its	Bs	Si
7	c	<1,1,1,1,1,1,1,0,0>	∅
6	e	<0,0,1,1,1,1,1,0,1>	∅
6	a	<1,1,0,1,0,1,1,1,0>	∅
6	b	<1,1,1,1,1,1,0,0,0>	{c}
2	d	<1,0,1,0,0,0,0,0,0>	{c, b}

Inference 1. Let Max(A, B) be the highest possible rank of AB generated by Join(A, B), if X, Y, Z are the three patterns of dataset ***D***, and $Rank_X \leq Rank_Y \leq Rank_Z$, then Max($X$,$Y$) ≤ Max($Y$, Z).

Proof. As $Rank_X \leq Rank_Y \leq Rank_Z$, i.e., $X.Sup \geq Y.Sup \geq Z.Sup$. From the ***Definition 1*** we have that $X.Sup=|X.Bs|$, $Y.Sup=|Y.Bs|$ and $Z.Sup=|Z.Bs|$. According to definition 2, $XY.Bs=X.Bs \ \& \ Y.Bs$, so $XY.Bs \subseteq X.Bs$, $XY.Bs \subseteq Y.Bs$. It is easy to have that $|XY.Bs| \leq |X.Bs|$ and $|XY.Bs| \leq |Y.Bs|$. Because $|X.Bs|=X.Sup \geq Y.Sup=|Y.Bs|$, so $|XY.Bs| \leq |Y.Bs|=Y.Sup$, i.e., Max($X$,$Y$)=$Rank_Y$. Similarly, Max($Y$, Z)=$Rank_Z$. Since $Rank_Y \leq Rank_Z$, so Max(X,

Y)≤Max(Y, Z). ***Q.E.D***.

Example 4. In combination with Tab. 2, $Rank_{[c]}<Rank_{[a]}\leq Rank_{[b]}$, $F_{[ab]}$=([ab], <1,1,0,1,0,1,0,0,0>, {c}), [ab].Sup=4, $Rank_{[ab]}$=4. Similarly, $F_{[ac]}$=**Join**($F_{[a]}$, $F_{[c]}$)=([ac], <1,1,0,1,0,1,1,0,0>, ∅), [ac].Sup=5, $Rank_{[ac]}$=3, we have that Max([a], [c])≤Max([a], [b]). Conclusion can be drawn from the *Inference 1*: Compared with the **Join** of any pattern and a low rank pattern, the **Join** of that pattern and a high rank pattern has a better chance of generating a high rank pattern. This conclusion provides a more powerful theoretical support for the analysis in Section 2. It is also the most direct basis for adopting rank-first-search that always joins the two patterns with highest rank to promote the early generation of high rank patterns.

3.1.2 The generation method of subsume indexes

Different from other patterns where the join operation must be performed, those patterns, including a representative item that has the same support as a representative item, can be generated directly by connecting the representative item with all the subsets of subsume index. To find the subsume indexes, the method of BTK is to check the definition of subsume index in Section 2.2 from the opposite direction when traversing 1-pattern list. However, when there are few subsume indexes of the dataset, this method has little contribution to the mining progress. Meanwhile, it will repeat the traversal of the 1-pattern list when calling the *Candidate_gen* method, the cost for repeat is unnecessary and with side effects on mining. So, this method does not make full use of computing resources and it is too time-consuming. In this paper, an optimization method of subsume index generation method called *Gen_Subsume* is given and mainly reflected in two aspects:

(1) Embed the generation process of subsume index into the process of candidate generation instead of before the process of candidate generation to be more time-saving. In this paper, when mining 2-patterns in the process of candidate generation, a 2-pattern X will be generated by joining two 1-patterns. According to *Property 1*, if the Bs size of F_X is equal to the support of one of these two patterns, then another pattern is recorded as subsume index. Therefore, there is no need to scan 1-patterns list repeatedly. The advantage of *Gen_Subsume* is that while scanning 1-patterns list, not only the subsume indexes are found, but also all frequent 2-patterns are obtained.

(2) Use the Bs of FPI for bitwise and operation instead of the B-info-code of B-list for comparison for high efficiency. *Gen_Subsume* makes use of bitsets to reduce the amount of pattern information storage space needed and take advantage of bit-level parallelism in hardware to increase performance.

After *Gen_Subsume*, the union of 1-patterns and 2-patterns including subsume indexes will be obtained and used to prepare for RSL initialization. Details of *Gen_Subsume* are presented in Fig. 3. Initially, Si of each FPI is initialized as the empty set, the S^t is set to zero. C_1(1-patterns) is obtained and arranged in descending order of support. If the length of C_1 is longer than k, S^t is updated, then it produces a frequent 1-patterns set L_1 and a set of Sup L_{sup}. Then the pattern in a range from the second one to the last one of L_1, joins with each pattern in front of it one by one, all 2-patterns generated by this pattern are stored in L_{tmp}. Once the subsume indexes of the pattern is found, the Si of each pattern in L_{tmp} should be updated. During this process, S^t has been rising.

Algorithm Gen_Subsume (***D***, *k*)
Input: ***D***, *k*
Output: L_U

```
1  Let C2=∅,Lsup=∅;
2  Get C1 consisting of FPIs of all 1-patterns, and sort C1 in descending of Sup;
3  if(|C1|≥k) then
4   St←C1[k-1].Sup;
5  end if
6  L1←{p | p∈C1∧p.Sup≥St };
7  Lsup←{p.Sup | p∈L1};
8  for (i<-1 until |L1|) do
9   if (L1[i].Sup≥St) then
10   Ltmp←∅
11   for (j<-0 until i) do
12    q.Bs←L1[i].Bs&L1[j].Bs;
13    q.Sup←|q.Bs|;
14    if(q.Sup≥St) then
15     q. Its←L1[i]. Its∪L1[j]. Its;
16     if(L1[i].Sup !=q.Sup) then
17      Ltmp←Ltmp+q;
18     else
19      L1[i].Si←L1[i].Si+L1[j]. Its;
20      p.Si←(p.Si+L1[j].Its)|∀p∈Ltmp∧Ltmp!=∅
21     end if
22     if(q.Sup∉Lsup) then
23      Lsup←Lsup+q.Sup;
24     end if
25     if (|Lsup|>k) then
26      Lsup←Lsup-St;
27      St←Lsup.min;
28     end if
29    end if
30   end for
31  end if
32  C2←C2+Ltmp;
33 end for
34 LU←{q|q∈(C2∪L1)∧q.Sup≥St}
35 Group FPIs in LU by Sup and arrange them in descending order according to Sup;
36 return LU
37 end Gen_Subsume
```

Figure 3: Gen_subsume algorithm

Example 5. Taking the dataset $\boldsymbol{D}_1$, k=5 as an example. As shown in Tab. 3, firstly, $F_{[e]}$ joins with $F_{[c]}$, [ec].Sup=5, because it is greater than the initial S^t and not equal to [e].Sup, [ec] is added to L_{tmp}. Since L_{sup} does not contain 5, 5 is added to L_{sup}. For the smaller number of elements in L_{sup} compared with k, S^t is unchanged. After the Join of [e], L_{tmp} is added to C_2. Repeat the process above according to the order from [e] to [d] while the support of the pattern is greater than S^t. When $F_{[b]}$ joins with $F_{[c]}$, the support of [bc] equals that of [b], by *Property 1*, it can be found that [c]∈Subsume ([b]). L_U is shown in Tab. 4.

3.2 Structure of RSL

All kinds of algorithms based on PPC-tree or TB-tree structure applied a table structure called Tab_k for storing top-rank-k frequent patterns. Tab_k has a fixed number of entries, usually k, and each entry contains all patterns with the same rank. During the mining proce-

Table 4: The FPIs of L_U

Rank	Sup	Its	Bs	Si
1	7	c	<1,1,1,1,1,1,1,0,0>	⌀
2	6	e	<0,0,1,1,1,1,1,0,1>	⌀
2	6	a	<1,1,0,1,0,1,1,1,0>	⌀
2	6	b	<1,1,1,1,1,1,0,0,0>	c
3	5	ec	<0,0,1,1,1,1,1,0,0>	⌀
3	5	ac	<1,1,0,1,0,1,1,0,0>	⌀
4	4	be	<0,0,1,1,1,1,0,0,0>	c
4	4	ba	<1,1,0,1,0,1,0,0,0>	c
5	3	ae	<0,0,0,1,0,1,1,0,0>	⌀

ss, Tab_k involves abundant curd operations. How to reduce the operation complexity of Tab_k is a problem worth considering. In this regard, a static doubly-linked list structure named RSL (Static Doubly-Linked Lists of top-rank-k) is used in this paper.

Definition 3 (Rnode). Rnode, the node of RSL, is composed of two parts, the cursor domain and the data domain. The cursor domain contains a link back to the previous node (prev) and a link to the next node (next). The data domain including a support (Sup) and an FS, an FPI set of the patterns with support equal to Sup. The structure of Rnode is shown as Fig. 4.

Definition 4 (RSL). RSL, a static doubly-linked list structure described by an array with a fixed length of k+1, is made up of a head node and k Rnodes. The head node only points to the highest Rnode of rank without storing any information of patterns. All Rnodes of RSL are arranged in a descending order of rank.

RSL boasts the advantages of both sequential storage structure and linked storage structure. During the insertion and deletion operations, it only needs to modify the cursor of Rnode without any element movement required. So, the consumption of insert and delete operations of Tab_k is improved.

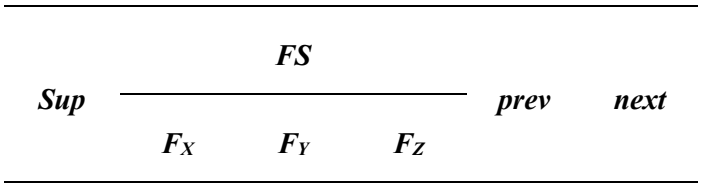

Figure 4: The structure of Rnode

3.3 RSL initialization

The construction of RSL is implemented by the *Append* function and the *Sort* function. The initialization process of RSL is shown in Fig. 5, where S is used to store the Sup of each Rnode and L is used to record the number of nonempty Rnodes of the RSL. When k=5, take the L_U shown in Tab. 4 as an example, the RSL after initialization is shown in Tab. 5.

```
Algorithm Initialization (RSL, L_U)
Input: RSL, L_U
Output: L_U
1  Let head=0; rear=0;
2  Let RSL [head]. prev=RSL[head].next=-1;
3  Let S=∅; L=0;
4  for (e<-L_U) do
5   Append (e,head);
6  end for
7  return RSL
8  end Initialization
1  function Append (e,b)
2   if(e.Sup∈S) then
3    for(Rnode<-RSL) do
4     if(Sup==Rnode.sup) then
5      Rnode.FS←Rnode.FS+e.FS;
6     end if
7     Break ()
8    end for
9   else
10   Rnode.Sup←e.Sup; Rnode.FS←e.FS;
11   if (L<k) then
12    L←L+1; RSL[L]←Rnode;
13    Sort(L,b); S←S+Sup;
14   else
15    S←S-S^t; r←rear;
16    RSL[RSL[rear].prev].next=-1;
17    rear←RSL[rear].prev;
18    RSL[r]←Rnode;
19    S^t←RSL[r].Sup ; S←S+Sup;
20   end if
21  end if
22 end Append
1  function Sort (l,b)
2   q←b; p←RSL[b].next;
3   while(p!=-1∧RSL[P].Sup>RSL[l].Sup)do
4    q←p;p← RSL[p].next
5   end while
6   RSL[l].next←p
7   RSL[l].prev←q
8   RSL[q].next←l
9   if (p==-1) then
10   rear=l
11  else
12   RSL[P]. prev=l
13  end if
14 end Sort
```

Figure 5: Initialization algorithm

Table 5: RSL after Initializing

Subscript	Sup	FS	prev	next
0 (head)			-1	1
1	7	F_c	0	2
2	6	F_e, F_a, F_b	1	3
3	5	F_{ec}, F_{ac}	2	4
4	4	F_{be}, F_{ba}	3	5
5	3	F_{ae}	4	6
6 (rear)	2	F_d, F_{be}, F_{ba}	5	-1

4 Design and implementation of GTK algorithm

Theorem 1. Given a dataset ***D***, a rank threshold *k* and the itemset I of ***D***, the top-rank-*k* frequent patterns of ***D*** are made up of at least *N* different items, and it satisfies:

$$N=\{\lceil \log_2(k+1) \rceil \mid N \leq |I|\} \quad (4)$$

Proof. When none of the top-rank-k frequent patterns belong to the same rank as the rest of the patterns, there will be only k top-rank-k frequent patterns, that is, only one pattern per rank. At this time, the number of top-rank-k frequent patterns is the smallest, thus the items constituting the top-rank-k frequent pattern set are the least. In the most ideal case, the top-rank-k frequent patterns with threshold k are composed of N items with the highest support, and the supports of those patterns are not the same. At this time, $\sum_{i=1}^{N} C_N^i > k$, i.e., $2^N-1>k$. Thus $N=\{\lceil \log_2(k+1) \rceil \mid N \leq |I|\}$. ***Q.E.D***.

It can be seen from the above that in the most ideal case, only the items N are required to obtain the top-rank-k frequent patterns, and the maximum length of the top-rank-k frequent patterns equals N. However, in normal times, more than N items are used to make up top-rank-k frequent patterns, so the maximum length of top-rank-k frequent pattern will not be greater than N. According to *Theorem 1*, this paper divides the patterns of different length into short patterns and long patterns, and defines that:

Definition 4. Assume that there is a pattern of length l and an η, an integer length threshold of pattern. Let $\eta=\lfloor \log_2(k+1)/2 \rfloor$ (η is at least 3), if $l \leq \eta$, then p is called short pattern, otherwise p is a long pattern.

In the previous problem analysis, this paper has explained the defect of level-wise-search by examples. In this regard, this chapter focuses on the fast generation of high rank patterns and proposes a mixed search method in terms of short patterns and long patterns of top-rank-k frequent patterns. In what follows, Section 4.1 describes how the *Ex_Short* method is used to mine short top-rank-k frequent patterns by adopting rank-first-search; Details of the GTK algorithm including the long patterns mining method Ex_Long are introduced in Section 4.2; For a better understanding of GTK, an example is shown in the Section 4.3.

4.1 Ex_Short method

The Ex_Short method ignores the limit of pattern length and takes advantage of rank-first-search to prompt the fast generation of high rank patterns. In order to accelerate the Ex_Short process, this paper also designed a PBJ strategy and CC strategy.

4.1.1 PBJ strategy (pruning before joining)

The main role of the PBJ strategy is to define the basic conditions of the FPI in the Join operation. Let the FPI of pattern X be F_X, the FPI of pattern Y be F_Y, and Y.Sup>X.Sup. Firstly, since the 2-patterns has been obtained by looking for the subsume indexes of 1-patterns, the 2-patterns is no longer mined in the Ex_Short process, so $|F_X.Its|$ and $|F_Y.Its|$ cannot be equal to 1 at the same time. Secondly, the new pattern's length cannot be greater than η, nor X and Y, the subsets of new patterns, $|F_X.Its|$ and $|F_Y.Its|$ should be less than η. Finally, because of the subsume index, to avoid useless Join operations, X and Y should satisfy the following:

- F_Y.Its is not a subset of F_X.Its.
- F_Y.Its and F_X.Si have no intersection.
- F_X.Its and F_Y.Si have no intersection.

4.1.2 CC strategy (check candidates)

In Tab. 3, all nonvoid subsets of the pattern [bae] can be exemplified as {[e], [a], [b], [ea], [be], [ba]}, and under the premise of meeting PBJ conditions, it is not difficult to have $F_{[bae]}$=Join($F_{[ea]}$, $F_{[b]}$)=Join($F_{[ba]}$, $F_{[e]}$)=Join($F_{[be]}$, $F_{[a]}$). Thus, when performing Join, it is necessary to check in time whether the new pattern has already existed to avoid double counting. To this end, the candidate pattern set C_i is set up to store all patterns with equal lengths, where i>2 and i is the length of pattern. Therefore, the significance of the CC strategy is: When pattern X is found, it should be filtered out immediately if it can be found in $C_{|X|}$.

4.1.3 Ex_Short method. (extracting short patterns)

The GTK algorithm produces the short top-rank-k frequent patterns through the Ex_Short method shown in Fig. 6. This procedure loops over each Rnode of which the support is

Algorithm Ex_Short(RSL,η)
Input: RSL, η
Output: RSL

```
1  p←RSL[RSL[head].next].next;
2  while(RSL[p].Sup>S^t)do
3   q←RSL[head].next;
4   while(q!=p)do
5    for(i<- RSL[p].FPIs;j<- RSL[q].FPIs)do
6    if(j.Its⊄i.Its∧(|i.Its|!=1 || |j.Its|!=1)∧
     |i.Its|<η∧|j.Its|<η∧(i.Its∩j.Si==∅)∧
     (j.Its∩i.Si==∅))then   //PBJ Strategy
7     Join (i,j, η,p)
8    end if
9    if(q==RSL[p].prev∧
     |RSL[p].FPIs|>1)then
10    f←RSL[p].FPIs
11    for(i<-1 until |f|;j<-0 until i)do
12    If((|f[i].Its|!=1 || |f[j].Its|!=1)∧
     |f[i].Its|<η∧|f[j].Its|<η∧
     (f[i].Its∩f[j].Si== ∅)∧(f[j].Its∩f[i].Si==
     ∅)) then
13     Join(f[j], f[i], η, p)
14    end if
15   end for
16  end if
17 end for
18  q=RSL[q].next
19 end while
20 p=RSL[p].next
21 end while
22 return RSL
23 end Ex_Short

1  function Join(F_X,F_Y, η, b)
2  F_Z.Its←F_Y.Its ∪F_X.Its
3  if(|F_Z.Its|≤η ∧ F_Z.Its ∉ C_|FZ.Its|)then   //CC
   Strategy
4   F_Z.Bs←F_Y.Bs &F_X.Bs
5   if(|F_Z.Bs |≥S^t) then
   F_Z.Si←F_Y. Si∪F_X.Si
6    Append ( (|F_Z.Bs|, F_Z), b )
7    C_|FZ.Its||←C_|FZ.Its|+ F_Z.Its
8   end if
9  end if
10 end Join
```

Figure 6: Ex_Short algorithm

greater than S^t, and checks them with each Rnode in front of them. When the two Rnodes do not coincide (line 4), join each FPI in FS of the two Rnodes one by one (line 5). If

PBJstrategy is met (line 6), Join function will be conducted (lines 7), and then the candidate will be checked by CC strategy, and if it has a greater support than S^t, it will be appended into RSL and the candidate pattern set (lines 1-10). When two Rnodes are continuous (lines 9), join each FPI in the FS of the latter Rnode as the above procedure (lines 10-16). So far, all short top-rank-*k* frequent patterns with length less than η are stored in the RSL in descending order. In the meantime, the St has been raised rapidly.

Algorithm GTK(***D***, *k*)
Input: ***D***, *k*
Output: $S_{top\text{-}k}$

1	Let S^t=0; η=⌊ $\log_2(k+1)$ /2⌋; RSL=∅;	9	**end while**
2	Call the **Gen_Subsume** (***D***, *k*) function to get L_U;	10	**for** (*j*<-0 until *i*)**do**
3	RSL←**Initialization** (L_U);	11	m←L_i[*j*].Its; n←L_i[*i*].Its
4	RSL←**Ex_Short**(RSL,η);	12	**if**((m[0] until m[|m|-1] == n[0] until n[|n|-1]) ∧ (m[|m|-1]∩L_i[i].Subsume== ∅) ∧ (n.[|n|-1]∩L_i[j].Subsume== ∅)) **then**
5	Extract all frequent patterns with length equal to η from RSL to form L_η, sort each pattern's Its in descending order of item's support, sort L_η in descending order of pattern's support;	13	F_q.Bs←L_i[j].Bs&L_i[i].Bs;
6	RSL←Ex_Long(RSL, L_η);	14	**if**(|F_q.Bs |≥S^t) **then**
7	**for** (p<-RSL) **do**	15	**if**(m[|m|-1].Sup>n[|n|-1].Sup)**then**
8	Put p into $S_{top\text{-}k}$	16	F_q.Its←m+n[|n|-1];
9	**if** (F_p.Si!=∅)**then**	17	F_q.Si←L_i[*j*].Si & L_i[*i*].Si;
10	Combine p with all nonvoid subsets of F_p.Si and put them into $S_{top\text{-}k}$.	18	**else**
11	**end if**	19	F_q.Its←n+m[|m|-1];
12	**end for**	20	F_q.Si ←L_i[*j*].Si & L_i[*i*].Si;
13	**return** $S_{top\text{-}k}$	21	**end if**
14	**end GTK**	22	**Append** ((|F_q.Bs|, F_q), *a*);
		23	L_{i+1}←L_{i+1}+*q*;
1	**Procedure Ex_Long** (RSL, L_i)	24	**end if**
2	**while** (|L_i |>1) **do**	25	**end if**
3	Let L_{i+1}=∅;	26	**end for**
4	**for**(i<-1 until | L_i |)**do**	27	**end if**
5	**if**(L_i[*i*].Sup>S^t) **then**	28	**end for**
6	*a*←1	29	L_i←{p| p∈L_{i+1}∧p.Sup≥S^t }
7	**while**(RSL[a].Sup>L_i[*i*].Sup) **then**	30	**end while**
8	*a* ← RSL[*a*].next	31	**end Ex_Long**

Figure 7: GTK algorithm

4.2 GTK algorithm

Since the GTK algorithm ignores the pattern length and only focuses on the highest rank patterns, it has great advantages in mining short patterns in the early stage of the algorithm. However, as patterns gradually increase, the FPI that conforms to the PBJ strategy is relatively reduced. Therefore, it is more and more time-consuming to traverse the Rnode one by one to find the long patterns. In order to solve this problem, after the Ex_Short procedure, GTK algorithms use the level-wise-search for short patterns with a length η to mine the long frequent patterns.

Because of the few long top-rank-*k* frequent patterns and the low cost of matching eligible short patterns, the hybrid mining adopted by GTK algorithms is more effective than using level- wise-search or rank-first-search alone. The GTK algorithm is shown in Fig. 7.

The GTK algorithm first mines the short top-rank-*k* frequent patterns (lines 1-4), and then extracts patterns with length of η from the RSL to construct the set L_η. It arranges L_η in descending order of pattern's support (line 5). After that, Ex_Long method is called to perform the long top-rank-*k* frequent pattern mining (line 6). At last, each pattern with subsume indexes is combined with its nonvoid subsets of subsume indexes to get the whole top-rank-*k* frequent patterns (lines 7-12).

The Ex_Long is a level-wise-search method that uses a loop to explore long patterns of greater length until no candidate can be generated. While L_i is not empty, create L_{i+1}, a new set of patterns, to store the pattern with length of i+1 (lines 2-3), and then join the patterns in L_i with each pattern in front. In the process, first get the subscript in RSL of the later (lines 7-9), after that generate the candidate, insert it into RSL and update the L_{i+1} (lines 10-26). Line 27 is the beginning of the next cycle.When the number of patterns in L_i does not exceed 1, the mining process ends.

4.3 Illustration

In the top-rank-*k* frequent patterns mining of conventional datasets, as η is assumed to be not less than 3 in this paper, *k* is greater than or equal to 64. Due to the large value of *k*, the number of the examples is limited. Because of the same principle, this paper let η=2, *k*=5, and takes the initialized RSLe in Tab. 6 as an example, the RSLe after Ex_Short procedure is presented in Tab. 7, the final Stop-k is shown in Tab. 8.

Table 6: RSL_e

Subscript	Sup	FS			prev	next
0 (head)					-1	1
1	7	c	<1,1,1,1,1,1,1,0,0>	ø	0	2
2	6	e	<0,0,1,1,1,1,1,0,1>	ø	1	3
		a	<1,1,0,1,0,1,1,1,0>	ø		
		b	<1,1,1,1,1,1,0,0,0>	c		
3 (rear)	2	d	<1,0,1,0,0,0,0,0,0>	c, b	2	-1
4						
5						

Table 7: RSL_e after Ex_Short

Subscript	Sup	FS			prev	next
0 (head)					-1	1
1	7	c	<1,1,1,1,1,1,1,0,0>	ø	0	2
2	6	e	<0,0,1,1,1,1,1,0,1>	ø	1	4
		a	<1,1,0,1,0,1,1,1,0>	ø		
		b	<1,1,1,1,1,1,0,0,0>	c		
3 (rear)	4	be	<0,0,1,1,1,1,0,0,0>	c	4	5
		ba	<1,1,0,1,0,1,0,0,0>	c		
4	5	ec	<0,0,1,1,1,1,1,0,0>	ø	2	3
		ac	<1,1,0,1,0,1,1,0,0>	ø		
5	3	ae	<0,0,0,1,0,1,1,0,0>	ø	5	-1

Table 8: $S_{top\text{-}k}$

Rank	Sup	Patterns
1	7	c
2	6	e, a, b, bc
3	5	ce, ca
4	4	eb, ab, ceb, cab
5	3	ea, cea

5 Experimental results

To accurately evaluate the performance of GTK, this paper adopts three methods, with the purpose of:

- Determining the reliability of subsume indexes generation method proposed in this paper.
- Verifying the validity of the rank-first-search based on greedy strategy.
- Evaluating the comprehensiveness and efficiency of the GTK algorithm in terms of time and memory usage.

Therefore, six datasets [4] with different characteristics, namely Chess, Connect, Mushroom, Pumsb, Retail, T10I4D100K, and two synthetic datasets Test990.99KD1 and Test2K50KD1 generated by LUCS-KDD[5] data generator are selected. Tab. 9 shows the characteristics of these datasets, including the numbers of items(num_Items) and transactions(num_Trans). A laptop with the built-in Intel® Core™ 3.0 GHz CPU and 12 G memory is equipped for running the tests. All programs are implemented in SCALA on the IntelliJ IDEA2018 software of win10 operating system.

[4] Downloaded from FIMI repository http://fimi.ua.ac.be/data/.

[5] Downloaded from https://cgi.csc.liv.ac.uk/~frans/KDD/Software.

Table 9: Characteristics of the experimental datasets

Dataset	Num_Trans	Num_Items
chess	3,196	75
mushroom	8,124	119
connect	67,557	129
T10I4D100K	10,000	870
Test990.99KD1	99,822	990
Test2K50KD1	50,000	2000
pumsb	49,046	2113
retail	88,162	16,470

5.1 The time of subsume indexes generation

Tab. 10 shows the time at which the BTK and GTK algorithms get the subsume indexes of 1-patterns. It can be clearly seen that BTK needs more time to generate subsume indexes, while GTK is more time-saving. There are three reasons for this:

(1) BTK uses all 1-patterns' B-lists to search subsume indexes, but there are not so many 1-patterns that can become top-rank-*k* frequent patterns. Therefore, the large search range increases the calculation time, and the effect may be even worse in the case of sparse datasets with large-scale 1-patterns such as Retail and Test2K50KD1. This paper has a preliminary filtering of the 1-patterns, thus reducing the time consumption.

(2) The generation of subsume indexes in BTK requires frequent calls to the *checkSubsume* function. This is a time-consuming process for B-lists that contain a large amount of B-info-code. GTK uses bitwise AND operation to reduce computational complexity and facilitate the fast generation of subsume indexes.

(3) In this paper, subsume indexes are generated through the acquisition of frequent 2-patterns. In this process, as the threshold continues to increase, some 1-patterns may not be able to be joined, so the search range of subsume indexes will become smaller and smaller. In addition, by observing the specific values of the experiment, it can be found that the GTK gets subsume indexes about 10 times faster than the BTK no matter in the face of sparse or dense datasets. Especially when dealing with pumsb, Test990.99KD1 and Test2K50KD1, the speed gap is even 100 times.

Thus, the reliability of the subsume indexes generation method proposed in this paper can be confirmed.

Table 10: Comparison of runtimes for the generation of subsume indexes of 1-patterns

Dataset	BTK (sec)	GTK (sec)	Dataset	BTK (sec)	GTK (sec)
Chess	1.0	0.1	Test990.99KD1	17.7	0.2
Mushroom	1.2	0.1	Test2K50KD1	21.2	0.3
Connect	4.2	0.2	Pumsb	13.3	0.1
T10I4D100K	7.0	1.6	Retail	7.2	0.9

Table 11: Number of candidates

Datasets	Chess		Connect	
k	BTK	GTK	BTK	GTK
100	251	252	192	188
200	755	736	452	447
300	1619	1495	746	619
400	3109	2966	1039	894
500	5330	5074	1340	1188
600	8833	8484	1656	1452
700	14282	13913	2010	1717
800	22894	22446	2385	1976
900	36100	35545	2807	2302
1000	57062	56380	3239	2661

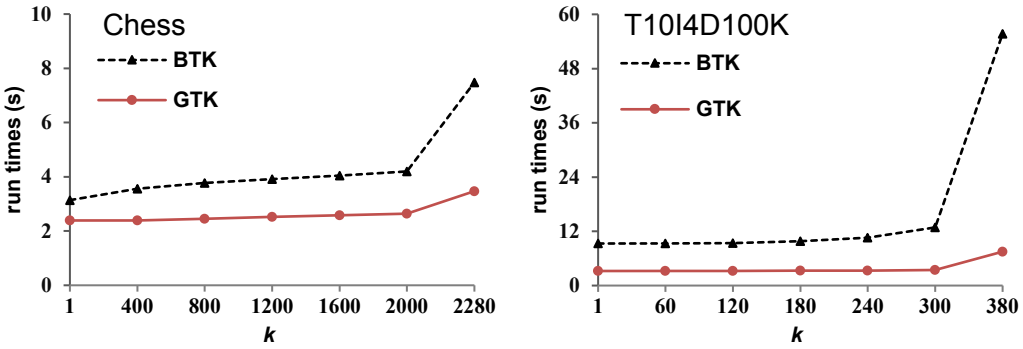

Figure 8: Comparison of the time to reach the threshold

5.2 Efficiency of the hybrid-search

The BTK algorithm is based on level-wise-search, while GTK is implemented by a way of hybrid-search. This method consists of two parts: the rank-first-search is firstly used to find short frequent patterns, and then the level-wise search is used to mine long frequent patterns. Because the high threshold can avoid numerous useless joins and reduce the amount of calculation, the purpose of using hybrid search is to speed up the support threshold rise so that final threshold could be quickly determined.

Tab. 11 shows the number of candidate patterns generated by the two algorithms during the mining process. As shown in the table, the candidate pattern generated in GTK algorithms is always less than BTK algorithms, and the gap increases as *k* increases. The cardinality of the pattern to be generated is positively related to the value of *k*. As the rank-first-search always gives priority to the generation of high rank patterns, in unit time, the rank-first-search has a greater chance to get more high rank patterns. It is a very good

method to promote a fast rise in the threshold that the rapidly rising threshold can greatly reduce the amount of search computation. In contrast, since the level-wise search is limited by length requirement of the pattern, there is a delay in the generation of high rank patterns. Moreover, it is quite time-consuming to search a lot of patterns with level-wise-search, especially the search of 2 and 3-patterns. Therefore, this is one aspect of the evidence that hybrid-search is more effective than level-wise-search.

To highlight the advantages of hybrid-search, the dynamic rise of support threshold of these two search ways is recorded in Fig. 8, where the x-axis represents the threshold and the y-axis represents the time when the corresponding threshold is reached. It is not difficult to observe that compared with level-wise-search, the support threshold is improved faster by hybrid-search, and the larger the threshold is, the larger the gap is. In general, the hybrid-search proposed in this paper outgoes the level-wise-search.

5.3 Mining performance

This paper evaluates the time efficiency and space efficiency of the GTK algorithm by testing time and memory usage on all eight datasets for various values of *k*.

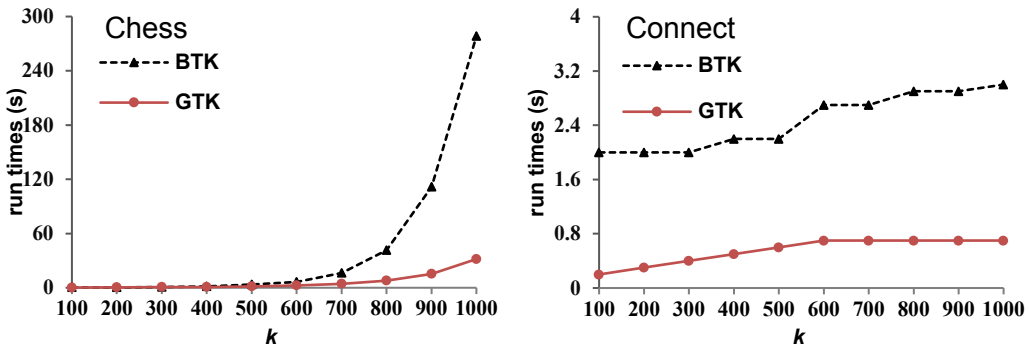

Figure 9: Comparison of mining times for Chess and Connect dataset

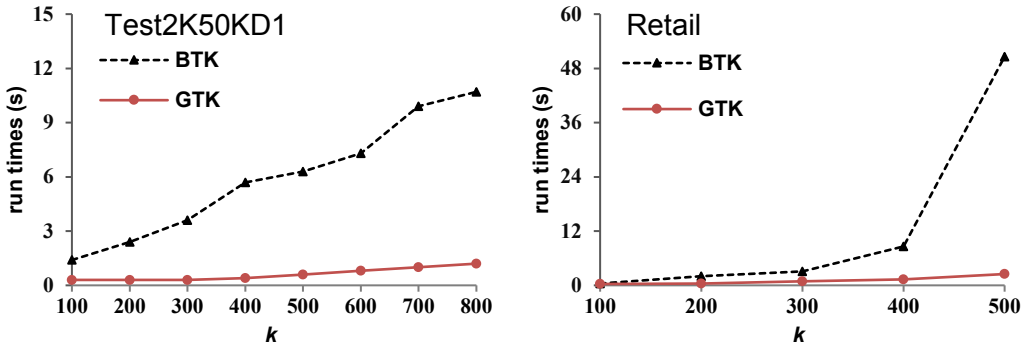

Figure 10: Comparison of mining times for Test2K50KD1 and Retail dataset

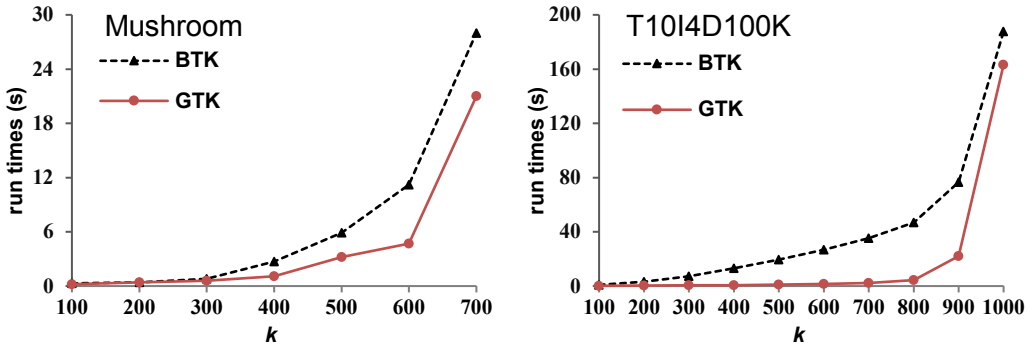

Figure 11: Comparison of mining times for Mushroom and T10I4D100K dataset

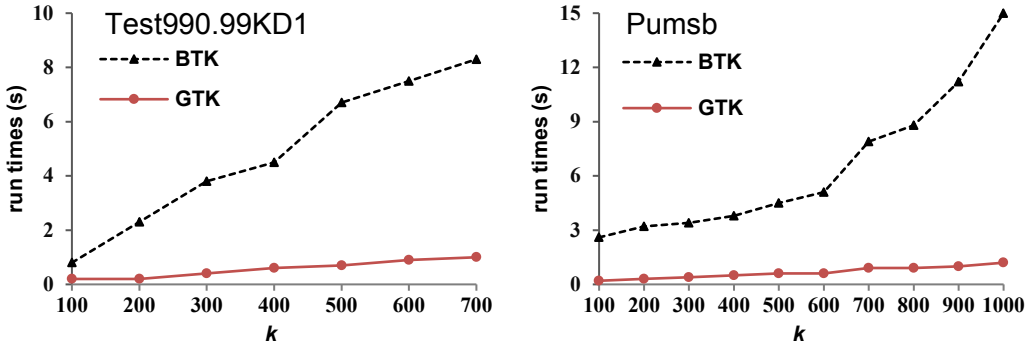

Figure 12: Comparison of mining times for Test990.99KD1 and Pumsb dataset

5.3.1 Mining time

Figs. 9, 10, 11 and 12 compare the mining time of GTK and BTK on 8 datasets. It should be noted that the timing starting points of the two algorithms are different. BTK starts from calling *Candidate_gen* function, while GTK starts with getting the 1-pattern subsume indexes. The experimental results show that GTK is more time efficient than BTK when facing different datasets and *k*, and the mining time of GTK is less affected by the increase of *k*, while BTK is the opposite.

There are many factors contributing to the efficiency of GTK, including the following: The main reason is no other than the fast rank-first-search, and its advantage is most obvious when *k* is less than 500. This is because when *k* does not exceed 500, there are not so many patterns to be mined, and most patterns are short patterns with a length less than 4 that can be mined quickly. What is more, using RSL to store frequent patterns saves a lot of time. It can be seen from the Fig. 11, when *k* is larger, more candidate patterns will be generated. Storing these patterns involves abundant operations, which will cause huge time consumption for BTK. However, RSL of GTK only need to modify the cursor when performing insert and sort operations. Thus, the time consumption is much smaller than GTK. Furthermore, GTK's efficiency also benefits from subsume

indexes, PBJ and CC strategies, which avoid a lot of useless joins. To sum up, GTK is not as sensitive to sparseand dense datasets as BTK, and GTK is about 6 times faster than BTK. When facing retail, the time gap between the two algorithms is the largest that GTK's time consumption is only 1/20 of that of BTK.

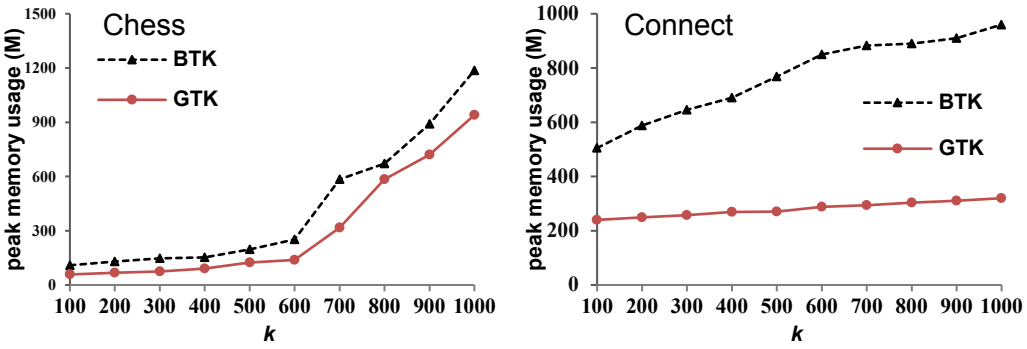

Figure 13: Comparison of memory usage for Chess and Connect dataset

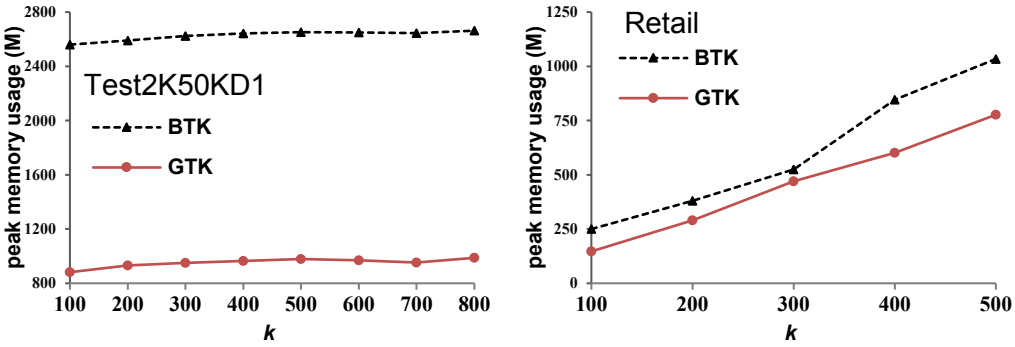

Figure 14: Comparison of memory usage for Test2K50KD1 and Retail dataset

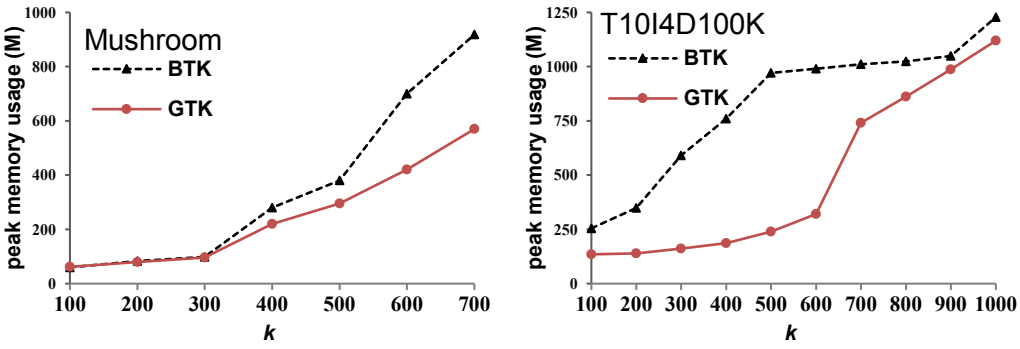

Figure 15: Comparison of memory usage for Mushroom and T10I4D100K dataset

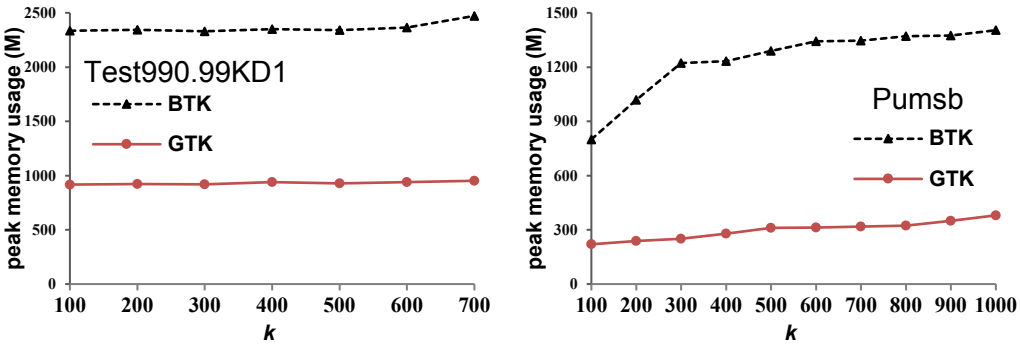

Figure 16: Comparison of memory usage for Test990.99KD1 and Pumsb dataset

5.3.2 Peak memory usage

Figs. 13, 14, 15 and 16 are comparisons of the memory consumption of BTK and GTK. Since the experimental programs are written in the JVM-based SCALA language, the memory usage here refers to the peak usage memory of the JVM during program running. JVM peak memory is dynamic and variable. In this paper, the average value of several experiments is taken to reflect the size of the actual memory occupied by the algorithm. As you can see from the figures, for the same dataset, GTK takes up less memory when the k of both is the same. There are three main reasons: (1) Compared with BTK, in which B-list is used to store the start-build and finish-build information of the patterns, GTK uses bitset to represent the pattern's transaction sets for space saving. Especially in the face of large datasets, the B-lists with a large amount of B-info-code lead to a lot of pressure on memory. (2) As shown in Fig. 8, the threshold rises rapidly by rank-first-search, thus the number of candidate patterns has been greatly reduced. Rank-first-search is also a very space efficient method. (3) The GTK algorithm only needs to maintain the array RSL and C_i used to store patterns of the same length during the mining process, so the space cost is low. The fixed length of RSL is k+1, and when it is full of k Rnodes, each new Rnode will only overwrite the tail Rnode Instead of reopening memory space. In addition, Ci only needs to store short frequent patterns other than 1-patterns and 2-patterns, which usually could not cause huge memory consumption. It is worth noting that BTK occupies up to 2.5 G of memory in the face of Test990.99KD1 and Test2K50KD1 (Figs. 14 and 16). It is clear that BTK compresses large datasets onto a unique TB-Tree, which is very large and memory-consuming, Therefore, it is susceptible to memory performance. However, GTK adopts a vertical data layout, it will not be easily affected by such problems.

6 Conclusion and future work

A hybrid mining algorithm of top-rank-*k* frequent pattern called GTK is proposed in this paper. GTK uses a static doubly linked list and a mining method of hybrid-search based on greedy strategy to find frequent patterns. To speed up the process, an optimized subsume indexes generation method and several useful pruning strategies are also

designed. Experimental results show that the proposed algorithm has a better space-time efficiency. In addition, during the experiment, it was found that the mining efficiency can further be improved by appropriately reducing the value of η when faced with some sparse datasets or large datasets. Concurrently, the experimental data also reflect some aspects of GTK that can be improved. For example, when dealing with dense data sets, it may be more time and space saving to compute the difference of the Tids than to compute the intersection.

In recent years, big data field is increasingly becoming more popular because of its powerful application function. Therefore, the parallel mining of top-rank-*k* frequent patterns will continue to be studied.

Acknowledgement: This research was supported in part by the Hunan Province's Strategic and Emerging Industrial Projects under Grant 2018GK4035, in part by the Hunan Province's Changsha Zhuzhou Xiangtan National Independent Innovation Demonstration Zone projects under Grant 2017XK2058, in part by the National Natural Science Foundation of China under Grant 61602171, and in part by the Scientific Research Fund of Hunan Provincial Education Department under Grant 17C0960 and 18B037.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

Agrawal, R.; Imieliński, T.; Swami, A. (1993): Mining association rules between sets of items in large databases. *ACM Sigmod Record*, vol. 22, no. 2, pp. 207-216.

Agrawal, R.; Srikant, R. (1994): Fast algorithms for mining association rules. *Proceedings of 20th International Conference Very Large Data Bases*, vol. 1215, pp. 487-499.

Burdick, D.; Calimlim, M.; Flannick, J.; Gehrke, J.; Yiu, T. (2005): MAFIA: a maximal frequent itemset algorithm. *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 11, pp. 1490-1504.

Deng, Z. H.; Fang, G. D. (2007): Mining top-rank-*k* frequent patterns. *International Conference on Machine Learning and Cybernetics*, vol. 2, pp. 851-856.

Deng, Z. H. (2014): Fast mining top-rank-*k* frequent patterns by using node-lists. *Expert Systems with Applications*, vol. 41, no. 4, pp. 1763-1768.

Dam, T. L.; Li, K., Fournier-Viger, P.; Duong, Q. H. (2016): An efficient algorithm for mining top-rank-*k* frequent patterns. *Applied Intelligence*, vol. 45, no. 1, pp. 96-111.

Deng, Z.; Wang, Z. (2010): A new fast vertical method for mining frequent patterns. *International Journal of Computational Intelligence Systems*, vol. 3, no. 6, pp. 733-744.

Deng, Z.; Wang, Z.; Jiang, J. (2012): A new algorithm for fast mining frequent itemsets using N-lists. *Science China Information Sciences*, vol. 55, no. 9, pp. 2008-2030.

Erwin, A.; Gopalan, R. P.; Achuthan, N. R. (2007): CTU-Mine: an efficient high utility itemset mining algorithm using the pattern growth approach. *7th IEEE International Conference on Computer and Information Technology*, pp. 71-76.

Fang, G.; Wu, Y.; Li, M.; Chen, J. (2015): An efficient algorithm for mining frequent closed itemsets. *Informatica*, vol. 39, no. 1.

Fang, G. D.; Deng, Z. H. (2008): VTK: vertical mining of top-rank-*k* frequent patterns. *Fifth International Conference on Fuzzy Systems and Knowledge Discovery*, vol. 2, pp. 620-624.

Guo, F.; Liu, P.; Ren, W.; Cao, N.; Zhang, C. et al. (2019): Research on the relationship between garlic and young garlic shoot based on big data. *Computers, Materials & Continua*, vol. 58, no. 2, pp. 363-378.

Han, J.; Pei, J.; Yin, Y.; Mao, R. (2004): Mining frequent patterns without candidate generation: a frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, vol. 8, no. 1, pp. 53-87.

Hu, J.; Mojsilovic, A. (2007): High-utility pattern mining: a method for discovery of high-utility item sets. *Pattern Recognition*, vol. 40, no. 11, pp. 3317-3324.

Hong, T. P.; Lin, K. Y.; Lin, C. W.; Vo, B. (2017): An incremental mining algorithm for erasable itemsets. *IEEE International Conference on Innovations in Intelligent Systems and Applications*, pp. 286-289.

Han, J.; Wang, J.; Lu, Y.; Tzvetkov, P. (2002): Mining top-*k* frequent closed patterns without minimum support. *IEEE International Conference on Data Mining*, pp. 211-218.

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*, vol. 42, no. 1, pp. 156-164.

Jia, L.; Xiang, L.; Liu, X. (2018): An improved BTK algorithm based on Cell-Like P system with active membranes. *12th International Conference on Learning and Intelligent Optimization*, pp. 36-48.

Le, T.; Vo, B. (2014): MEI: an efficient algorithm for mining erasable itemsets. *Engineering Applications of Artificial Intelligence*, vol. 27, pp. 155-166.

Nguyen, G.; Le, T.; Vo, B.; Le, B. (2015): EIFDD: an efficient approach for erasable itemset mining of very dense datasets. *Applied Intelligence*, vol. 43, no. 1, pp. 85-94.

Ogihara, Z. P.; Zaki, M. J.; Parthasarathy, S.; Ogihara, M.; Li, W. (1997): New algorithms for fast discovery of association rules. *3rd International Conference on Knowledge Discovery and Data Mining*.

Ruiz, D. R.; Alisha, S. (2019): Quantitative analysis of crime incidents in Chicago using data analytics techniques. *Computers, Materials & Continua*, vol. 59, no. 2, pp. 389-396.

Song, W.; Yang, B.; Xu, Z. (2008): Index-BitTableFI: an improved algorithm for mining frequent itemsets. *Knowledge-Based Systems*, vol. 21, no. 6, pp. 507-513.

Tsay, Y. J.; Chiang, J. Y. (2005): CBAR: an efficient method for mining association rules. *Knowledge-Based Systems*, vol. 18, no. 2-3, pp. 99-105.

Vo, B.; Coenen, F.; Le, T.; Hong, T. P. (2013): A hybrid approach for mining frequent itemsets. *IEEE International Conference on Systems, Man, and Cybernetics*, pp. 4647-4651.

Vo, B.; Le, T., Coenen, F.; Hong, T. P. (2016): Mining frequent itemsets using the N-list and subsume concepts. *International Journal of Machine Learning and Cybernetics*, vol. 7, no. 2, pp. 253-265.

Wang, J.; Han, J.; Pei, J. (2003): Closet+: searching for the best strategies for mining frequent closed itemsets. *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 236-245.

Wang, J.; Han, J.; Lu, Y.; Tzvetkov, P. (2005): TFP: an efficient algorithm for mining top-k frequent closed itemsets. *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 5, pp. 652-663.

Wang, Q.; Ren, J.; Davis, N.; Cheng, D. (2017): An algorithm for fast mining top-rank-*k* frequent patterns based on node-list data structure. *Intelligent Automation & Soft Computing*, vol. 9, pp. 1-6.

Yun, U.; Lee, G. (2016): Incremental mining of weighted maximal frequent itemsets from dynamic databases. *Expert Systems with Applications*, vol. 54, pp. 304-327.

Yun, U.; Ryang, H. (2015): Incremental high utility pattern mining with static and dynamic databases. *Applied Intelligence*, vol. 42, no. 2, pp. 323-352.

Zaki, M. J.; Gouda, K. (2003): Fast vertical mining using diffsets. *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 326-335.

Zaki, M. J.; Hsiao, C. J. (2002): CHARM: an efficient algorithm for closed itemset mining. *Proceedings of the 2002 SIAM International Conference on Data Mining*, pp. 457-473.

Zeng, X.; Pei, J.; Wang, K.; Li, J. (2009): PADS: a simple yet effective pattern-aware dynamic search method for fast maximal frequent pattern mining. *Knowledge and Information Systems*, vol. 20, no. 3, pp. 375-391.