2006 IEEE International Conference on
Systems, Man, and Cybernetics
October 8-11, 2006, Taipei, Taiwan

A Fast Updated Frequent Pattern Tree

Tzung-Pei Hong, Jun-Wei Lin, and Yu-Lung Wu

Abstract—**In this paper, we attempt to modify the FP-tree construction algorithm for efficiently handling new transactions. A fast updated FP-tree (FUFP-tree) structure is proposed, which makes the tree update process become easier. An incremental FUFP-tree maintenance algorithm is also proposed for reducing the execution time in reconstructing the tree when new transactions are inserted. Experimental results show that the proposed FUFP-tree maintenance algorithm runs faster than the batch FP-tree construction algorithm for handling new transactions and generates nearly the same tree structure as the FP-tree algorithm. The proposed approach can thus achieve a good trade-off between execution time and tree complexity**

I. Introduction

Years of effort in data mining have produced a variety of efficient techniques. Depending on the type of databases processed, these mining approaches may be classified as working on transaction databases, temporal databases, relational databases, and multimedia databases, among others. On the other hand, depending on the classes of knowledge derived, the mining approaches may be classified as finding association rules, classification rules, clustering rules, and sequential patterns [4], among others. Among them, finding association rules in transaction databases is most commonly seen in data mining [1][2][4][9][10][14][15][18][19].

Many algorithms for mining association rules from transactions have been proposed. Most of them were based on the *Apriori* algorithm [1], which generated and tested candidate itemsets level-by–level. This may cause iterative database scan and high computational cost. Han *et al.* thus proposed the Frequent-Pattern-tree (FP-tree) structure for efficiently mining association rules without generation of candidate itemsets [11]. The FP-tree [11] was used to compress a database into a tree structure which stored only large items. It was condensed and complete for finding all the frequent patterns. The construction process was executed tuple by tuple, from the first transaction to the last one. After that, a recursive mining procedure called FP-Growth was executed to derive frequent patterns from the FP-tree. They showed the approach could have a better performance than Apriori.

Both the Apriori and the FP-tree mining approaches belong to batch mining. That is, they must process all the transactions in a batch way. In real-world applications, new transactions are usually inserted into databases. In this case, the originally desired large itemsets may become invalid, or new large itemsets may appear in the resulting updated databases [6][7][13][17][20]. Designing an efficient algorithm that can maintain association rules as a database grows is thus critically important.

One noticeable incremental mining algorithm was the Fast Updated Algorithm (called FUP), which was proposed by Cheung *et al.* [6] for avoiding the shortcomings mentioned above. In this paper, we attempt to modify the batch procedure of the FP-tree algorithm for incremental mining based on the FUP concept. A fast updated FP-tree (FUFP-tree) structure is proposed, which will make the tree update easier. It is similar to the FP-tree structure except that the links between parent nodes and their child nodes are bi-directional. Besides, the counts of the sorted frequent items are also kept in the Header_Table of the FP-tree algorithm. Bi-directional linking and storing the counts in the Header_Table will help fasten the maintenance process. An incremental FUFP-tree maintenance algorithm is then proposed for processing newly inserted transactions. It first partitions items into four parts according to whether they are large or small in the original database and in the new transactions. Each part is then processed in its own way. The Header_Table and the FUFP-tree are correspondingly updated whenever necessary. Experimental results also show the proposed FUFP-tree maintenance algorithm has a good performance.

The remainder of this paper is organized as follows. Related works are reviewed in Section 2. The proposed FUFP structure and maintenance algorithm are described in Section 3. An example is given to illustrate the proposed algorithm in Section 4. Experimental results for showing the performance of the proposed algorithm are provided in Section 5. Conclusions are given in Section 6.

II. Review of Related Works

In this section, some related researches about mining of association rules are briefly reviewed. They are the FUP algorithm and the FP-tree algorithm.

A. The FUP algorithm

In real-world applications, transaction databases usually grow over time and the association rules mined from them must be re-evaluated. Some new association rules may be generated and some old ones may become invalid.

This research was supported by the National Science Council of the Republic of China under contract NSC 94-2213-E-390-005.

T. P. Hong is with the Department of Electrical Engineering, National University of Kaohsiung, Kaohsiung, 811, Taiwan, R.O.C. (corresponding author; phone: +886+7+5919191; fax: +886+7+5919374; e-mail: tphong@nuk.edu.tw).

J. W. Lin is with the Department of Information Management, I-Shou University, Kaohsiung, 84008, Taiwan, R.O.C. (e-mail: m9322028@stmail.isu.edu.tw).

Y. L. Wu is with the Department of Information Management, I-Shou University, Kaohsiung, 84008, Taiwan, R.O.C. (e-mail: wuyulung@isu.edu.tw).

Conventional batch-mining algorithms solve this problem by re-processing the entire new databases when new transactions are inserted into original databases. They, however, require lots of computational time and waste existing mined knowledge.

Cheung *et al.* thus proposed the FUP algorithm [6] to effectively handle new transactions for maintaining association rules. Considering an original database and some new inserted transactions, the following four cases (illustrated in Figure 1) may arise:

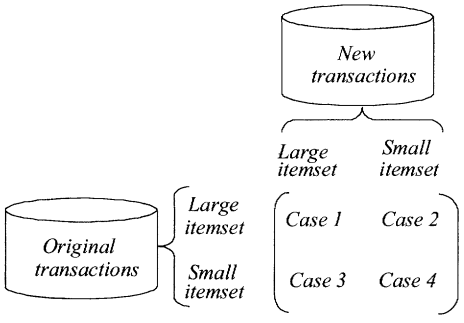

Fig.1. Four cases when new transactions are inserted into existing databases.

Since itemsets in Case 1 are large in both the original database and the new transactions, they will still be large after the weighted average of the counts. Similarly, itemsets in Case 4 will still be small after the new transactions are inserted. Thus Cases 1 and 4 will not affect the final large itemsets. Case 2 may remove existing large itemsets, and Case 3 may add new large itemsets.

B. The Frequent Pattern Tree

Han *et al.* proposed the Frequent-Pattern-tree structure (FP-tree) for efficiently mining association rules without generation of candidate itemsets [12]. The FP-tree mining algorithm consists of two phases. The first phase focuses on constructing the FP-tree from the database, and the second phase focuses on deriving frequent patterns from the FP-tree.

Three steps are involved in FP-tree construction. The database is first scanned to find all items with their frequency. The items with their supports larger than a predefined minimum support are selected as large 1-itemsets (items). Next, the large items are sorted in descending frequency. At last, the database is scanned again to construct the FP-tree according to the sorted order of large items. The construction process is executed tuple by tuple, from the first transaction to the last one. After all transactions are processed, the FP-tree is completely constructed.

The Header_Table is also built to help tree traversal. The Header_Table includes the sorted large items and their pointers (called frequency head) linking to their first occurrence nodes in the FP-tree. If more than one node have the same item name, they are also linked in sequence. Note that the links between nodes are single-directional from parents to children.

After the FP-tree is constructed from a database, a mining procedure called FP-Growth [12] is executed to find all large itemsets. FP-Growth does not need to generate candidate itemsets for mining, but derives frequent patterns directly from the FP-tree. It is a recursive process, handling the frequent items one by one and bottom-up according to the Header_Table. A conditional FP-tree is generated for each frequent item, and from the tree the large itemsets with the processed item can be recursively derived. Besides, many mining methods for finding association rules based on the FP-tree structure have also been proposed [8][12][16][21]. Some related researches are still in progress.

III. The Proposed Algorithm

An FUFP-tree must be built in advance from the original database before new transactions come. The FUFP-tree construction algorithm is the same as the FP-tree algorithm [11] except that the links between parent nodes and their child nodes are bi-directional. Bi-directional linking will help fasten the process of item deletion in the maintenance process. Besides, the counts of the sorted frequent items are also kept in the Header_Table.

When new transactions are added, the proposed incremental maintenance algorithm will process them to maintain the FUFP-tree. It first partitions items into four parts according to whether they are large or small in the original database and in the new transactions. Each part is then processed in its own way. The Header_Table and the FUFP-tree are correspondingly updated whenever necessary.

In the process for updating the FUFP-tree, item deletion is done before item insertion. When an originally large item becomes small, it is directly removed from the FUFP-tree and its parent and child nodes are then linked together. On the contrary, when an originally small item becomes large, it is added to the end of the Header_Table and then inserted into the leaf nodes of the FUFP-tree. It is reasonable to insert the item at the end of the Headrer_Table since when an originally small item becomes large due to the new transactions, its updated support is usually only a little larger than the minimum support. The FUFP-tree can thus be least updated in this way, and the performance of the proposed maintenance algorithm can be greatly improved. The entire FUFP-tree can be re-constructed in a batch way when a sufficiently large number of transactions are inserted. The details of the proposed algorithm are described below.

INPUT: An old database, its corresponding Header_Table storing the frequent items in descending order, its corresponding FUFP-tree, a support threshold *Sup* and a set of t new transactions.

OUTPUT: A new FUFP-tree for the updated database.

STEP 1: Scan the new transactions to get all the items and their counts.

STEP 2: Check whether the items are large in the new transactions by comparing their counts in the new transactions with the minimum count (t*Sup).

STEP 3: For each item I which are large both in the new transactions and in the original database (appearing in the Header_Table), do the following substeps (**Case 1**):

Substep 3-1: Set the new count $S^U(I)$ of I in the entire updated database as:

$$S^U(I)=S^D(I)+S^T(I),$$

where $S^D(I)$ is the count of I in the Header_Table (original database) and $S^T(I)$ is the count of I in the new transactions.

Substep 3-2: Update the count of I in the Header_Table as $S^U(I)$.

Substep 3-3: Put I in the set of *Insert_Items*, which will be further processed in STEP 6.

STEP 4: For each item I which are small in the new transactions but large in the original database (appearing in the Header_Table), do the following substeps (**Case 2**):

Substep 4-1: Set the new count $S^U(I)$ of I in the entire updated database as:

$$S^U(I)=S^D(I)+S^T(I).$$

Substep 4-2: If $S^U(I) \geq (d+t)*Sup$, item I will still be large after the database is updated; update the count of I in the Header_Table as $S^U(I)$ and add I to the set of *Insert_Items*.

Substep 4-3: If $S^U(I)<(d+t)*Sup$, item I will become small after the database is updated; Remove I from the Header_Table, connect each parent node of I directly to the corresponding child node of I, and remove I from the FUFP-tree.

STEP 5: For each item I which are large in the new transactions but small in the original database (not appearing in the Header_Table), do the following substeps (**Case 3**):

Substep 5-1: Rescan the original database to find out the transactions with item I, and calculate the count $S^D(I)$ of I in the original database.

Substep 5-2: Set the new count $S^U(I)$ of I in the entire updated database as:

$$S^U(I)=S^D(I)+S^T(I),$$

where $S^D(I)$ is the count of I obtained from Substep 5-1 and $S^T(I)$ is the count of I in the new transactions.

Substep 5-3: If $S^U(I) \geq (d+t)*Sup$, item I will be large after the database is updated; Add item I both in the set of *Insert_Items* and in the set of *Rescan_Items*, and put the transaction IDs with item I in the set of *Rescan_Transactions*.

STEP 6: Sort the items in the *Rescan_Items* in a descending order of their updated counts.

STEP 7: Insert the items in the *Rescan_Items* to the end of the Header_Table according to the descending order of their counts.

STEP 8: For each original transaction in the *Rescan_Transactions* with an item J existing in the *Rescan_Items*, if J has not been at the corresponding branch of the FUFP-tree for the transaction, insert J at the end of the branch and set its count as 1; Otherwise, add 1 to the count of the node J.

STEP 9: For each new transaction with an item J existing in the *Insert_Items*, if J has not been at the corresponding branch of the FUFP-tree for the new transactions, insert J at the end of the branch and set its count 1; Otherwise, add 1 to the count of J node.

In Step 4, a parent node may have the same child nodes after deletion. In this case, the child nodes are merged and their counts are summed up. In Step 8, a *corresponding branch* is the branch generated from the large items in a transaction and corresponding to the order of items appearing in the Header_Table. After Step 9, the final updated FUFP-tree is constructed. The new transactions can then be integrated into the original database. Based on the FUFP-tree, the desired association rules can then be found by the FP-Growth mining approach as proposed in [11].

IV. An Example

In this session, an example is given to illustrate the proposed incremental mining approach for maintaining an FUFP tree when new transactions are inserted. Table 1 shows a database to be used in the example. The database contains 10 transactions and 9 items, denoted *a to i*.

TABLE 1
THE ORIGINAL DATABASE IN THE EXAMPLE

Old database	
Transaction No.	Items
1	*a, b, c, d, e, g, h*
2	*a, b, f, g*
3	*b, d, e, f, g*
4	*a, b, f, h*
5	*a, b, f, i*
6	*a, c, d, e, g, h*
7	*a, b, h, i*
8	*b, c, d, f, g*
9	*a, b, f, h*
10	*a, b, g, h*

Assume the support threshold is set at 50%. For the given database, the large 1-itemsets are *a, b, f, g* and *h,* from which the Header_Table can be constructed. The FUFP-tree are then formed from the database and the Header_Table, with the results shown in Figure 2.

Assume the five new transactions shown in Table 2 appear. The proposed incremental mining algorithm proceeds as follows.

STEP 1: The five new transactions are first scanned to get the items and their counts.

STEP 2: The counts of the items appearing in the new transactions are checked against with the minimum count. In this case, the minimum support is set at 0.5. The minimum count for an item to be large in the new transactions is thus 5*0.5 (= 2.5). In this example, items *a*, *b*, *c* and *d* are large in the new transactions, and the remaining ones are small.

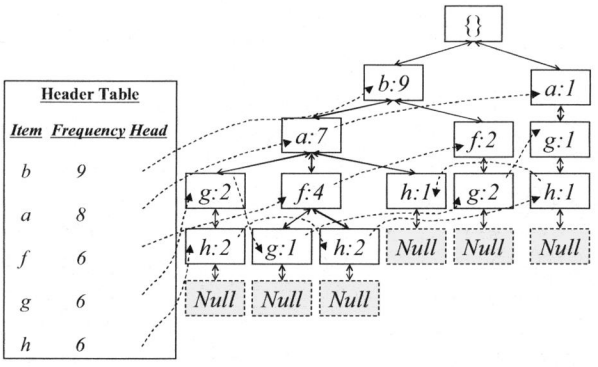

Fig. 2. The Header_Table and the FUFP-tree constructed

TABLE 2
THE FIVE NEW TRANSACTIONS

Transaction No.	Items
1	*a, b, d, e, g*
2	*c, e, i, f*
3	*a, b, d, h*
4	*a, b, c, d, g*
5	*b, c, d, i*

STEP 3: The items which are large both in the new transactions and in the original database are processed. In this example, items *a* and *b* satisfy the condition and are processed. Take item *a* as an example to illustrate the substeps. The count of item *a* in the Header_Table is 8, and the count in the new transactions is 3. The new count of item *a* is thus 8+3 (= 11). The frequency value of item *a* in the Header_Table is changed as 11. Item *a* is then put into the set *Insert_Items*. After STEP 3, *Insert_Items* = {*a, b*}.

STEP 4: The items which are small in the new transactions but large in the original database are processed. In this example, items *f, g* and *h* satisfy the condition and are processed. The minimum count for an item to be large in the updated database is 7.5. Take item *g* first as an example to illustrate the supsteps. The count of item *g* in the Header_Table is 6, and the count in the new transactions is 2. The new count of item *g* is thus 6+2 (= 8), larger than the minimum count. *g* is thus still large for the updated database, and is put into the set of *Insert_Items*. The frequency value of item *g* in the Header_Table is changed as 8. Moreover, both the new counts of items *f* and *h* in the Header_Table are calculated as 7, smaller than the minimum count. Items *f* and *h* are thus removed from the Header_Table. In this case, the FUFP-tree needs to be processed as well. Besides, item *a* has the same two child nodes (*g*:1 and *g*:2). Item *g* is then merged and its count is 3. After STEP 4, *Insert_Items* = {*a, b, g*}, and the updated FUFP-tree is shown in Figure 3.

STEP 5: The items which are large in the new transactions but small in the original database (not appearing in the Header_Table) are processed. In this example, items *c* and *d* satisfy the condition and will be processed. The original database is then rescanned to find the transactions with items *c* and *d* and their counts. The counts of items *c* and *d* are respectively 3 and 4 in the original database. The updated count of item *c* after the database is updated is then 3+4 (= 7), smaller than the minimum count. *c* is thus a small item and will not affect the Header_Table and the FUFP-tree. It is thus directly ignored. On the contrary, the count of item *d* is 4+4 (= 8), larger than the minimum count. *d* is thus a large itemset after the database is updated. Item *d* is then inserted into the Header_Table. *d* is then put into both the sets of *Insert_Items* and *Rescan_Items*. After STEP 5, *Insert_Items* = {*a, b, g, d*}, *Rescan_Items* = {*d*}, and *Rescan_Transactions* = {1, 3, 6, 8}. The corresponding transactions with their IDs in the *Rescan_Transactions* are shown in Table 3.

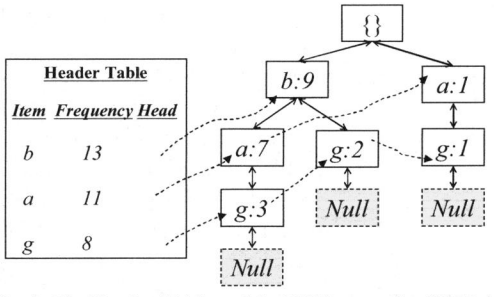

Fig. 3. The Header_Table and the FUFP-tree after STEP 4

TABLE 3
THE TRANSACTIONS WITH THEIR ID IN THE RESCAN_TRANSACTIONS

Transaction No.	Items
1	*a, b, c, d, g, h, e*
3	*b, d, e, f, g*
6	*a, c, d, e, g, h*
8	*b, c, d, f, g*

STEP 6: The items in the set of *Rescan_Items* are sorted in descending order of their updated counts. In this example, *Rescan_Items* contains only *d*, and no sorting is needed.

STEP 7: The items in the *Rescan_Items* are inserted into the end of the Header_Table. In this example, *d* is thus inserted.

STEP 8: The FUFP-tree is updated according to the original transactions with items existing in the *Rescan_Items*. In this example, *Rescan_Items* = *{d}*. The corresponding branches for the original transactions with *d* are shown in Table 4.

The first branch is then processed. This branch shares the same prefix *(b, a, g)* as the current FUFP-tree. A new node *(d:1)* is thus created and linked to *(g:3)* as its child. Note that the counts for items *b, a* and *g* are not increased since they have already been counted in the construction of the previous FUFP-tree. The same process is then executed for the other three corresponding branches. The final results are shown in Figure 4.

TABLE 4
THE CORRESPONDING BRANCHES FOR THE ORIGINAL TRANSACTIONS WITH ITEM *d*

Transaction No.	Items	Corresponding branches
1	*a, b, c, d, e, g, h*	*b, a, g, d*
3	*b, d, e, f, g*	*b, g, d*
6	*a, c, d, e, g, h*	*a, g, d*
8	*b, c, d, f, g*	*b, g, d*

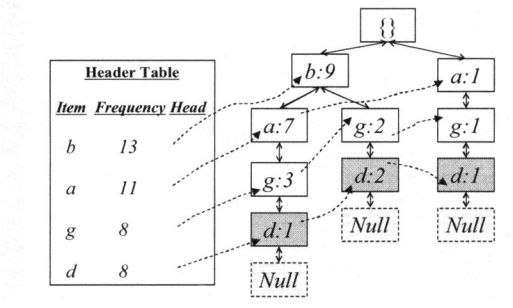

Fig. 4. The Header_Table and the FUFP-tree after Step 8

STEP 9: The FUFP-tree is updated according to the new transactions with items existing in the *Insert_Items*. In this example, *Insert_Items* = *{b, a, g, d}*. The corresponding branches for the new transactions with any of these items are shown in Table 5.

TABLE 5
THE CORRESPONDING BRANCHES FOR THE NEW TRANSACTIONS

Transaction No.	Items	Corresponding branches
1	*a, b, d, e, g*	*b, a, g, d*
3	*a, b, d, h*	*b, a, d*
4	*a, b, c, d, g*	*b, a, g, d*
5	*b, c, d, i*	*b, d*

The first branch shares the same prefix *(b, a, g, d)* as the current FUFP-tree. The counts for items *b, a, g,* and *d* are then increased by 1 since they have not yet counted in the construction of the previous FUFP-tree. The same process is then executed for the other three branches. The final results are shown in Figure 5.

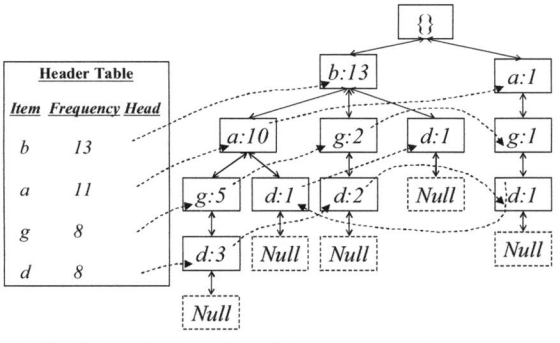

Fig. 5. The final FUFP-tree after all the new transactions are processed

V. Experimental Results

Experiments were made to compare the performance of the batch FP-tree construction algorithm and the incremental FUFP-tree maintenance algorithm for processing new transactions. When new transactions came, the batch FP-tree construction algorithm integrated new transactions into the original database and constructed a new FP-tree from the updated database. The process was executed whenever new transactions came. The incremental FUFP-tree maintenance algorithm processed new transactions incrementally in the way mentioned in Section 3.

The experiments were performed in C++ on an Intel x86 PC with a 2.8G Hz processor and 512 MB main memory and running the Microsoft Windows XP operating system. A real dataset called *BMS-POS* [22] were used in the experiments. This dataset was also used in the KDDCUP 2000 competition. The *BMS-POS* dataset contained several years of point-of-sale data from a large electronics retailer. Each transaction in this dataset consisted of all the product categories purchased by a customer at one time. There were 515,597 transactions with 1657 items in the dataset. The maximal length of a transaction was 164 and the average length of the transactions was 6.5.

The first 400,000 transactions were extracted from the BMS-POS database to construct an initial FP-tree. The next 5,000 transactions were then sequentially used each time as new transactions for the experiments. The minimum support was set at 4%. The execution times and the numbers of nodes obtained from both the batch FP-tree construction algorithm and the incremental FUFP-tree maintenance algorithm were compared. Figure 6 shows the execution times required by the batch FP-tree construction algorithm and by the FUFP-tree maintenance algorithm for processing each 5000 new transactions.

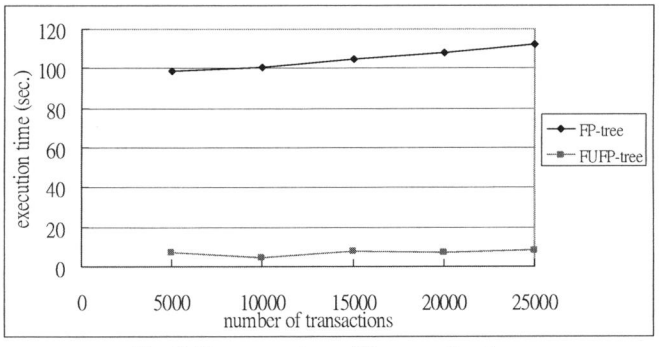

Fig. 6. The comparison of the execution time

In Figure 6, it easily observed that the execution time by the proposed approach was much less than that by the batch FP-tree construction algorithm for handling new transactions. Especially, when the transaction numbers in the original database became larger, the FUFP-tree maintenance algorithm had a better speed-up.

The FUFP-tree maintenance algorithm may generate a less concise tree than the FP-tree construction algorithm since the latter completely follows the sorted frequent items to build the tree. As mentioned above, when an originally small item becomes large due to new transactions, its updated support is usually only a little larger than the minimum

support. It is thus reasonable to put a new large item at the end of the Headrer_Table. The difference between the FP and the FUFP tree-structures will thus not be significant. For showing this effect, the numbers of nodes between the two algorithms are shown in Figure 7.

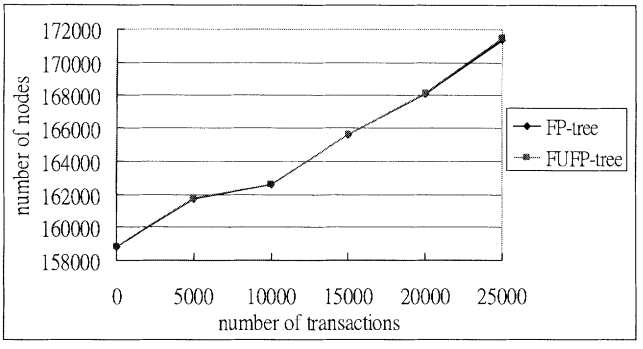

Fig. 7. The comparison of the numbers of nodes

It is observed from Figure 7 that the FUFP-tree maintenance algorithm generated nearly the same nodes as the FP-tree construction algorithm. The effectiveness of the FUFP-tree maintenance algorithm is thus acceptable.

VI. Conclusion

In this paper, we have proposed the FUFP maintenance structure and algorithm to efficiently and effectively handle new transaction insertion in data mining. When new transactions are added, the proposed incremental maintenance algorithm processes them to maintain the FUFP-tree. It first partitions items into four parts according to whether they are large or small in the original database and in the new transactions. Each part is then processed in its own way. The Header_Table and the FUFP-tree are correspondingly updated whenever necessary. It is reasonable to insert a new large item at the end of the Header_Table since when an originally small item becomes large due to new transactions, its updated support is usually only a little larger than the minimum support.

Experimental results also show that the proposed FUFP-tree maintenance algorithm runs faster than the batch FP-tree construction algorithm for handling new transactions and generates nearly the same tree structure as the FP-tree algorithm. The proposed approach can thus achieve a good trade-off between execution time and tree complexity.

References

[1] R. Agrawal, T. Imielinksi and A. Swami, "Mining association rules between sets of items in large database," *The ACM SIGMOD Conference,* pp. 207-216, 1993.

[2] R. Agrawal and R. Srikant, "Fast algorithm for mining association rules," *The International Conference on Very Large Data Bases*, pp. 487-499, 1994.

[3] R. Agrawal and R. Srikant, "Mining sequential patterns," *The Eleventh IEEE International Conference on Data Engineering,* pp. 3-14, 1995.

[4] R. Agrawal, R. Srikant and Q. Vu, "Mining association rules with item constraints," *The Third International Conference on Knowledge Discovery in Databases and Data Mining,* pp. 67-73, 1997.

[5] M.S. Chen, J. Han and P.S. Yu, "Data mining: An overview from a database perspective," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 8, No. 6, pp. 866-883, 1996.

[6] D.W. Cheung, J. Han, V.T. Ng, and C.Y. Wong, "Maintenance of discovered association rules in large databases: An incremental updating approach," *The Twelfth IEEE International Conference on Data Engineering,* pp. 106-114, 1996.

[7] D.W. Cheung, S.D. Lee, and B. Kao, "A general incremental technique for maintaining discovered association rules," *In Proceedings of Database Systems for Advanced Applications,* pp. 185-194, 1997.

[8] C. I. Ezeife, "Mining incremental association rules with generalized FP-tree," *The 15th Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, pp. 147-160, 2002.

[9] T. Fukuda, Y. Morimoto, S. Morishita and T. Tokuyama, "Mining optimized association rules for numeric attributes," *The ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 182-191, 1996.

[10] J. Han and Y. Fu, "Discovery of multiple-level association rules from large database," *The Twenty-first International Conference on Very Large Data Bases*, pp. 420-431, 1995.

[11] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation" *The 2000 ACM SIGMOD International Conference on Management of Data*, 2000.

[12] J. L. Koh and S.F. Shieh, "An efficient approach for maintaining association rules based on adjusting FP-tree structures," *The Ninth International Conference on Database Systems for Advanced Applications*, pp. 417-424, 2004.

[13] M.Y. Lin and S.Y. Lee, "Incremental update on sequential patterns in large databases," *The Tenth IEEE International Conference on Tools with Artificial Intelligence,* pp. 24-31, 1998.

[14] H. Mannila, H. Toivonen, and A.I. Verkamo, "Efficient algorithm for discovering association rules," *The AAAI Workshop on Knowledge Discovery in Databases*, pp. 181-192, 1994.

[15] J.S. Park, M.S. Chen, P.S. Yu, "Using a hash-based method with transaction trimming for mining association rules," *IEEE Transactions on Knowledge and Data Engineering,* Vol. 9, No. 5, pp. 812-825, 1997.

[16] Y. Qiu, Y. J. Lan and Q. S. Xie, "An improved algorithm of mining from FP-tree," *The Third International Conference on Machine Learning and Cybernetics*, pp 26-29, 2004.

[17] N. L. Sarda and N. V. Srinivas, "An adaptive algorithm for incremental mining of association rules," *The Ninth International Workshop on Database and Expert Systems*, pp. 240-245, 1998.

[18] R. Srikant and R. Agrawal, "Mining generalized association rules," *The Twenty-first International Conference on Very Large Data Bases*, pp. 407-419, 1995.

[19] R. Srikant and R. Agrawal, "Mining quantitative association rules in large relational tables," *The 1996 ACM SIGMOD International Conference on Management of Data*, pp. 1-12, Montreal, Canada, 1996.

[20] S. Zhang, "Aggregation and maintenance for database mining," *Intelligent Data Analysis*, Vol. 3, No. 6, pp. 475-490, 1999.

[21] O. R. Zaiane and E. H. Mohammed, "COFI-tree mining: A new approach to pattern growth with reduced candidacy generation," *IEEE International Conference on Data Mining*, 2003.

[22] Z. Zheng, R. Kohavi, L. Mason, "Real world performance of association rule algorithms", *The International Conference on Knowledge Discovery and Data Mining*, pp. 401-406, 2001.