An Incremental FUSP-Tree Maintenance Algorithm

Chun-Wei Lin[1], Tzung-Pei Hong[2,3], Wen-Hsiang Lu[1] and Wen-Yang Lin[2]
[1]Department of Computer Science and Information Engineering
National Cheng Kung University
Kaohsiung, 811, Taiwan, R.O.C.
[2,4]Department of Computer Science and Information Engineering
National University of Kaohsiung
Kaohsiung, 811, Taiwan, R.O.C.
[3]Department of Computer Science and Engineering
National Sun Yat-sen University
Kaohsiung, 804, Taiwan, R.O.C.
[1]{p7895122; whlu}@mail.ncku.edu.tw, [2]{tphong, wylin}@nuk.edu.tw

Abstract

In this paper, we attempt to handle the maintenance of sequential patterns. New transactions may come from both the new customers and old customers. A fast updated sequential pattern tree (called FUSP-tree) structure is proposed to make the tree update process become easy. An incremental FUSP-tree maintenance algorithm is also proposed for reducing the execution time in reconstructing the tree. The proposed approach is expected to achieve a good trade-off between execution time and tree complexity.

1. Introduction

Due to the data explosion, mining useful information and helpful knowledge from large databases has evolved into an important research area [1][2][3]. Among them, finding sequential patterns in temporal transaction databases is important since it allows modeling of customer behavior in business.

Mining sequential patterns was first proposed by Agrawal et al. in 1995 [4], and is a non-trivial task. It attempts to find customer purchase sequences and to predict whether there is a high probability that when customers buy some products, they will buy some other products in later transactions. Transactions may continuously occur along with the time, so effectively and efficiently maintaining the mined sequential patterns becomes an important task. An intuitive approach is to re-mine the entire modified database to get correct sequential patterns for maintenance. However, when the database is massive in size, this will require considerable computation time.

Lin et al. proposed the FASTUP algorithm [10] to maintain sequential patterns by extending the FUP algorithm for association rules [5]. Their approach works well except when newly coming candidate sequences are not frequent (large) in the original database. If this occurs frequently, the database has to be often rescanned and the performance of the FASTUP algorithm will correspondingly decrease.

Han et al. then proposed the Frequent-Pattern-tree (FP-tree) structure for efficiently mining association rules without generation of candidate itemsets [7]. They showed the approach could have a better performance than the Apriori approach. Hong et al. also modified the FP-tree structure and designed the fast updated frequent pattern trees (FUFP-trees) [9] to efficiently handle newly inserted transactions for maintaining association rules based on the FUP concept [5].

Hong et al. also proposed the pre-large concept to reduce the need of rescanning original database for maintaining sequential patterns [8]. A pre-large sequence is not truly large, but may be large with a high probability in the future. A pre-large sequence was defined based on two support thresholds, a lower support threshold and an upper support threshold. Prelarge sequences act like buffers and are used to reduce the movement of sequences directly from large to small and vice-versa in the maintenance process. Therefore, when few new transactions are inserted, the original small sequences will at most become pre-large and cannot become large, thus reducing the amount of rescanning necessary.

Cheng et al. then further proposed the IncSpan (Incremental mining of sequential patterns) algorithm

978-0-7695-3382-7/08 $25.00 © 2008 IEEE
DOI 10.1109/ISDA.2008.126

[6] for efficiently maintaining sequential patterns in a tree structure. They used the frequent and semi-frequent sequences, which were a little like large and pre-large sequences in Hong et al's approach, to speed up the maintenance for items appended to old sequences, and used the IncSpan tree to keep the frequent and semi-frequent sequences. Their approach did not generate the frequent sequences from the tree as in the FP-tree, but used the tree to maintain the frequent sequence.

In general, when new transactions occur, there are two possible cases below.

Case 1: The new transactions come from some old customers in the original databases;

Case 2: The new transactions come from new customers not in the original database.

The IncSpan algorithm considered only the first case, in which new transactions were appended to the existing customer sequence.

In this paper, we thus attempt to extend the concept of the FUFP-tree maintenance approach for efficiently handling new transactions and finding sequential patterns. The proposed approach can handle both the above cases that the new transactions are from the new and the old customers. A fast updated sequential pattern tree (FUSP-tree) structure is proposed, which makes the tree update process become easy. The frequent items are first mined to build the initial FUSP tree. Like the FP tree, only frequent items are used to build the tree, such that the tree can be reduced. The complete customer sequences with only large items in the given database are kept in the FUSP tree. The advantage of this way is when frequent items are not changed, the approach doesn't need to rescan the original database, but can get the results only from the tree. After the tree is maintained, the final frequent sequences can then be found by a recursive method from the tree.

The customer IDs will be kept in the last nodes of the corresponding branch to reduce the database rescan in the maintenance process. An incremental FUSP-tree maintenance algorithm is also proposed for reducing the execution time in re-constructing the tree when new transactions come. The proposed approach can achieve a trade-off between execution time and tree complexity. This approach may be especially efficient for mining the traversal paths from the web log since only one item exists in a large itemset. Our structure is especially useful for this situation.

2. The FUSP-Tree Structure

In this paper, we will use a similar concept to maintain the tree structure called the FUSP tree, from which the sequential patterns may be derived. The concepts of the FUSP tree are extended from both the FUFP tree [9] and the IncSpan tree [6]. By taking the characters of the FUFP tree, the size of a customer sequence is reduced to make the FUSP tree compact since only the frequent items are kept. Besides, the complete customer sequences are kept in the FUSP tree to avoid and reduce the movement for rescanning the original database. The FUSP tree must be built in advance from the initial database before new transactions come. A FUSP tree is shown in Figure 2.

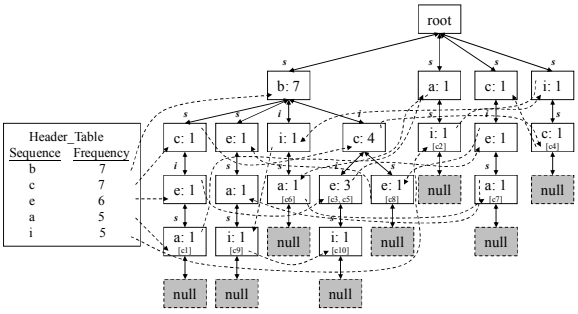

Figure 2: A FUSP tree

In Figure 2, only the frequent items (1-itemsets) are kept. Like the IncSpan algorithm [6], the link between two connected nodes is marked by the symbol *s* (representing the sequence relation) if the sequence is within the sequence relation in a sequence; otherwise, the link is marked by the symbol *i*, which indicates the sequence is within the itemset relation in a sequence. The sequence IDs are also kept in the last nodes of the corresponding branch which helps update the tree structure efficiently. In addition to the FUSP tree, the Header_Table is also kept to help find appropriate items or sequences in the tree. It sorts the frequent items initially in a descending order. Infrequent items will not be used in building the trees. The construction process is executed tuple by tuple, from the first customer sequence to the last one. After all the customer sequences are processed, the FUSP tree is completely constructed. The tree thus contains the complete customer sequences from the database.

3. The Proposed FUSP-tree Maintenance Algorithm

In this section, the FUSP-tree maintenance algorithm is proposed to handle the sequential patterns in incremental mining by maintaining the FUSP tree. As mentioned above, mining sequential patterns was first proposed by Agrawal et al. in 1995 [4]. Lin et al. then proposed the FASTUP algorithm [10] to maintain

sequential patterns by extending the FUP algorithm [5]. Considering an original database and some new transactions, the following four cases (illustrated in Figure 1) may arise:

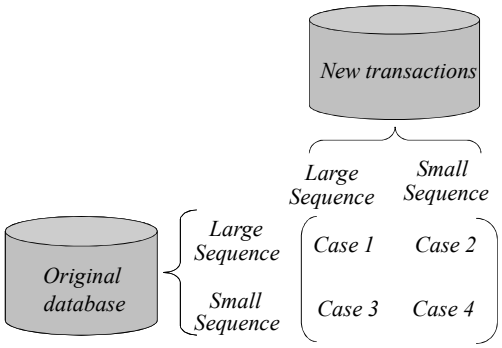

Figure 1: Four cases when new transactions are inserted into existing databases

The proposed approach will maintain the FUSP tree based on the FUP concept. When new transactions are added, the FUSP-tree maintenance algorithm will process them to maintain the FUSP tree. The entire FUSP tree can then be re-constructed in a batch way when a sufficiently large number of transactions have been inserted. The details of the proposed algorithm are described below.

The FUSP-tree maintenance algorithm:

INPUT: An old database which contains the D customer sequences, its corresponding Header_Table which sorts the frequent 1-itemsets initially in a descending order, its corresponding FUSP tree, a support threshold S, and a set of d transactions with t new customers.

OUTPUT: A new FUSP tree for the updated database.

STEP 1: Scan the new transactions to get all the 1-itemset and their counts grouped by the customer ID. If a 1-itemset has existed in an old customer sequence in the original database which can be found and traced from the Header_Table and the FUSP tree, the count of the 1-itemset remains unchanged; otherwise, the count of the 1-itemset will be increased by 1. For each 1-itemset I from the new transactions which do not appear in the Header_Table, keep its customer IDs in the set of *New_Cidseq*(I). The *New_Cusid*(I) will be processed in STEP 5.

STEP 2: Divide the 1-itemset from the new transactions into two parts according to whether they are large or small in the original database.

STEP 3: For each 1-itemset I which is 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 grouped by the customer ID.

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_Seqs*, which will be further processed in STEP 9.

STEP 4: For each 1-itemset 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 $S^U(I) \geq (t+D)*Sup$, I will 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_Seqs*.

Substep 4-3: If $S^U(I)<(t+D)*Sup$, 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 FUSP tree.

STEP 5: For each 1-itemset 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 calculate the count $S^D(I)$ of the 1-itemset I in the original database; put the IDs of the customer sequences with I in the set of *Rescan_Cidseq*(I).

Substep 5-2: For each 1-itemset I, find the number of intersection $S^{IS}(I)$ of the two sets *New_Cidseq*(I) and *Rescan_Cidseq*(I).

Substep 5-3: Set the new count $S^U(I)$ of I in the entire updated database as:
$S^U(I) = S^D(I) + S^T(I) - S^{IS}(I)$,
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 customer sequences.

Substep 5-4: If $S^U(I) \geq (t+D)*Sup$, I will be large after the database is updated; Add the sequence I both in the set of *Insert_Seqs* and the set of *Rescan_Seqs*.

STEP 6: Sort the 1-itemset in the *Rescan_Seqs* in a descending order of their updated counts.

STEP 7: Insert the 1-itemset in the *Rescan_Seqs* to the end of the Header_Table according to the descending order of their counts.

STEP 8: For each transaction in the original database with an itemset J existing in the *Rescan_Seqs*, if J has not been at the corresponding branch of the FUSP tree for the transaction, insert J at its sequential position of the branch in the original databases and set its count as 1; the inserted J should be marked as i or s according to whether the relation is a sequence-extended sequence or an itemset-extended sequence. Otherwise, add 1 to the count of the node J.

STEP 9: For each new transaction with 1-itemset I existing in the I*nsert_Seqs*, do the following substeps:

Substep 9-1: For the new transactions from old customers, if I has not been at a corresponding branch of FUSP tree, insert I at the end of the branch and set its count as 1; Otherwise, add 1 to the count of the node I.

Substep 9-2: For the new transactions from new customers not existing in the original databases, if I has not been at a corresponding branch of the FUSP tree for the new transactions, insert I at the end of the branch and set its count as 1; otherwise, add 1 to the count of the node I.

After STEP 9, the maintenance process is finished. Note that in STEP 8, a corresponding branch is the branch generated from the frequent itemsets in the transactions and corresponding to the order of itemsets in the databases. After STEP 9, the final updated FUSP tree is formed by the maintenance algorithm. Based on the FUSP tree, the desired large sequences for the updated database can be determined by a recursive mining procedure, which is similar but much more complex than those proposed in [6][7].

4. An Example

In this section, an example is given to illustrate the proposed incremental mining approach for sequential patterns based on the FUSP tree. Table 1 shows a database to be used in the example. It contains 10 customer sequences and 9 items, denoted *a* to *i*.

Table 2: The old customer sequences

Cust_ID	Customer Sequence
1	(*b*)(*ce*)(*a*)
2	(*a*)(*fi*)
3	(*d*)(*bce*)
4	(*i*)(*cg*)(*d*)
5	(*bce*)(*f*)(*h*)
6	(*bi*)(*a*)(*d*)
7	(*ce*)(*a*)(*g*)
8	(*bc*)(*e*)(*d*)
9	(*b*)(*e*)(*a*)(*i*)
10	(*bce*)(*f*)(*i*)

Assume the support threshold is set at 50%. For the given database, the large 1-itemsets are (*a*), (*b*), (*c*), (*e*) and (*i*), from which the Header_Table can be constructed. The FUSP tree is then found from the database and the Header_Table, with the results shown in Figure 2 in Section 3. Note that the IDs of the customer sequences are kept in the last nodes of the corresponding branches which make the tracing become easy. Assume that some new transactions appear, which have been transformed into customer sequences shown in Table 3.

Table 3: The new customer sequences

Cust_ID	Customer Sequence
2	(*bf*)(*a*)
5	(*f*)(*g*)
11	(*df*)(*i*)
12	(*ed*)
13	(*ce*)(*fg*)
14	(*bc*)(*f*)(*d*)

The proposed incremental mining algorithm for sequential patterns then proceeds according to the steps. At last, the results are shown in Figure 3.

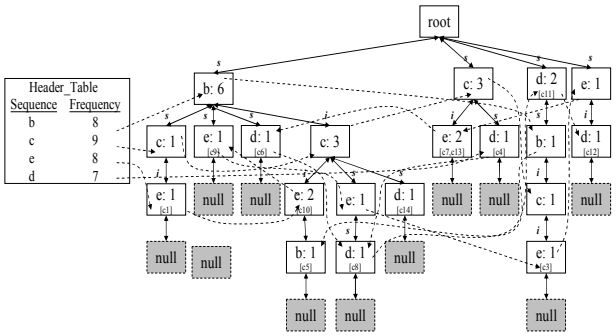

Figure 3: The final results

5. Conclusions

In this paper, the fast updated sequential pattern tree (FUSP tree) is proposed to maintain the sequential patterns in incremental mining. When new transactions come, the proposed incremental maintenance algorithm processes them to maintain the FUSP tree. The algorithm can work well for the transactions coming from old customers and new customers. The obtained tree complexity may not be optimal due to the frequency order of items may not be kept. We expect the proposed approach can thus achieve a good trade-off between execution time and tree complexity. Further verification will be done in the future.

6. 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, Washington DC, USA, 1993.

[2] R. Agrawal, T. Imielinksi and A. Swami, "Database mining: a performance perspective," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 6, pp. 914-925, 1993.

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

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

[5] 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.

[6] H. Cheng, X. Yan and J. Han, "IncSpan: incremental mining of sequential patterns in large database," *The ACM SIGKDD international conference on Knowledge discovery and data mining*, pp.527-532, 2004.

[7] J. Han, J. Pei and Y. Yin, "Mining frequent patterns without candidate generation," *The 2000 ACM SIGMOD International Conference on Management of Data*, pp. 1-12, 2000.

[8] T. P. Hong, C. Y. Wang, S. S. Tseng, "Incremental data mining for sequential patterns using pre-large sequences," *The International Multiconference on Systemics, Cybernetics and Informatics*, Vol. 14, pp. 543-548, 2001.

[9] T. P. Hong, C. W. Lin and Y. L. Wu, "Incrementally fast updated frequent pattern trees," *Expert Systems with Applications* , Vol. 34, Issue 5, pp. 2424-2435, 2008.

[10] 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.