A Space-Time Trade Off for FUFP-trees Maintenance

Bac Le[1], Chanh-Truc Tran[1], Tzung-Pei Hong[2], and Bay Vo[3]

[1] University of Science, Ho Chi Minh City, Viet Nam
lhbac@fit.hcmus.edu.vn, tructc@gemadept.com.vn
[2] Department of CSIE, National University of Kaohsiung, Taiwan, R.O.C.
tphong@nuk.edu.tw
[3] Information Technology College, Ho Chi Minh City, Viet Nam
vdbay@itc.edu.vn

Abstract. In the past, Hong *et al.* proposed an algorithm to maintain the fast updated frequent pattern tree (FUFP-tree), which was an efficient data structure for association-rule mining. However in the maintenance process, the counts of infrequent items and the IDs of transactions with those items were determined by rescanning all the transactions in the original database. This step might be quite time-consuming depending on the number of transactions in the original database and the number of rescanned items. This study improves that approach by storing 1-items during the maintenance process and based on the properties of FUFP-trees, such that the rescanned items and inserted items are processed more efficiently to reduce execution time. Experimental results show that the improved algorithm needs some more memory to store infrequent 1-items but the performance is better than the original one.

Keywords: data mining, frequent itemset, FUFP-tree, infrequent itemset, incremental mining.

1 Introduction

Data mining is one of the most interesting subjects with many techniques and algorithms developed [1]. Among the research topics of data mining, improving the efficiency of mining association rules from transaction databases has attracted much attention [1-11]. The first several algorithms for mining association rules were based on the *Apiori* algorithm [2], which repeatedly scanned a database to generate and process candidate itemsets level by level and thus needed a high computational cost. In 2000, the frequent pattern-tree (FP-tree) structure was proposed by Han *et al.* [6] for efficiently mining association rules without the generation of candidate itemsets. In real-world applications, a transaction database keeps being updated, and insertion is a very common operation. Efficient maintenance algorithms are thus needed when transactions are inserted [8-9]. In 2008, the incremental fast updated frequent pattern-tree (FUFP-tree) maintenance algorithm for handling transaction insertion was proposed [8]. In that approach, the FUFP-tree is incrementally handled without reconstructing the FUFP-tree from the beginning. However, the original database needs to be rescanned to determine the occurrence of infrequent items, which are not stored during

A. Selamat et al. (Eds.): ACIIDS 2013, Part II, LNAI 7803, pp. 206–214, 2013.

the maintenance process, and to determine the transaction IDs in which the rescanned items appear. This paper improves the above approach for transaction insertion by storing 1-items during the maintenance process and using the properties of FUFP-trees, such that the rescanned items and inserted items are processed more efficiently to reduce execution time.

2 Review of FUFP-trees

An FUFP-tree [8] is similar to an FP-tree except that it has bi-directional links between parent nodes and their child nodes. When new transactions are inserted to the original database, Hong *et al.*'s algorithm processes them to maintain the FUFP-tree without reconstructing it from the updated database. Depending on whether items are frequent (large) in the original database and in the new transactions, there are 4 cases to consider, which are shown in **Table 1**. Each case is processed separately. The Header-Table and the FUFP-tree are then appropriately updated if necessary.

Table 1. Four cases for transaction insertion [8]

Case	*Org. DB*	*New Trans*	*Results*
1	Frequent	Frequent	Always Frequent
2	Frequent	Infrequent	Determined from existing info.
3	Infrequent	Frequent	Determined by rescanning DB
4	Infrequent	Infrequent	Always infrequent

There are some points which can be improved in the original approach. When the original approach processes the items in case 3, the transactions in the original database need to be rescanned for determining the occurrences of infrequent items, which are not stored during the maintenance process. This step is thus the most time-consuming step. The computation time of this step is positively related to the number of transactions in the original database, the number of items in each transaction (the length of each transaction) and the number of items in the set of rescanned items.

3 Improved Algorithm

3.1 Notations

D, T, U: the original database, new transactions, updated database, respectively;
Sup: the minimum support threshold for frequent itemsets;
minSup_Org, *minSup_New*, *minSup*: *the* minimum support count of *D*, *T*, *U*, respectively;
$Count_{Org}(I)$, $Count_{New}(I)$, $Count_{Upd}(I)$: frequency of *I* in *D*, *T*, *U,* respectively;
Flist, *IFlist*: the list of large and small items in *D,* respectively*;*
Flist_New, *IFlist_New*: the list of large and small items in *T,* respectively*;*
Item_Case1, *Item_Case2*, *Item_Case3*, *Item_Case4:* list of items of the four cases, respectively;

Items: a temporary list to store items;
Htable: the *Header-Table* of FUFP-tree;
FUFP_tree: the current FUFP-tree;
Rescan_Items: the list of items to update the FUFP-tree based on the original database;
Insert_Items: the list of items to update the FUFP-tree based on new transactions;
Corresponding branch: the branch generated from the frequent items in a transaction according to the order of items appearing in *Header-Table.*

3.2 Proposed Algorithm

The details of the improved algorithm are shown below.

INPUT: Original database (*D*), Header-Table (*Htable*), FUFP-tree (*FUFP_tree*), support threshold (*Sup*), set of *t* new transactions (*T*).

OUTPUT: A new FUFP-tree for the updated database (*U*).

STEP 1: Scan the new transactions *T* to find their items and counts, and store large items into *Flist_New* and small items into *IFlist_New*.
STEP 2: Based on *Flist*, *IFlist, Flist_New* and *IFlist_New*, find and store items into *Items_Case1*, *Items_Case2*, *Items_Case3 and Items_Case4*, respectively.
STEP 3: For each item *I* in *Items_Case1*, do the following substeps:
Substep 3-1: The new count of *I* in *U*: $Count_{Upd}(I) = Count_{Org}(I) + Count_{New}(I)$.
Substep 3-2: Set the count of *I* in *Htable* = $\text{Count}_{Upd}(I)$.
Substep 3-3: Set the count of *I* in *Flist* = $\text{Count}_{Upd}(I)$.
Substep 3-4: Add *I* to the set of *Insert_Items*.
STEP 4: For each item *I* in *Items_Case2*, do the following substeps:
Substep 4-1: The new count of *I* in *U*: $Count_{Upd}(I) = Count_{Org}(I) + Count_{New}(I)$.
Substep 4-2: Set the count of *I* in *Flist* = $\text{Count}_{Upd}(I)$.
Substep 4-3: If ($Count_{Upd}(I) \geq minSup$), item *I* will still be large in updated DB; update the count of *I* in *Htable* as $\text{Count}_{Upd}(I)$ and add *I* to the set of *Insert_Items*.
Substep 4-4: If ($Count_{Upd}(I) < minSup$), item *I* will become small in updated DB; move *I* from *Flist* to *IFlist,* and remove *I* from the *Htable* and the *FUFP-tree*.
STEP 5: For each item *I* in *Items_Case3*, do the following substeps:
Substep 5-1: The new count of *I* in *U*: $Count_{Upd}(I) = Count_{Org}(I) + Count_{New}(I)$.
Substep 5-2: Set the count of *I* in *IFlist* = $\text{Count}_{Upd}(I)$.
Substep 5-3: If ($Count_{Upd}(I) \geq minSup$), add *I* both to *Insert_Items* and *Rescan_Items*.
STEP 6: Sort the items in *Rescan_Items* in descending order of their updated counts.
STEP 7: Insert the items in the *Rescan_Items* to the end of the *Htable* according to the descending order of their counts and move *I* from *IFlist* to *Flist.*
STEP 8: Update the *FUFP-tree* according to the set of *Rescan_Items*. For each transaction *J* in the original database, do the following substeps:
Substep 8-1: Determine which items of *Rescan_Items* appear in *J*, and store the results to a temporary list *Items*. If the list *Items* has no items, it means that there is no items of *Rescan_Items* appearing in *J*, and redo substep 8-1 with next transaction *J*.

Substep 8-2: Find the corresponding branch *B* of *J* in *FUFP-tree*, and store *B* to the temporary branch, *Branch*.
Substep 8-3: For each item *I* in *Items*, if *I* appears in the corresponding branch *Branch*, add 1 to the count of the node *I* and remove node *I* from *Branch* (from the properties of FUFP-trees, if a node in a specific branch is different from the others, it should not be considered in the next run after being processed. This will speed up the algorithm); otherwise, insert *I* at the end of the branch, set its count as 1, then re-find the new corresponding branch *B*, and store *B* to *Branch*.
STEP 9: Update the FUFP-tree according to the set of *Insert_Items*. For each transaction *J* in the new transactions, do the following substeps:
Substep 9-1: Determine which items of *Insert_Items* appear in *J*, and store the results to a temporary list *Items*. If the list *Items* has no items, it means that there is no items of *Insert_Items* appearing in *J*, and redo substep 9-1 with the next transaction *J*.
Substep 9-2: Find the corresponding branch *B* of *J* in *FUFP-tree* and store *B* to the temporary branch, *Branch*.
Substep 9-3: For each item *I* in *Items*, if *I* appears in the corresponding branch *Branch*, add 1 to the count of the node *I* and remove node *I* from *Branch*, (like substep 8-3); otherwise, insert *I* at the end of the branch, set its count as 1, re-find the new corresponding branch *B*, and store *B* to *Branch*.
STEP 10: For each item *I* in *Items_Case4*, do the following substeps:
Substep 10-1: The new count of *I* in *U*: $Count_{Upd}(I) = Count_{Org}(I) + Count_{New}(I)$.
Substep 10-2: Set the count of *I* in *IFlist* = $Count_{Upd}(I)$.

4 An Example

This section illustrates the proposed algorithm for maintaining an FUFP-tree after transactions are inserted. An original database with 10 transactions and 8 items, from *a* to *h*, is used in this example, which shown in **Table 2**.

Table 2. Original database used for the example

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

Assume the support threshold was set at 50%. For the original database, *minSup_Org* is 5, and the frequent 1-itemsets are *b*, *a*, *d*, and *c*, which are used to construct the *Header-Table.* The FUFP-tree is then built from the original database and *Header-Table*. **Fig.1** shows the results. Assume there are five transactions inserted to the original database as in **Table 3**.The proposed algorithm proceeds as follow.

STEP 1: The five new transactions are first scanned to get the items and their counts. Large items are stored in *Flist_New = {b:4, f:4, a:3, e:3}* and small items are stored in

IFlist_New = {c:2, d:2, g:1} based on *minSup_New* = 5 × 50% = 2.5 (3 by integer). The large items and small items of the original database are stored in *Flist = {b:9, a:8, d:6, c:5}* and *IFlist = {e:3, f:4, h:4, g:4}*, respectively, during the FUFP-tree construction.

Table 3. New inserted transactions

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

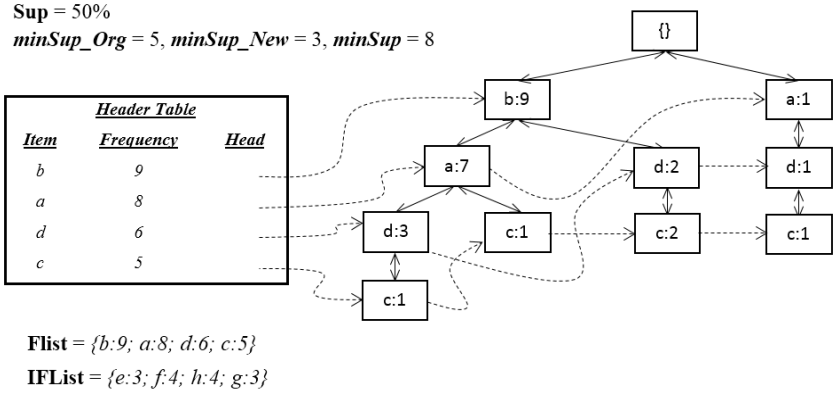

Fig. 1. FUFP-tree and *Header-Table* for the example

STEP 2: From *Flist*, *IFlist*, *Flist_New*, *IFlist_New*, the items of the 4 cases are calculated. In case 1, the items which appear both in *Flist* and *Flist_New* are stored in *Items_Case1* (= {*b, a*}). In case 2, the items which appear in *Flist* but don't exist in *Flist_New* are stored in *Items_Case2* (= {*d, c*}). In case 3, the items which appear in *Flist_New* but do not exist in *Flist* are stored in *Items_Case3* (= {*f, e*}). In case 4, the items which appear in *IFlist* but do not exist in *Flist_New* are stored in *Items_Case4* (= {*h, g*}).

STEP 3 to STEP 5: Each item in *Items_Case1, Items_Case2* and *Items_Case3* are processed by its individual step. After STEP 5, *Insert_Items = {b, a, d, f}* and *Rescan_Items = {f}*. *FUFP-tree*, *Header-Table*, *Flist* and *IFlist* are also updated correspondingly.

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

STEP 7: The items in the *Rescan_Items* are inserted to the end of the *Header-Table* according to the descending order of their counts. Thus, *f* is added to the end of *Header-Table*, and then *f* is moved from *IFlist* to *Flist.* The results after STEP 7 are shown in **Fig. 2**.

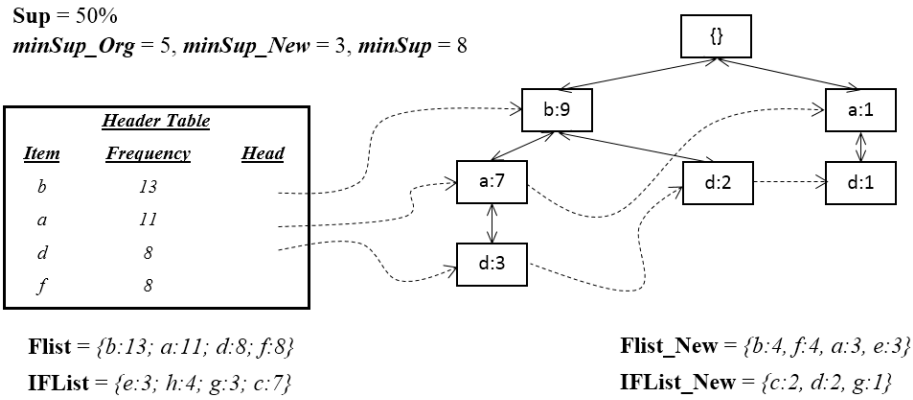

Fig. 2. FUFP-tree, *Header-Table*, *Flist* and *IFlist* after step 7 has been processed

STEP 8: The FUFP-tree is updated according to the transactions in the original database and the *Rescan_Items* (= {*f*}). **Table 4** shows the corresponding branches of the original database with items in *Rescan_Items*.

Table 4. Original transactions and items appear in *Rescan-Items*

No	Original DB	Items	Cor. branch	No	Original DB	Items	Cor. branch
1	*a, b, c, d, e*	-	-	6	*a, c, d, e, g*	-	-
2	*a, b, c, f, h*	*f*	*b* → *a*	7	*a, b, h*	-	-
3	*b, c, d, e, g*	-	-	8	*b, c, d, g*	-	-
4	*a, b, f, h*	*f*	*b* → *a* → *f*	9	*a, b, d, f*	*f*	*b* → *a* → *d*
5	*a, b, f*	*f*	*b* → *a* → *f*	10	*a, b, d, h*	-	-

In this example, each transaction in the original database is processed. Transactions 2, 4, 5 and 9 are processed because they include an item appearing in *Rescan_Items.* The results are shown in **Fig. 3**.

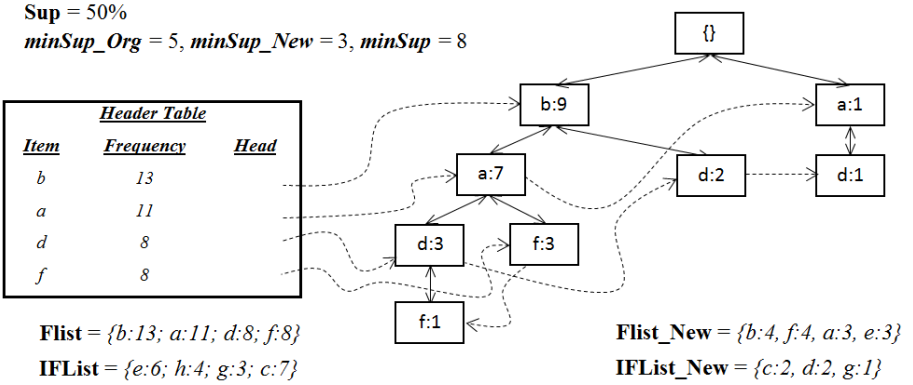

Fig. 3. FUFP-tree, *Header-Table*, *Flist* and *IFlist* after STEP 8

STEP 9: The FUFP-tree is updated according to the transactions in the new transactions and the *Insert_Items* (= {*b, a, d, f*}). **Table 5** shows the corresponding branches of the new transactions with items in *Insert_Items*. Each transaction with its corresponding branch in the new transactions is then processed.

Table 5. New transactions and items appear in *Insert-Items*

No	New trans.	Items	Cor. branch
1	*a, b, e, f*	*b, a, f*	$b \rightarrow a \rightarrow f$
2	*c, e, f*	*f*	-
3	*a, b, f*	*b, a, f*	$b \rightarrow a \rightarrow f$
4	*a, b, d, f, g*	*b, a, d, f*	$b \rightarrow a \rightarrow d \rightarrow f$
5	*b, c, d, e*	*b, d*	$B \rightarrow d$

STEP 10: The counts in *IFlist* of items in case 4 are then updated. Each item in *Items_Case4* is processed. After STEP 10, the final results are shown in **Fig. 4**.

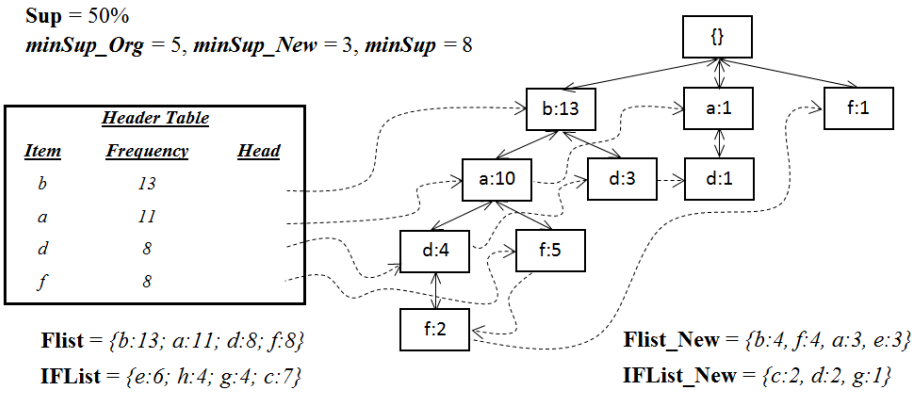

Fig. 4. FUFP-tree, *Header-Table*, *Flist* and *IFlist* after STEP 10 has been processed

5 Experimental Results

Experiments were programmed in C# on a laptop with an Intel 1.73 GHz quad-core CPU and 8GBs of RAM, running Windows 7 Ultimate 64 bits. Two real databases were used in the experiments. One is the BMS-POS and the other is MUSHROOM. The BMS-POS contained several years of point-of-sale data from a large electronics retailer with 515,597 transactions and 1,657 items. The maximal length of a transaction was 164 and the average length of the transactions was 6.5. There are 8,124 transactions with 22 items in the MUSHROOM. The parameters were set the same as Hong et al.'s. For the BMS-POS, the first 400,000 transactions were used to build the initial FUFP-tree and the next 5,000 transactions were sequentially used as new transactions; while for the MUSROOM, the first 5,000 transactions were used initially and the next 500 transactions were inserted each time. The *minSup* was set to 4%, 6%, and 8%. **Table 6** shows the execution time of the two algorithms with three different minimum support thresholds. Each value is the average execution time over 5 runs.

Table 6. Execution time of the two algorithms with different thresholds

%	Algorithms	Run time(s) of each 5,000 trans. inserted					BMS-POS
		5,000	10,000	15,000	20,000	25,000	
4	Hong et al.'s alg.	12.703	9.184	9.355	9.189	9.145	
	Proposed alg.	0.104	0.055	0.054	0.052	0.059	
6	Hong et al.'s alg.	10.861	9.157	9.270	9.173	9.176	
	Proposed alg.	0.128	0.054	0.055	0.056	0.055	
8	Hong et al.'s alg.	11.802	9.224	9.176	9.210	9.143	
	Proposed alg.	0.164	0.055	0.054	0.055	0.054	
%	Algorithms	Run time(s) of each 500 trans. inserted					MUSHROOM
		500	1,000	1,500	2,000	2,500	
4	Hong et al.'s alg.	0.367	0.278	0.291	0.304	0.314	
	Proposed alg.	0.031	0.024	0.021	0.019	0.017	
6	Hong et al.'s alg.	0.353	0.301	0.292	0.253	0.135	
	Proposed alg.	0.028	0.019	0.020	0.065	0.017	
8	Hong et al.'s alg.	0.363	0.382	0.288	0.241	0.139	
	Proposed alg.	0.031	0.141	0.018	0.019	0.019	

The results indicated that the proposed algorithm ran faster than the original approach. The main reasons are that Hong et al.'s approach has to rescan the transactions in the original database to determine the counts of infrequent items and the IDs of transactions in which the infrequent items appear, while the new approach gets the counts of infrequent items directly from *IFlist*, which is stored during FUFP-tree construction. Additionally, the proposed algorithm processes the *Recan_Items* and *Insert_Items* more efficiently based on the properties of the FUFP-tree. The number of nodes and the structure of the result trees generated are the same.

6 Conclusion and Future Work

An improved FUFP-tree maintenance approach for transaction insertion has been proposed. The proposed algorithm does not need to rescan the original database by storing the 1-items during the maintenance process. Moreover, based on the properties of the FUFP-tree, the item of a node in a specific branch is different from the others, thus the steps of updating the FUFP-tree according to *Rescan_Items* and *Insert_Items* are processed more efficiently by pruning out the processed item steps by steps. The execution time of the proposed algorithm is much lower than that of the original algorithm. The numbers of nodes of the FUFP-tree constructed by the two algorithms are the same. The proposed approach, however, requires some more memory to store 1-items. There is a trade-off between memory and execution time. The proposed approach is more efficient for large databases. For small databases with a few thousand of records, such as MUSHROOM, the difference is not very clear.

Lattice-based approaches for efficient mining association rules have been proposed in recent years [12-13]. In the future, we will study how to build frequent itemsets lattice when the database is changed. Besides, we will consider expanding the work in [14] to mine high utility itemsets.

Acknowledgement. This work was supported by Vietnam's National Foundation for Science and Technology Development (NAFOSTED).

References

1. Agrawal, R., Imielinski, T., Swami, A.: Database mining: A performance perspective. IEEE Transactions on Knowledge and Data Engineering 5(6), 914–925 (1993)
2. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: The 20th International Conference on Very Large Databases, pp. 487–499 (1994)
3. Agrawal, R., Srikant, R., Vu, Q.: Mining association rules with item constraints. In: The Third International Conference on Knowledge Discovery in Databases and Data Mining, pp. 67–73 (1997)
4. Fukuda, T., Morimoto, Y., Morishita, S., Tokuyama, T.: Mining optimized association rules for numeric attributes. In: The ACM Sigact-Sigmod Symposium on Principles of Database Systems, pp. 182–191 (1996)
5. Han, J., Fu, Y.: Discovery of multiple-level association rules from large database. In: The Twenty-first International Conference on Very Large Data Bases, pp. 420–431 (1995)
6. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: SIGMOD Conference, pp. 1–12 (2000)
7. Hong, T.P., Lin, C.W., Wu, Y.L.: Maintenance of fast updated frequent pattern trees for record deletion. Computational Statistics & Data Analysis 53(7), 2485–2499 (2009)
8. Hong, T.P., Lin, C.W., Wu, Y.L.: Incrementally fast updated frequent pattern trees. Expert Systems with Applications 34(4), 2424–2435 (2008)
9. Lin, C.W., Hong, T.P., Wu, Y.L.: The Pre-FUFP algorithm for incremental mining. Expert Systems with Applications 36(5), 9498–9505 (2009)
10. Mannila, H., Toivonen, H., Verkamo, A.I.: Efficient algorithm for discovering association rules. In: The AAAI Workshop on Knowledge Discovery in Databases, pp. 181–192 (1994)
11. Park, J.S., Chen, M.S., Yu, P.S.: Using a hash-based method with transaction trimming for mining association rules. IEEE Transactions on Knowledge and Data Engineering 9(5), 812–825 (1997)
12. Vo, B., Le, B.: Mining minimal non-redundant association rules using frequent itemsets lattice. Journal of Intelligent Systems Technology and Applications 10(1), 92–106 (2011)
13. Vo, B., Le, B.: Interestingness for association rules: Combination between lattice and hash tables. Expert Systems with Applications 38(9), 11630–11640 (2011)
14. Le, B., Nguyen, H., Vo, B.: An efficient strategy for mining high utility itemsets. International Journal of Intelligent Information and Database Systems 5(2), 164–176 (2011)