ShrFP-Tree: An Efficient Tree Structure for Mining Share-Frequent Patterns

Chowdhury Farhan Ahmed, Syed Khairuzzaman Tanbeer, Byeong-Soo Jeong, and Young-Koo Lee

Database Lab, Department of Computer Engineering, Kyung Hee University
1 Seochun-dong, Kihung-gu, Youngin-si, Kyunggi-do, 446-701, Republic of Korea
Email: {farhan, tanbeer, jeong, yklee}@khu.ac.kr

Abstract

Share-frequent pattern mining discovers more useful and realistic knowledge from database compared to the traditional frequent pattern mining by considering the non-binary frequency values of items in transactions. Therefore, recently share-frequent pattern mining problem becomes a very important research issue in data mining and knowledge discovery. Existing algorithms of share-frequent pattern mining are based on the level-wise candidate set generation-and-test methodology. As a result, they need several database scans and generate-and-test a huge number of candidate patterns. Moreover, their numbers of database scans are dependent on the maximum length of the candidate patterns. In this paper, we propose a novel tree structure ShrFP-Tree (Share-frequent pattern tree) for share-frequent pattern mining. It exploits a pattern growth mining approach to avoid the level-wise candidate set generation-and-test problem and huge number of candidate generation. Its number of database scans is totally independent of the maximum length of the candidate patterns. It needs maximum three database scans to calculate the complete set of share-frequent patterns. Extensive performance analyses show that our approach is very efficient for share-frequent pattern mining and it outperforms the existing most efficient algorithms.

Keywords: Data mining, Knowledge discovery, Frequent pattern mining, Share-frequent pattern mining, Pattern growth mining.

1 Introduction

Frequent pattern mining (Agrawal et al., 1993; Agrawal and Srikant, 1994; Han et al., 2004, 2007) plays an important role in data mining and knowledge discovery techniques such as association rule mining, classification, clustering, time-series mining, graph mining, web mining etc. The initial solution of frequent pattern mining is the Apriori algorithm (Agrawal et al., 1993; Agrawal and Srikant, 1994) which is based on the level-wise candidate set generation-and-test methodology and needs several database scans. For the first database scan, it finds all the single-element frequent patterns and based on that result it generates the candidates for two-element frequent patterns. In the second database scan, it finds all the two-element frequent patterns and based on that result it generates the candidates for three-element frequent patterns and so on. FP-growth (Han et al., 2004) solved this problem by introducing a prefix-tree (FP-tree) based algorithm without candidate set generation-and-test. This algorithm is called the pattern growth or FP-growth algorithm and needs two database scans.

In practice, considering the binary frequency (either absent or present) or support of a pattern may not be a sufficient indicator for finding meaningful patterns from a transaction database because it only reflects the number of transactions in the database which contain that pattern. In our real world scenarios, one user can buy multiple copies of items. We can consider an example in market basket data. For example, customer X has bought 3 pens, 4 pencils and 1 eraser, customer Y has bought 10 apples, customer Z has bought 3 breads and 5 milks and customer R has bought 2 shirts and 1 shoe. Therefore, traditional frequency/support measure cannot analyse the exact number of items (itemsets) purchased. For that reason, itemset share approach (Carter et al., 1997; Barber and Hamilton, 2000, 2001, 2003; Li et al., 2005a,b) has been proposed to discover more important knowledge in association rule mining (Agrawal et al., 1993; Agrawal and Srikant, 1994; Verma and Vyas, 2005; Wei et al., 2006). Share measure can provide useful knowledge about the numerical values that are typically associated with the transactions items. In addition to our real world retail market, it is also well applicable to find more useful web path traversal patterns because time spent in each website by the user is different. Other application areas, such as biological gene database, stock tickers, network traffic measurements, web-server logs, data feeds from sensor networks and telecom call records can have similar solutions.

Motivated from these real world scenarios, we propose one efficient approach to discover share-frequent patterns. The existing solutions (Carter et al., 1997; Barber and Hamilton, 2000, 2001, 2003; Li et al., 2005a,b) suffer in the level-wise candidate set generation-and-test problem and they need several database scans depending on the length of the candidate patterns. In this paper, we propose a novel tree structure ShrFP-Tree (Share-frequent pattern tree) for share-frequent pattern mining. By holding useful property and exploiting a pattern growth technique, ShrFP-Tree finds all the actual share-frequent patterns very efficiently. Moreover, it needs maximum three database scans for finding out complete set of the share-frequent patterns, i.e. its number of database scans is not dependent on the maximum length of candidate. Extensive performance analyses show that our approach outperforms the existing most efficient algorithms ShFSM (Li et al., 2005a) and DCG (Li et al., 2005b) in both dense and sparse datasets.

In summary, the main contributions of this paper

are (1) Devising a novel tree structure that can efficiently maintain the maximum share information of each item in a tree for finding all the candidate share-frequent patterns, (2) Applying a pattern growth mining approach on the above tree structure in order to eliminate the level-wise candidate generation-and-test methodology in share-frequent pattern mining, (3) Demonstration of how to achieve the complete share-frequent patterns by using the proposed approach and by scanning the database maximum three times, and (4) Extensive performance study to compare the performance of our algorithm with the existing most efficient algorithms ShFSM (Li et al., 2005*a*) and DCG (Li et al., 2005*b*).

The remainder of this paper is organized as follows. In Section 2, we describe related work and the main problems of the existing most efficient ShFSM and DCG algorithms for mining share-frequent patterns. In Section 3, we describe the share-frequent pattern mining problem. In Section 4, we describe the construction and mining process of our proposed ShrFP-Tree structure for share-frequent pattern mining. In Section 5, experimental results are presented and analysed. Finally, conclusions are presented in Section 6.

2 Related Work

2.1 Frequent Pattern Mining

Let $I = \{i_1, i_2, ...i_m\}$ be a set of items and D be a transaction database $\{T_1, T_2, ...T_n\}$ where each transaction $T_i \in D$ is a subset of I. The support/frequency of a pattern $X\{x_1, x_2, ...x_p\}$ is the number of transactions containing the pattern in the transaction database. The problem of frequent pattern mining is to find the complete set of patterns satisfying a minimum support in the transaction database. The *downward closure* property (Agrawal et al., 1993; Agrawal and Srikant, 1994) is used to prune the infrequent patterns. This property tells that if a pattern is infrequent then all of its super patterns must be infrequent. Apriori (Agrawal et al., 1993; Agrawal and Srikant, 1994) algorithm is the initial solution of frequent pattern mining problem and very useful in association rule mining. But it suffers from the candidate generation-and-test problem and needs several database scans.

FP-growth (Han et al., 2004) solved the problem of candidate generation-and-test by using a tree-based (FP-tree) solution without any candidate generation. It needs only two database scans to find all the frequent patterns. FP-array (Grahne and Zhu, 2005) technique was proposed to reduce the FP-tree traversals and it efficiently works especially in sparse datasets. One interesting measure *h-confidence* (Xiong et al., 2006) was proposed to identify the strong support affinity frequent patterns. Some other research (Wang et al., 2005; Han et al., 2007; Dong and Han, 2007; Liu et al., 2007) has been done for frequent pattern mining. Tree structures have been proposed to calculate all the frequent patterns using a single pass of database such as CanTree (Leung et al., 2007), CP-tree (Tanbeer et al., 2008), etc. This traditional frequent pattern mining problem considers only the binary occurrence (0/1), i.e. either absent or present of the items in one transaction.

2.2 Share-Frequent Pattern Mining

Carter et al. 1997 first introduced the share-confidence model to discover useful knowledge about numerical attributes associated with items in a transaction. ZP and ZSP (Barber and Hamilton, 2000, 2003) algorithms use heuristic methods to generate all the candidate patterns. Moreover, they cannot rely on the *downward closure* property and therefore their searching method is very time-consuming and does not work efficiently in large databases. Some other algorithms such as SIP, CAC and IAB (Barber and Hamilton, 2000, 2001, 2003) have been proposed to mine share-frequent patterns but they may not discover all the share-frequent patterns. Fast share measure (ShFSM), (Li et al., 2005*a*) improves the previous algorithms by using the *level closure* property. This property cannot maintain the *downward closure* property. ShFSM wastes the computation time on the join and the prune steps of candidate generation in each pass, and generates too many useless candidates.

Direct Candidates Generation (DCG) (Li et al., 2005*b*) algorithm outperforms ShFSM by generating candidates directly without the prune and the join steps in each pass. Moreover, the number of candidates generated by DCG is less than ShFSM. DCG can maintain the *downward closure* property by using the potential maximum *local measure value* (Definition 5) of an itemset which is actually the *transaction measure value* (Definition 6) of an itemset. Still, DCG has a big problem of level-wise candidate generation-and-test methodology. As a result, its number of database scans is dependent on maximum candidate length and it tests huge unnecessary candidate patterns. In the *k-th* pass, DCG scans the whole database to count the *local measure value* of each candidate k-itemset X and counts the potential maximum share value of each monotone $(k+1)$-superset of X (Li et al., 2005*b*). Therefore, DCG generates and tests too many candidate patterns in the mining process. For example, if the number of distinct items is 10000, then it tests $\binom{10000}{2}$ two-element candidate patterns in pass-1 to get the actual candidate patterns of pass-2. Moreover, its number of database scans is dependent on the maximum length of the actual candidate patterns. For example, if the maximum length of a candidate pattern is 4 ("*abcd*"), DCG has to scan database 4 times to find all the share-frequent itemsets. If the maximum actual candidate length is 20, a total of 20 database scans are required. So, for maximum actual candidate length N, a total of N database scans are required. As a result, DCG is very inefficient for (1) dataset where the number of distinct items is large and (2) dense datasets where the maximum candidate pattern length is big.

In this paper, we propose a novel tree structure to remove these problems of the existing most efficient known algorithm DCG. Our approach generates a very few candidates using a pattern growth technique and its maximum number of database scans is three which is totally independent of the maximum length of the candidate patterns.

3 Problem Definition

We have adopted similar definitions presented in the previous works (Carter et al., 1997; Barber and Hamilton, 2000, 2001, 2003; Li et al., 2005*a*,*b*). Let $I = \{i_1, i_2, ...i_m\}$ be a set of items and D be a transaction database $\{T_1, T_2, ...T_n\}$ where each transaction $T_i \in D$ is a subset of I.

Definition 1: The *measure value* $mv(i_p, T_q)$, represents the quantity of item i_p in transaction T_q. For example, in Table 1, $mv(a, T_3) = 5$.

Definition 2: The *transaction measure value* of a transaction T_q denoted as $tmv(T_q)$ means the *total*

Table 1: Example of a transaction database with counting

TID	Transaction	Count	Total count
T_1	$\{a, b, f, g\}$	$\{2, 1, 2, 1\}$	6
T_2	$\{b, c, h\}$	$\{3, 2, 2\}$	7
T_3	$\{a, c, e\}$	$\{5, 3, 3\}$	11
T_4	$\{c\}$	$\{5\}$	5
T_5	$\{b, c, d\}$	$\{4, 3, 2\}$	9
T_6	$\{a, c, e, f, g\}$	$\{1, 3, 1, 2, 1\}$	8
T_7	$\{a, d\}$	$\{1, 3\}$	4
T_8	$\{a, c, e, f\}$	$\{4, 2, 1, 5\}$	12

measure value of a transaction T_q and it is defined by,

$$tmv(T_q) = \sum_{i_p \in T_q} mv(i_p, T_q) \quad (1)$$

For example, $tmv(T_7) = mv(a, T_7) + mv(d, T_7) = 1 + 3 = 4$ in Table1.

Definition 3: The total measure value $Tmv(DB)$ represents the *total measure value* in *DB.* It is defined by,

$$Tmv(DB) = \sum_{T_q \in DB} \sum_{i_p \in T_q} mv(i_p, T_q) \quad (2)$$

For example, $Tmv(DB) = 62$ in Table1.

Definition 4: The *itemset measure value* of an itemset X in transaction T_q, $imv(X, T_q)$ is defined by,

$$imv(X, T_q) = \sum_{i_p \in X} mv(i_p, T_q) \quad (3)$$

where $X = \{i_1, i_2, ...i_k\}$ is a k-itemset, $X \subseteq T_q$ and $1 \leq k \leq m$. For example, $imv(bc, T_2) = 3 + 2 = 5$ in Table1.

Definition 5: The *local measure value* of an itemset X is defined by,

$$lmv(X) = \sum_{T_q \in DB_X} \sum_{i_p \in X} imv(i_p, T_q) \quad (4)$$

where DB_X is the set of transactions contain itemset X. For example, $lmv(ac) = imv(ac, T_3) + imv(ac, T_6) + imv(ac, T_8) = 8 + 4 + 6 = 18$ in Table1.

Definition 6: The *transaction measure value* of an itemset X, denoted by $tmv(X)$, is the sum of the *tmv* values of all the transactions containing X.

$$tmv(X) = Tmv(DB_X) = \sum_{X \subseteq T_q \in DB_X} tmv(T_q) \quad (5)$$

For example, $tmv(g) = tmv(T_1) + tmv(T_6) = 6 + 8 = 14$ in Table 1.

Definition 7: The *share* value of an itemset X, denoted as *SH(X),* is the ratio of the *local measure value* of X to the *total measure value* in *DB.* So, *SH(X)* is defined by,

$$SH(X) = \frac{lmv(X)}{Tmv(DB)} \quad (6)$$

For example, $SH(ac) = 18/62 = 0.29$, in Table 1.

Definition 8: Given a minimum share threshold, *minShare,* an itemset is *share-frequent* if $SH(X) \geq minShare$. If *minShare* is 0.25(we can also express it as 25%), in the example database, *"ac"* is a share-frequent itemset, as $SH(ac) = 0.29$.

Definition 9: The *minimum local measure value,* min_lmv, is defined as

$$min\_lmv = ceiling(minShare \times Tmv(DB)) \quad (7)$$

In Table 1, if $minShare = 0.25$, then $min\_lmv = ceiling\ (0.25 \times 62) = ceiling\ (15.5) = 16$. So, for any itemset X, if $lmv(X) \geq min\_lmv$, then we can say that X is a share-frequent pattern.

Main challenging problem of share-frequent pattern mining area is, itemset share does not have the *downward closure* property. For example, $SH(a) = 0.2096$ in Table 1, so "a" is a share-infrequent item in Table 1 for $minShare = 0.25$, but $SH(ac) = 0.29$, so "ac" is a share-frequent itemset. As a result, the *downward closure* property does not satisfy. Therefore, maintaining the *downward closure* property is very challenging here.

The *downward closure* property can be maintained by using the *transaction measure value* (Definition 6). For a pattern X, if $tmv(X) < min\_lmv$, then we can prune that pattern without further consideration. For example, In Table 1 if we consider $minShare = 0.25$, then $tmv(g) = 14 < min\_lmv(16)$. As a result, according to the *downward closure* property none of the super patterns of "g" can be a share-frequent pattern and therefore we can easily prune "g" at the early stage.

4 ShrFP-Tree: Our Proposed Tree Structure

4.1 Construction Process of ShrFP-Tree

In this section, we describe the construction process of the ShrFP-Tree (Share-frequent pattern tree) for share-frequent pattern mining. We maintain Header table and keep item name and *tmv* values of items in it. To facilitate the tree traversals adjacent links are also maintained (not shown in the figures for simplicity) in the ShrFP-Tree.

In the first database scan, ShrFP-Tree captures the *tmv* value of all the items. Consider the database shown in Table 1 and $minShare = 0.25$. According to equation 7, $min\_lmv = 16$. After the first database scan, the *tmv* values of the individual items are $< a : 41, b : 22, c : 52, d : 13, e : 31, f : 26, g : 14$ and $h : 7 >$. To be a candidate share-frequent item, the *tmv* of an item must be at least 16. Therefore, the items "d", "g" and "h" are not candidate items. According to the *downward closure* property, we can prune these items without further consideration. Next, we sort

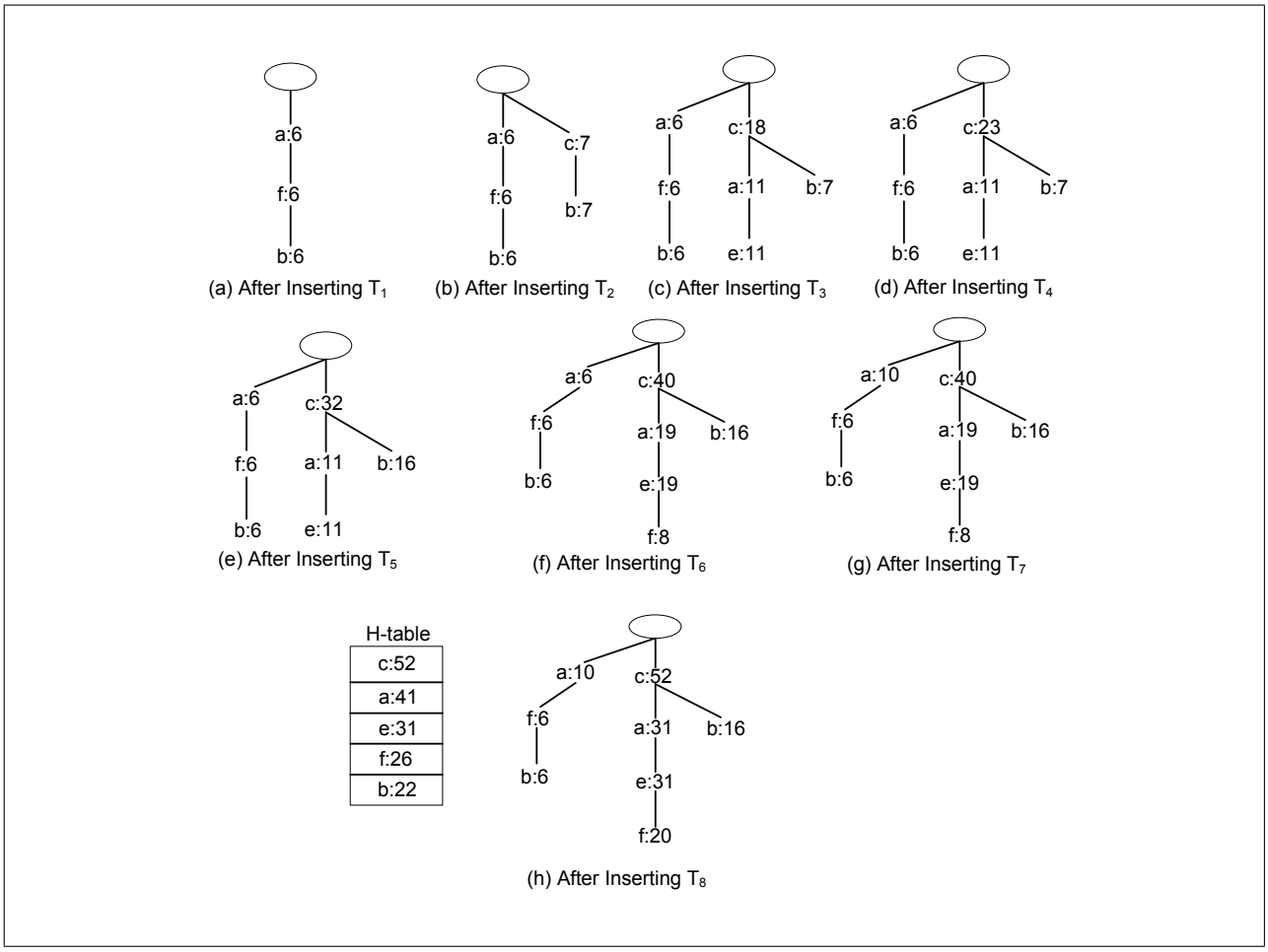

Figure 1: Construction process of ShrFP-Tree

the header table in descending order according to *tmv* values of the items. The header table order of items is $< c : 52, a : 41, e : 31, f : 26$ and $b : 22 >$.

In the second database scan we consider only candidate items from each transaction and sort them according to the header table sort order, and then we insert them into the tree. For the first transaction T_1, which contains item "a", "b", "f" and "g", we discard the non-candidate item "g" at first and then arrange the items according to the header table sort order. Items "a", "f" and "b" get the *tmv* value of T_1, which is 6. Figure 1(a) shows the ShrFP-Tree after inserting T_1. In T_2, non-candidate item "h" is removed and the remaining items of T_2 are arranged (at first "c" then "b") in the descending *tmv* order presented in the header table. After that, items "c" and "b" are inserted in the ShrFP-Tree as a new path with *tmv* value 7, shown in Figure 1(b). In T_3, all items are candidate items. They are arranged in the same way and the order is "c", "a", "e". Item "c" gets the prefix sharing with the existing node containing item "c". The *tmv* value of "c" becomes 7+11=18, item "a" becomes its child with *tmv* value 11 and item "e" becomes the child of "a" with same *tmv* value 11(shown in Figure 1(c)). Figure 1(d) to Figure 1(h) show the insertion process of transactions T_4 to T_8. Figure 1(h) shows the final tree with the header table for the full database presented in Table 1. In the next section we will perform the mining operation in this tree presented at Figure 1(h).

Property 1: The total count of *tmv* value of any node in ShrFP-Tree is greater than or equal to the sum of total counts of *tmv* values of its children.

4.2 Mining Process of ShrFP-Tree

ShrFP-Tree exploits a pattern growth mining approach to mine all the candidate share-frequent patterns. As our tree-structure has the important property of FP-tree stated in property 1, pattern growth mining algorithm can be directly applicable to it by using the *tmv* value.

Consider the database of Table 1 and *minShare* = 0.25 in that database. The final ShrFP-Tree is created for that database is shown in Figure 1(h). At first the conditional tree of the bottom most item "b" (shown in Figure 2(a)) is created by taking all the branches prefixing the item "b" and deleting the nodes containing an item which cannot be a candidate pattern with the item "b". Obviously, items "f" and "a" cannot be candidate patterns with item "b" as they have low *tmv* values with it. Both the items "f" and "a" has *tmv* value 6 with the item "b" and minimum *tmv* value must be 16 to be a candidate pattern. So, the conditional tree of item "b" does not contain the items "f" and "a". Therefore, candidate patterns (1) $\{b, c\}$ and (2) $\{b\}$ are generated here.

In the similar fashion, conditional tree for item "f" is created in Figure 2(b) and candidate patterns (3) $\{e, f\}$, (4) $\{a, f\}$, (5) $\{c, f\}$ and (6) $\{f\}$ are generated. The conditional tree of itemset "fe" is shown in Figure 2(c) and candidate patterns (7) $\{a, e, f\}$, (8) $\{c, e, f\}$ and (9) $\{a, c, e, f\}$ are generated. The conditional tree of itemset "fa" is shown in Figure 2(d) and candidate pattern (10) $\{a, c, f\}$ is generated. The conditional tree of item "e" is shown in Figure 2(e) and candidate patterns (11) $\{a, e\}$, (12) $\{c, e\}$, (13) $\{a, c, e\}$ and (14) $\{e\}$ are generated. The conditional tree of item "a" is shown in Figure 2(f) and candidate patterns (15)

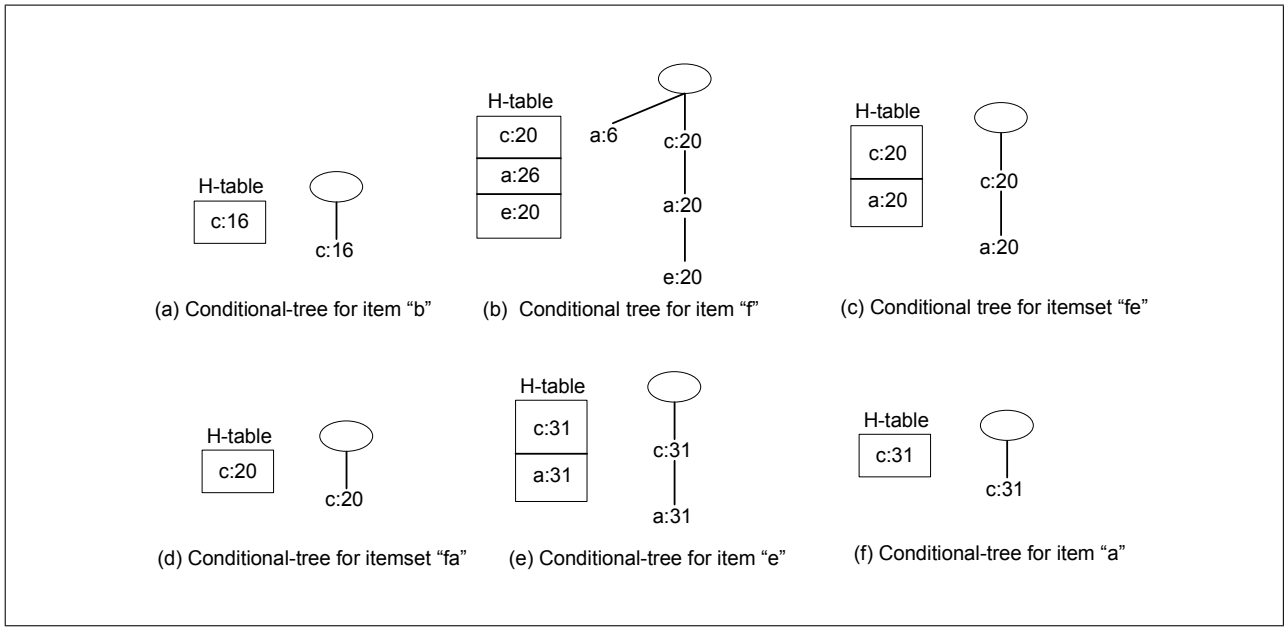

Figure 2: Mining process of ShrFP-Tree

Table 2: Calculation process of share-frequent patterns

No.	Candidate patterns	tmv	lmv	SH	Share-frequent patterns
1.	bc	16	12	0.1935	No
2.	b	22	8	0.129	No
3.	ef	20	9	0.145	No
4.	af	26	16	0.258	Yes
5.	cf	20	12	0.1935	No
6.	f	26	9	0.145	No
7.	aef	20	14	0.2258	No
8.	cef	20	14	0.2258	No
9.	$acef$	20	19	0.306	Yes
10.	acf	20	17	0.274	Yes
11.	ae	31	15	0.2419	No
12.	ce	31	13	0.2096	No
13.	ace	31	23	0.37	Yes
14.	e	31	5	0.08	No
15.	ac	31	18	0.29	Yes
16.	a	41	13	0.2096	No
17.	c	52	18	0.29	Yes

$\{a, c\}$ and (16) $\{a\}$ are generated. The last candidate pattern (17) $\{c\}$ is generated for the top-most item "c". Our approach performs the third database scan to find share-frequent patterns from these 17 candidate patterns. Table 2 shows the calculation process of the actual share-frequent patterns from the candidate patterns. The resultant share-frequent patterns are, $\{a, f\}$, $\{a, c, e, f\}$, $\{a, c, f\}$, $\{a, c, e\}$, $\{a, c\}$ and $\{c\}$.

5 Experimental Results

In this section, we present our experimental results on the performance of our proposed approach in comparison with the most efficient share-frequent pattern mining algorithms, DCG (Li et al., 2005b) and ShFSM (Li et al., 2005a). The main purpose of this experiment is to show how efficiently and effectively the share-frequent patterns can be discovered in both dense and sparse datasets by our approach compared to the existing algorithms.

To evaluate the performance of our proposed tree structure, we have performed several experiments on IBM synthetic dataset $T10I4D100K$ and real life datasets *mushroom* and *kosarak* from frequent itemset mining dataset repository (http://fimi.cs.helsinki.fi/data/) and UCI Machine Learning Repository (http://kdd.ics.uci.edu/). These datasets provides binary quantity of each item for each transaction. As like the performance evaluation of the previous share-frequent pattern mining (Li et al., 2005a,b) we have generated random numbers for the quantity of each item in each transaction, ranging from 1 to 10. Our programs were written in Microsoft Visual C++ 6.0 and run with the Windows XP operating system on a Pentium dual core 2.13 GHz CPU with 2GB main memory.

Dense datasets (Sucahyo et al., 2003) have too many long frequent as well as share-frequent patterns. The probability of an item's occurrence is very high in every transaction. As a result, for comparatively higher threshold, dense datasets have too many can-

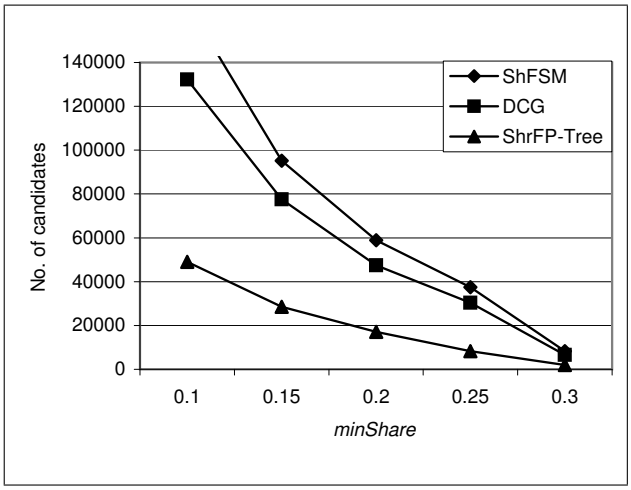

Figure 3: No. of candidates comparison on the *mushroom* dataset

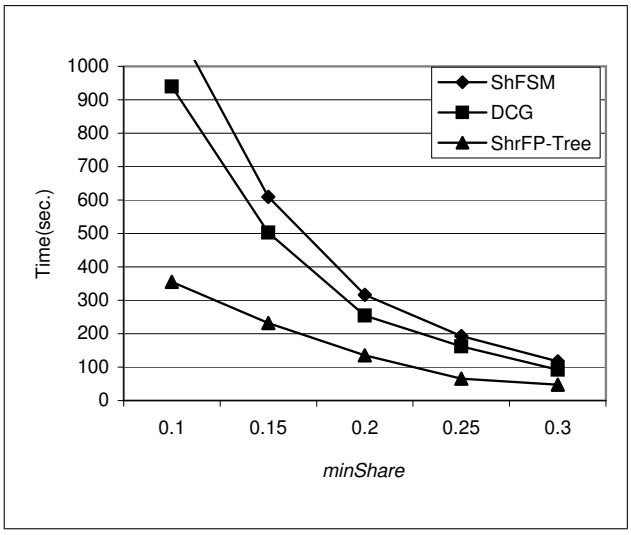

Figure 4: Execution time comparison on the *mushroom* dataset

didate patterns. Actually, long patterns need several database scans. That's why, when the dataset becomes denser, or minimum threshold becomes low, the number of candidates and total running time sharply increases in the Apriori based existing algorithms.

The *mushroom* dataset contains 8,124 transactions and 119 distinct items. Its means transactions size is 23, around 20% ((23/119)×100) of its distinct items are present in every transaction and therefore it is a dense dataset. At first we compare the number of candidate patterns that have been tested by each algorithm. Figure 3 shows the number of candidate patterns comparison. The numbers of candidates of the existing algorithms rapidly increase below *minShare* = 0.2 (i.e. 20%). For *minShare* 0.1 and 0.15, the amounts of their candidate patterns are remarkable larger from our candidate patterns.

Figure 4 shows the running time comparison in the *mushroom* dataset. In the case of existing algorithms, for lower thresholds they have too many long candidate patterns and several database scans are needed for the huge number of long candidate patterns. As a result, time difference between existing algorithms and our algorithm becomes larger when the *minShare* decreases. So, these results demonstrate that existing algorithms are very inefficient for dense dataset when the *minShare* is low.

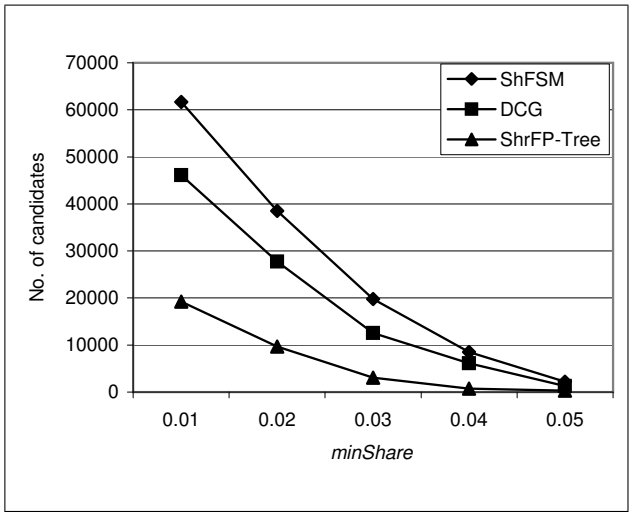

Figure 5: No. of candidates comparison on the $T10I4D100K$ dataset

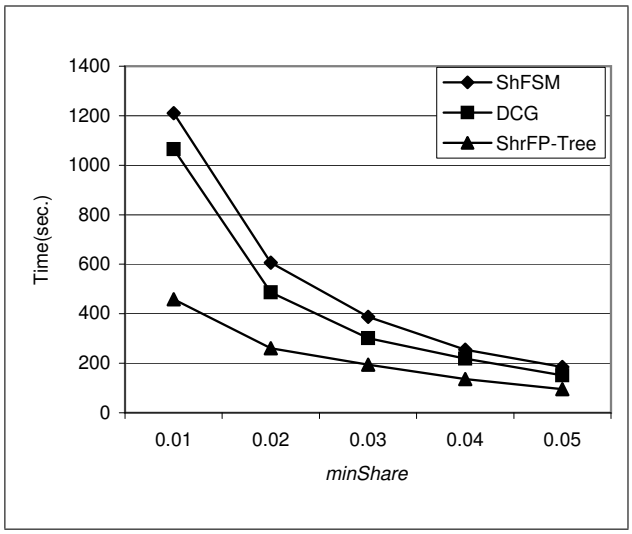

Figure 6: Execution time comparison on the $T10I4D100K$ dataset

Sparse datasets (Ye et al., 2005; Grahne and Zhu, 2005) normally have too many distinct items. Although in the average case their transactions length is small, but they normally have many transactions. As we described in Section 2.2, handling too many distinct items is a severe problem in Apriori-like existing algorithms. We show here that handling large number of distinct items and several database scans over long sparse datasets also make the existing algorithms inefficient in sparse datasets.

The $T10I4D100K$ dataset contains 100,000 transactions and 870 distinct items. Its mean transaction size is 10.1, around 1.16% ((10.1/870)×100) of its distinct items are present in every transaction and therefore it is a sparse dataset. The performance of our algorithm is better than the existing algorithms in both number of candidates and execution time comparisons, shown in Figure 5 and Figure 6 respectively. Obviously, the difference of candidate patterns and running time of existing algorithms and our approach becomes larger when the *minShare* becomes low.

The dataset *kosarak* contains click-stream data of a Hungarian on-line news portal. It contains 990,002 transactions and 41,270 distinct items. Its mean transaction size is 8.1, and it is a large sparse dataset. Around 0.0196% ((8.1/41270)×100) of its distinct items are present in every transaction. As we

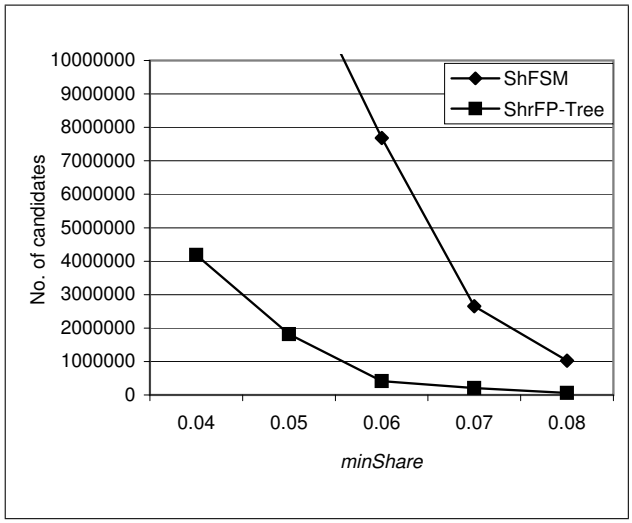

Figure 7: No. of candidates comparison on the *kosarak* dataset

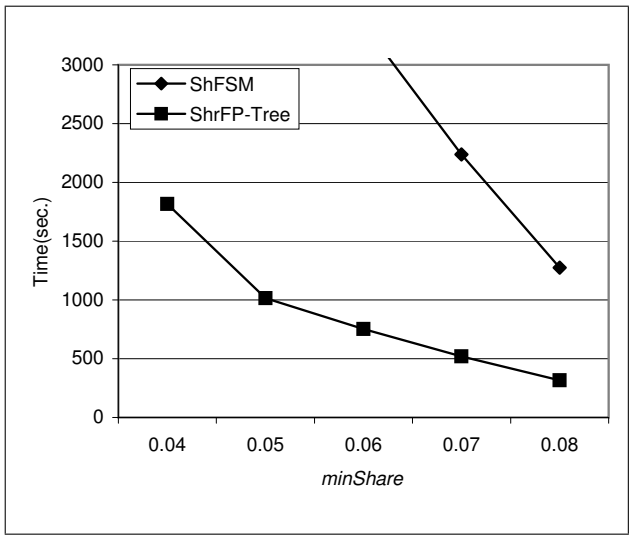

Figure 8: Execution time comparison on the *kosarak* dataset

described in Section 2.2, existing algorithms generate-and-test too many candidate patterns for this large number of distinct items shown in Figure 7. Obviously, too much time is needed for handling these candidates and scanning the long *kosarak* dataset with them. Running time comparison is shown in Figure 8. We compared the performance of our approach with the existing ShFSM algorithm. Since DCG maintains an extra array for each candidate (Li et al., 2005*b*), we could not keep all its candidates in each pass in the main memory. Figure 8 shows that our approach outperforms the existing algorithm on the *kosarak* dataset. Therefore, the existing algorithms are very inefficient for sparse datasets having too many distinct items and number of transactions.

Fig. 8 also shows that ShrFP-Tree has efficiently handled the 41,270 distinct items and around 1 million transactions in the *kosarak* dataset. Therefore, these experimental results demonstrate the scalability of our tree structure to handle a large number of distinct items and transactions.

6 Conclusions

The main contribution of this paper is to provide a very efficient research work for share-frequent pattern mining in the area of data mining and knowledge discovery. To solve the level-wise candidate set generation-and-test problem of the existing algorithms, we propose a novel tree structure ShrFP-Tree. Our technique prunes huge number of unnecessary candidates during tree creation time by eliminating non-candidate single-element patterns and also during mining time by using a pattern growth approach. Its maximum number of database scans is totally independent of the maximum length of candidate patterns. It needs maximum three database scans in contrast to several database scans needed for the existing algorithms. Moreover, ShrFP-Tree is very simple, easy to construct and handle. Extensive performance analyses show that our approach is very efficient and it outperforms the existing most efficient algorithms in both dense and sparse datasets.

Acknowledgement

This study was supported by a grant of the Korea Health 21 R&D Project, Ministry for Health Welfare and Family Affairs, Republic of Korea (A020602).

References

Agrawal, R., Imielinski, T., and Swami, A. (1993), Mining association rules between sets of items in large databases, *in* 'Proceedings of the 12th ACM SIGMOD International Conference on Management of Data', pp. 207-216.

Agrawal, R. and Srikant, R. (1994), Fast Algorithms for Mining Association Rules in Large Databases, *in* 'Proceedings of the 20th International Conference on Very Large Data Bases', pp. 487-499.

Barber, B. and Hamilton, H.J. (2000), Algorithms for mining share frequent itemsets containing infrequent subsets, in D.A. Zighed, H.J. Komorowski, J.M. Zytkow (Eds.), '4th European Conf. on Principles of Data Mining and Knowledge Discovery (PKDD 2000)', Lecture Notes in Computer Science, Vol. 1910, Springer-Verlag, Berlin, pp. 316-324.

Barber, B. and Hamilton, H.J. (2001), 'Parametric algorithm for mining share frequent itemsets', *Journal of Intelligent Information Systems*, Vol. 16, pp. 277-293.

Barber, B. and Hamilton, H.J. (2003), 'Extracting share frequent itemsets with infrequent subsets', *Data Mining and Knowledge Discovery*, Vol. 7, pp. 153-185.

Carter, C.L., Hamilton, H.J., and Cercone, N. (1997), Share based measures for itemsets, in H.J. Komorowski, J.M. Zytkow (Eds.), '1st European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD 1997)', Lecture Notes in Computer Science, Vol. 1263, Springer-Verlag, Berlin, pp. 14-24.

Dong, J. and Han, M. (2007), 'BitTableFI: An efficient mining frequent itemsets algorithm', *Knowledge-Based Systems*, Vol. 20, pp. 329-335.

Grahne, G. and Zhu, J. (2005), 'Fast Algorithms for frequent itemset mining using FP-Trees', *IEEE Transactions on Knowledge and Data Engineering*, Vol. 17, no. 10, pp. 1347-1362.

Han, J., Cheng, H., Xin, D. and Yan, X. (2007), 'Frequent pattern mining: current status and future directions', *Data Mining and Knowledge Discovery*, Vol. 15, pp. 55-86.

Han, J., Pei, J., Yin, Y., and Mao, R., (2004)'Mining frequent patterns without candidate generation: a frequent-pattern tree approach', *Data Mining and Knowledge Discovery,* Vol. 8, pp. 53-87.

Leung, C. K.-S., Khan, Q.I., Li, Z. and Hoque, T. (2007), 'CanTree: a canonical-order tree for incremental frequent-pattern mining', *Knowledge and Information Systems*, Vol. 11, no. 3, pp. 287-311.

Li, Y.-C., Yeh, J.-S. and Chang, C.-C. (2005*a*), A fast algorithm for mining share-frequent itemsets, *in* 'Proceedings of the 7th Asia-Pacific Web Conference on Web Technologies Research and Development (APWeb)', Lecture Notes in Computer Science, Vol. 3399, Springer-Verlag, Berlin, pp. 417-428.

Li, Y.-C., Yeh, J.-S. and Chang, C.-C. (2005*b*), Direct candidates generation: a novel algorithm for discovering complete share-frequent itemsets, *in* 'Proceedings of the 2nd Intl. Conf. on Fuzzy Systems and Knowledge Discovery (FSKD)', Lecture Notes in Artificial Intelligence, Vol. 3614, Springer-Verlag, Berlin, pp. 551-560.

Liu, G., Tsai, Lu, H. and Yu, J. X. (2007), 'CFP-tree: A compact disk-based structure for storing and querying frequent itemsets', *Information Systems*, Vol. 32, pp. 295-319.

Sucahyo, Y. G., Gopalan, R. P. and Rudra, A. (2003), 'Efficient mining frequent patterns from Dense Datasets Using a Cluster of Computers', AI 2003: Advances in Artificial Intelligence, LNAI 2903, pp. 233-244.

Tanbeer, S. K., Ahmed, C. F., Jeong, B. -S. and Lee, Y. -K. (2008), CP-tree: A tree structure for single pass frequent pattern mining, *in* 'Proceedings of the 12th Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD)', pp. 1022-1027.

Verma, K. and Vyas, O.P. (2005), 'Efficient calendar based temporal association rule', *SIGMOD Record*, Vol. 34, no. 3, pp. 63-70.

Wang, J., Han, J., Lu, Y. and Tzvetkov, P. (2005), 'TFP: an efficient algorithm for mining top-k frequent closed itemsets', *IEEE Transactions on Knowledge and Data Engineering*, Vol. 17, pp. 652-664.

Wei, J. -M., Yi, W. -G. and Wang, M. -Y. (2006), 'Novel measurement for mining effective association rules', *Knowledge-Based Systems*, Vol. 19, pp. 739-743.

Xiong, H., Tan, P.-N. and Kumar, V. (2006), 'Hyperclique Pattern Discovery', *Data Mining and Knowledge Discovery*, Vol. 13, pp. 219-242.

Ye, F. -Y., Wang, J. -D. and Shao, B. -L. (2005), New algorithm for mining frequent itemsets in sparse database, *in* 'Proceedings of the 4th International Conference on Machine learning and Cybernetics', pp. 1554-1558.