A Subset-Lattice Algorithm for Mining High Utility Patterns over the Data Stream Sliding Window

Ye-In Chang, Chia-En Li, Rong-Fu Chen, Siang-Jia Du, Ching-Yi Yen
Department of Computer Science and Engineering
National Sun Yat-sen University
Kaohsiung, Taiwan
lice@db.cse.nsysu.edu.tw

ABSTRACT

High utility pattern mining considers each item with a distinct profit or price. The problem is that infrequent patterns may contribute a great number of profit, whereas frequent patterns may only contribute a small amount of profit. The SHU-Grow algorithm uses the tree-based data structure to mine high utility patterns. In such a structure, the SHU-Grow algorithm always records the estimated value of each pattern. Then, such an algorithm has to identify actual high utility patterns from the candidate patterns. In this paper, we propose the Subset-Lattice algorithm based on the sliding window model. Our algorithm utilizes the lattice structure to record the information of the transactions and to store relationship between the child node and the parent node. From the performance study, we show that our Subset-Lattice algorithm could provide better performance than the SHU-Grow algorithm both in the processing time and storage space.

Keywords: data mining, frequent pattern, high utility pattern, lattice structure, sliding window

I. INTRODUCTION

In the real world, each item has different profit and the number of items purchased by consumers could be not only one, but more than one. In utility mining, each item has internal utility value that represents the quantity of the item in each transaction, and external utility value such as profit or price. Based on these definitions, we can get the utility of each item and identify high utility patterns in high utility pattern mining [1-3]. In this utility framework, patterns with high utility or high purchase quantity will be identified as high utility patterns even if they occur infrequently. In other words, high utility pattern mining does not satisfy the downward closure property, which means that the item is infrequent, and its superset is also infrequent [4-13]. However, the superset of the low utility pattern will probably be a high utility pattern. So, high utility pattern mining is more difficult than traditional frequent pattern mining. In high utility pattern mining, it considers that each item with a distinct profit or price and non-binary item quantity in a transaction. On the other hand, the profit represents the importance of an item. It is the important difference with the previous frequent pattern mining. Thus, the high utility mining is based on two kinds of numeric data, the quantity and the profit of each item, and it can play an important role in market analysis. If the utility of an itemset is greater than the minimum utility threshold, this itemset is identified as a high utility pattern and reflects real world market data.

Ryang *et al*. proposed the *SHU-Grow* algorithm. They use *SHU-Tree* data structure to store transactions within the current window. The technology of their algorithm can handle several batches of transactions within the current sliding window. In the *SHU-Grow* algorithm, for each batch in the current sliding window, a utility counter with *b* batches is constructed. If an item is in the *i*th batch of the current sliding window, the *i*th batch is set to be the estimated utility of the item. When the window slides, the list of utility counter of nodes is needed to left-shift to delete the oldest batch. For example, the utility counter of node A is (30, 20, 0). If the oldest batch is deleted, the utility counter of node A becomes (20, 0, 0). Moreover, their algorithm has a header table to store the estimated utility of each single item called *RTWU*. When the oldest batch is deleted, the *RTWU* of item A will be decreased by 30 in the header table. The *SHU-Grow* algorithm needs to check each candidate pattern to decide whether it is a high utility pattern or not. Thus, they need to scan each transaction within the current window again.

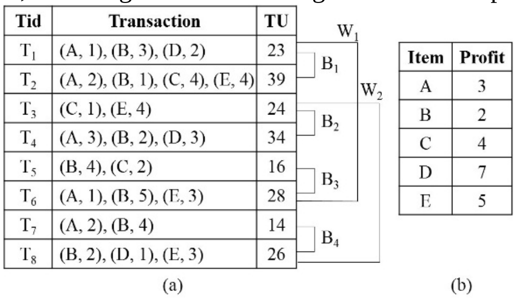
Therefore, in this paper, we propose an algorithm, *Subset-Lattice* algorithm, to find the result based on the real value, instead of the estimated value, of the pattern. In our data structure, we use a lattice to mine high utility patterns. We check the relation between the incoming transaction and the current transactions in the lattice structure, when the new transaction comes [14]. There are five relations which are concerned in our algorithm: (1) empty, (2) equivalent, (3) superset, (4) subset, and (5) intersection. Because the relations exists between the set and the subset, there is one advantage that there are fewer number of nodes in the lattice than those nodes stored in the tree of the *SHU-Grow* algorithm. We can calculate the real value of the each single item. Furthermore, our lattice structure requires fewer nodes than the *SHU-Tree*. Besides, our proposed algorithm records the real value rather than the estimated value of the pattern. Therefore, the *Subset-Lattice* algorithm has better performance than the *SHU-Grow* algorithm both in the processing time and storage space (the number of created nodes).

The rest of the paper is organized as follows. In Section 2, we give a brief description of the *SHU-Grow* algorithm. In Section 3, we present the proposed *Subset-Lattice* algorithm. In Section 4, we present the performance study of our algorithm and make a comparison between our algorithm and the *SHU-Grow* algorithm [15]. Finally, Section 5 gives the conclusion.

II. RELATED WORK

Ryang *et al*. proposed a sliding window based algorithm, *SHU-Grow* (Sliding window based High Utility Grow) [2] with a data structure *SHU-Tree* (Sliding window based High Utility

Tree) to mine high utility patterns from the data stream. Moreover, they use two techniques, *RGE* (Reducing Global Estimated utilities) and *RLE* (Reducing Local Estimated utilities). The problem is defined as mining high utility patterns within the current window. The mining result must be updated by every window sliding. Figure 1-(a) shows a data stream. In Figure 1-(a), each item i_m in a transaction T_f is associated with a quantity value which is called *internal utility* and denoted as $iu(i_m, T_f)$. In Figure 1-(b), let $I = \{i_1, i_2, ..., i_v\}$ be a finite set of distinct items, a transaction T_f be a subset of I, and a pattern P be a set of items $\{i_1, i_2, ..., i_l\}$ in I, where $P \in I$ and $1 \leq l \leq v$. Each item i_m in a database has a unit profit which is called *external utility* and denoted as $eu(i_m)$. If *TWU* of a pattern P in W_k is no smaller than $minutil_{Wk}$, where $minutil_{Wk}$ is the minimum utility threshold, P is a high transaction weighted utilization pattern.



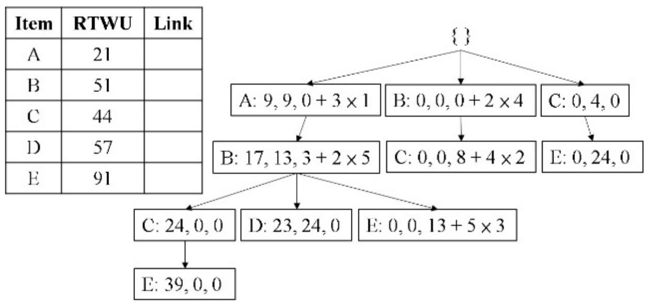
Tid	Transaction	TU
T_1	(A, 1), (B, 3), (D, 2)	23
T_2	(A, 2), (B, 1), (C, 4), (E, 4)	39
T_3	(C, 1), (E, 4)	24
T_4	(A, 3), (B, 2), (D, 3)	34
T_5	(B, 4), (C, 2)	16
T_6	(A, 1), (B, 5), (E, 3)	28
T_7	(A, 2), (B, 4)	14
T_8	(B, 2), (D, 1), (E, 3)	26

W_1
W_2
B_1
B_2
B_3
B_4
(a)
(b)

Item	Profit
A	3
B	2
C	4
D	7
E	5

Figure 1. (a) Data stream; (b) Profit table [2].

The framework of the algorithm is composed of three steps. In the first step, their method constructs a global tree through a single scan of the current window in data stream by the *RGE* technique. In the second step, it generates candidate patterns from the constructed tree by the *RLE* technique. In the last step, the algorithm identifiers a set of high utility patterns from the candidates. Meanwhile, the global tree can be updated, whenever a window is full and a new batch arrives by eliminating the oldest batch and reflecting the new one. In the *SHU-Tree*, each node N has the following basic elements: the item name, $N._{name}$; the more reduced overestimation utilities than *TWU* is called *RTWU*(Reduced *TWU*), $N._{nu}$; two node pointer, $N._{parent}$ and $N._{nodelink}$; a set of child nodes. $N._{tail}$ have an array of boolean values. If there are n batches in the current window, the number of node utilities in the counter is n. Figure 2 shows the constructed global *SHU-Tree* after the last batch B_3 is inserted.



Item	RTWU	Link
A	21	
B	51	
C	44	
D	57	
E	91	

Figure 2. The result of inserting transactions T_1-T_6.

III. The Subset-Lattice Algorithm

In this section, we propose the *Subset-Lattice* algorithm to identify the high utility patterns over data streams using the sliding window model.

A. Data Structure

In utility mining, each item has the individual profit and the non-binary form quantity in the transaction. The high utility patterns are the sets of items that contribute the profitable value, which satisfy the threshold, in the database. However, the "Downward Closure Property" does not hold in this technique [5, 6, 9, 10, 12, 13]. (Note that the "Downward Closure Property" mean that the item is infrequent, and its superset is also infrequent.). The lattice structure contains the root, nodes, and child-link as shown in Figure 3. The root is a start point, which has no information. When a new transaction comes, we search the lattice structure from the root. In each node, we record the itemset and the array *QSRecords*. The child-link points to the subset node. With the child-link, we can check the relationship between nodes and insert the node into the lattice structure easily. Moreover, we can update the number of items in an itemset efficiently by the array *QSRecords* to know the number of each item. The array *QSRecords* has the quantity-sequence representation of the quantity of each item. The size of the array *QSRecords* is |*Batch*|, *i.e.*, the number of batches in a window, and the length of the quantity-sequence is |*Item*|, *i.e.*, the size of a set of items in the database. In other words, |*Item*| is the longest length of distinct items. The table *CntTable* is an array which stores the quantity of each item in the current window. When the inserting process and the deleting process are performed, we will update the contents of the table *CntTable* according to the quantity-sequence. The table *SWTable* is an array which stores all transactions of the current window and the corresponding quantity-sequence of each transactions. When the inserting process and the deleting process are performed, we will update the contents of the table *SWTable* by the transaction existing in this window. This lattice structure has three advantages. First, the relationship between the new transaction and the current transactions can be easily understood by using this lattice structure. Second, we can update the number of items in an itemset efficiently. Third, we calculate the actual value rather than the estimated value.

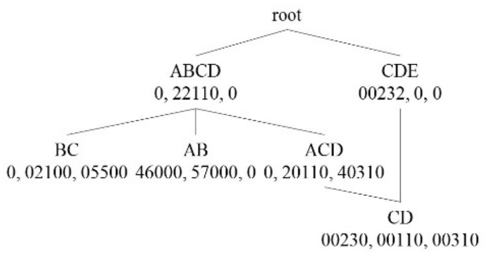


Figure 3. The lattice of the window W_1.

B. The Proposed Algorithm

Our algorithm has five main steps. First, we transform the itemset to the quantity sequence. Second, we check the relation between the new transaction and the current transactions, and the corresponding quantity-sequence of the current window is stored in the table *SWTable*. Next, we insert the quantity-sequence into the array *QSRecords* of the corresponding node and update its child nodes. Then, calculate the utility of the itemset on each node in the lattice structure. Finally, we examine the subset of a set of the current high utility patterns. In the first step, we will transfer the transaction into the quantity-sequence representation. We use the longest length of distinct items as the length of the quantity-sequence according to the lexical order. In the second step, we check the relation between the new transaction and the current transactions. There are five cases, as shown in Figure 4, in inserting itemsets into the lattice structure. When the current window becomes full, we will delete the information of the oldest batch (a set of transactions) and insert the information of a new batch. At this time, we set the array *QSRecords* of all nodes initially to zero (*i.e.*, (0, 0, 0)).

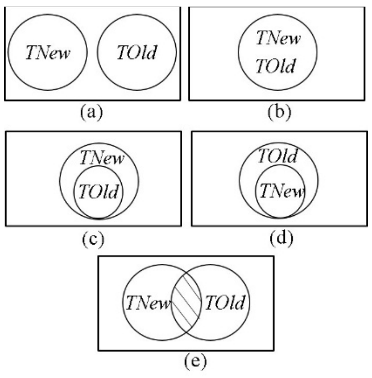
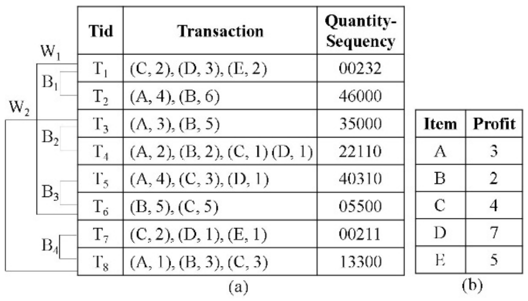


Figure 4. The set-relations diagram between the new transaction *TNew* and the old transaction *TOld*: (a) Case 1: Empty; (b) Case 2: Equivalent; (c) Case 3: Superset; (d) Case 4: Subset; (e) Case 5: Intersection.

In order to get the information of the quantity, we use the quantity-sequence representation to store the information of each transaction. The longest length of the quantity-sequence is the number of distinct items according to the lexical order. For the example of the transaction {(C, 2), (D, 3), (E, 2)}, we set the values to the corresponding quantity one by one according to the lexical order position. The quantity-sequence representation of {(C, 2), (D, 3), (E, 2)} is denoted as *QuantitySeq*, 00232, since there are five items {A, B, C, D, E}. Because this transaction does not contain item A and item B, the position of item A and item B are set to 0. The quantity-sequence of the window W_1 is shown in Figure 5-(a). During the processing of data insertion, we will record T_i and its related quantity-sequence in table *SWTable* as shown in Figure 5-(b).


W1
W2
B1
B2
B3
B4
(a)
(b)

Tid	Transaction	Quantity-Sequency
T_1	(C, 2), (D, 3), (E, 2)	00232
T_2	(A, 4), (B, 6)	46000
T_3	(A, 3), (B, 5)	35000
T_4	(A, 2), (B, 2), (C, 1) (D, 1)	22110
T_5	(A, 4), (C, 3), (D, 1)	40310
T_6	(B, 5), (C, 5)	05500
T_7	(C, 2), (D, 1), (E, 1)	00211
T_8	(A, 1), (B, 3), (C, 3)	13300

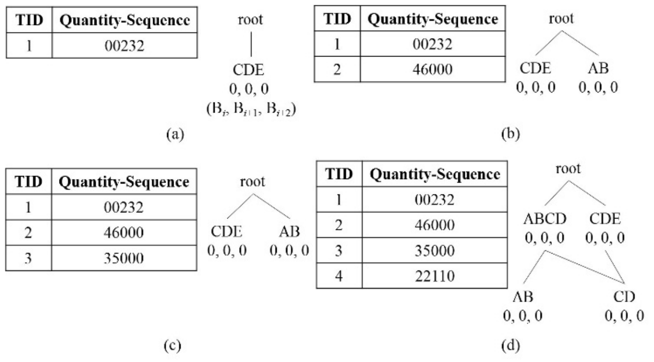
Item	Profit
A	3
B	2
C	4
D	7
E	5

Figure 5. (a) The quantity-sequence of each transaction in window W_1; (b) Profit table.

We use an example to describe our approach. Figure 5-(a) shows an example of the data stream and Figure 5-(b) shows an example of the profit table. We assume that the size of a batch is 2 transactions, the size of a window is 3 batches, and the minimum utility threshold is 22%. The length of quantity-sequence is 5 and the size of the array *QSRecords* is 3. When the new transaction comes, we process Procedure *InsertTransaction* as follows.

```
Procedure InsertTransaction (root, TNew);
begin
    foreach child TOld of root do
    begin
        if (TNew ∩ TOld = Ø)          /* case 1: empty*/
        begin
            create a new node for TNew;
        end;
        else if (TNew == TOld)        /* case 2: equivalence */
        begin
            break;
        end;
        else if (TNew ⊃ TOld)         /* case 3: superset */
        begin
            let TOld be TNew's child;
        end;
        else if (TNew ⊂ TOld)         /* case 4: subset */
        begin
            let TOld be TNew's parent;
        end;
        else if (TNew ∩ TOld ≠ Ø)     /* case 5: intersection */
        begin
            intersection := TNew ∩ TOld;
            if (TOld's descendant does not contain IntersectionX)
            begin
                create a new node for intersectionX;
                TNew link to intersectionX;
                TOld link to intersectionX;
                InsertTransaction (TNew, intersectionX);
                InsertTransaction (TOld, intersectionX);
            end;
        end;
    end;
end;
```

When the first transaction T_1 comes, the root does not have a child. So, a new node for the itemset {CDE} is created directly. The result of inserting transaction T_1 is shown in Figure 6-(a), where (0, 0, 0) will record the quantity-sequence of such an itemset in batch B_i, B_i+1, and B_i+2, respectively, later. When the second transaction T_2 comes, our algorithm will call Procedure *InsertTransaction* to check the set-relation among transaction T_2 and previous transactions. Transaction T_2 will process Case 1 (the empty relationship to {CDE}). Thus, a new node for itemset {AB} is created. The result of inserting transaction T_2 is shown in Figure 6-(b). When the third transaction T_3 comes, we will call Procedure *InsertTransaction* to check the set-relation among transaction T_3 and previous transactions. Transaction T_3 will process Case 2 (the equivalent relationship to {AB}). Moreover, our algorithm does not create a new node. The result of inserting transaction T_3 is shown in Figure 6-(c). When the fourth transaction T_4 comes, we will call Procedure *InsertTransaction* to check the set-relation among transaction T_4 and previous transactions. Transaction T_4 will pass through the conditions of Case 3 (the superset relationship of {AB}) and Case 5 (the intersection relationship with {CDE}). First, the itemset of transaction T_4 is the superset of itemset {AB}. Therefore, a new node for itemset {ABCD} is created and itemset {AB} becomes the child node of itemset {ABCD}. Second, itemset {ABCD} and itemset {CDE} have a common itemset {CD}. Next, we will call Function *FindLattice* to check whether the common itemset {CD} exists in the current lattice structure or not. Because the common itemset {CD} does not exist in the current lattice structure, our algorithm will create a new node {CD} to be the child node of itemset {ABCD} and itemset {CDE}. The result of inserting transaction T_4 is shown in Figure 6-(d).



TID	Quantity-Sequence
1	00232

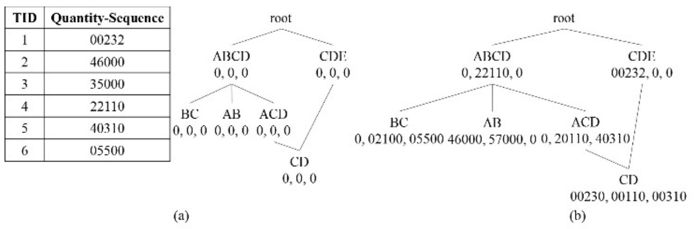
TID	Quantity-Sequence
1	00232
2	46000

TID	Quantity-Sequence
1	00232
2	46000
3	35000

TID	Quantity-Sequence
1	00232
2	46000
3	35000
4	22110

Figure 6. The result of inserting transaction and inserting the quantity-sequence of transaction into table *SWTable*: (a) T_1; (b) T_2: (c) T_3; (d) T_4.

All the intersection of all the transactions in this window W_1 are stored in each node of this lattice structure and the corresponding quantity-sequence of the current window is stored in the table *SWTable* as shown in Figure 7-(a). Moreover, we insert the quantity-sequence *QuantitySeq* of each transaction into our data structure and update its child nodes. We will perform the mining process to find high utility patterns. Therefore, our algorithm inserts the quantity-sequence of transaction T_1, 00232, which belongs to batch B_1 into this lattice structure. The quantity-sequence *QuantitySeq*, 00232, is updated in the corresponding position of the array *QSRecords* in the node. In the same time, itemset {CDE} has a child node, {CD}. Thus, our algorithm updates the array *QSRecords* of itemset {CD} into (00230, 0, 0) by the quantity-sequence of transaction T_1, 00230 since this node only contains item C and item D. That is, we record the quantity-sequence of transaction T_1 in the node {CDE} and all of its child nodes. Figure 7-(b) shows the window W_1 has been set up.



TID	Quantity-Sequence
1	00232
2	46000
3	35000
4	22110
5	40310
6	05500

Figure 7. (a) The result of inserting transaction T_1 - T_6 and inserting the quantity-sequence of transaction T1 - T_6 into table *SWTable*; (b) The result of inserting the quantity-sequence of transaction T_1 - T_6.

The minimum utility of the window W_1 is calculated from the table *CntTable* of the window *W*1 and the profit table. For example, in this case, *TotalTU* is 164 (=3 × 13 + 2 × 18 + 4 × 11 + 7 × 5 + 5 × 2). The threshold of the window W_1 is 36.08 (=164 × 22%). First, the count of each single item is stored in table *CntTable*. Therefore, we can calculate the utility value of each item to obtain the high utility patterns. Second, each node of the lattice has an array *QSRecords* which holds the information about the quantity of each item in the itemset. Therefore, we can calculate the utility value of each itemset in the lattice to obtain the high utility patterns. When the utility value of the itemset is greater than the threshold, we identify this itemset as a high utility pattern. Because the subset of each high utility patterns in the lattice may be the high utility pattern, we examine the high utility patterns, which its utility value is greater than the threshold and the size of this itemset is greater than 2. By the way, the size of the itemset is *k*. If an itemset satisfies the above two points, our algorithm will create all subsets of such an itemset, which its size is *k*-1, to calculate the utility value of each subset recursively; otherwise, the mining operation will stop. We use the above example to achieve the contents of the above description. First, we obtain the high utility patterns, {A: 39}, {C: 44}, from the table *CntTable*. Second, we have calculated the utility value of each itemset to obtain the high utility patterns, {CDE: 39}, {BC: 38}, {AB: 53}, {ACD: 48}, {CD: 59}. At this time, the itemset {CDE} and {ACD} satisfy the condition that the size of the itemset is greater than 2 and is a high utility pattern itself. Therefore, we will create the other subset of the itemset {CDE} and {ACD}, including {CE}, {DE}, {AC}, and {AD}, to check whether the itemsets is a high utility pattern or not. The result of the utility values of these four subsets are no greater than minimum utility. Thus, The utility values of these high utility patterns of the window W_1 are {A: 39}, {C: 44}, {CDE: 39} {BC: 38}, {AB: 53}, {ACD: 48}, {CD: 59}.

When a new batch information B_4 which contains transaction T_7 and transaction T_8, comes as shown in Figure 5-(a), our algorithm will perform the deletion process. First, the tables *CntTable* and *SWTable* of the window W_1 are updated by the information of the oldest batch B_1, T_1 and T_2. Second, our algorithm shifts the array *QSRecords* of all nodes to the left as shown in Figure 8-(a). For example, for itemset {AB}, the original array *QSRecords* is (46000, 57000, 0). After the shifting process, the array *QSRecords* becomes (57000, 0, 0). If the shifted array *QSRecords* becomes zero (*i.e.*, (0, 0, 0)), our algorithm removes this node from the lattice (*i.e.*, the node with itemset {CDE}). The result of deletion process as shown in Figure 8-(b). When the deletion process is finished, we will perform the insertion process continuously.

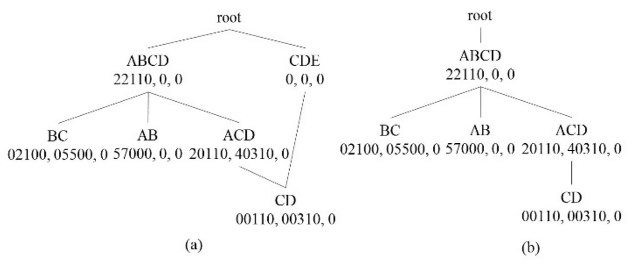
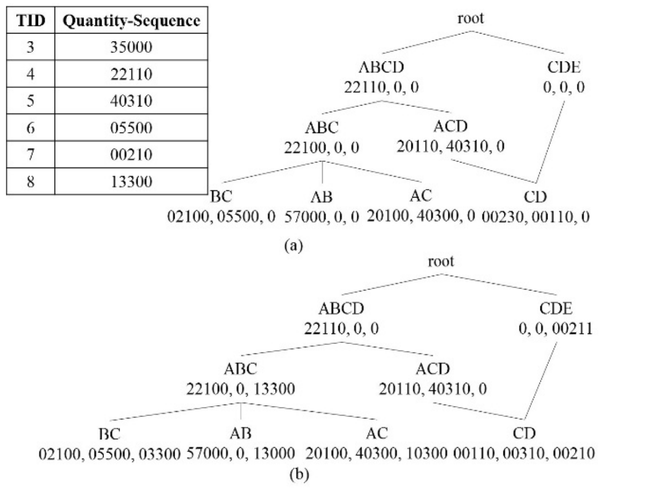


Figure 8. (a) The result of shifting the array *QSRecords* of all nodes to the left; (b) The result of deletion process.

When the seventh transaction T_7 comes, we will call Procedure *InsertTransaction* to check the set-relation among transaction T_7 and previous transactions. Transaction T_7 will pass through the conditions of Case 5 and Case 3. When the eighth transaction T_8 comes, we will call Procedure *InsertTransaction* to check the set-relation among transaction T_8 and previous transactions. Transaction T_8 will pass through the conditions of Case 4, Case 5, and Case 3. The result of inserting transaction T_7 and T_8 is shown in Figure 9-(a). All the intersection of all the transactions in this window W_2 are stored in each node of this lattice structure. Then, we insert the quantity-sequence *QuantitySeq* of each transaction into our data structure and update all of its child nodes. In the same way, we perform our insertion process again. Therefore, our algorithm inserts the quantity-sequence of transaction T_7, 00211, which belongs to batch B_4 into this lattice structure. The quantity-sequence *QuantitySeq*, 00211, is updated in the corresponding position of the array *QSRecords* in the node. In the same time, itemset {CDE} has a child node, {CD}. Thus, our algorithm updates the array *QSRecords* of itemset {CD} into (00110, 00310, 00210) by the quantity-sequence of transaction T_7, 00210 since this node only contains item C and item D. Next, we insert the quantity-sequence of transaction T_8, 13300, which belongs to batch *B*4 into this lattice structure. The quantity-sequence *QuantitySeq*, 13300, is updated in the corresponding position of the array *QSRecords* in the node. The result of inserting the quantity-sequence of transaction T_7 and T_8 is shown in Figure 9-(b).



TID	Quantity-Sequence
3	35000
4	22110
5	40310
6	05500
7	00210
8	13300

Figure 9. (a) The result of inserting transaction T_7 and T_8 and inserting the quantity-sequence of transaction T_7 and T_8 into table *SWTable*; (b) The result of inserting the quantity-sequence of transaction T_7 and T_8.

IV. Performance

In this section, we first present the performance model. Then, we present experiments result.

A. *Performance Model*

In this subsection, we will compare the performance between the *Subset-Lattice* algorithm and the *SHU-Grow* algorithm for the synthetic database. And, in the sliding window model, algorithms use two parameters, the size of the window and the size of the batch. Therefore, we evaluate mining performance of the both algorithms in terms of the processing time under the change of the size of the batch and the change of the size of the window. Besides, we also evaluate mining performance of the both algorithms in terms of the processing time under the change of the minimum utility threshold. The profit of each item is generated between 1 and 5, and the count of each item is generated between 1 and 10. The parameters used in the generation of the data are shown in Table I [2]. For example, we set the size of the window as 3 and the size of the batch as 2. Therefore, each window contains 2 × 3 = 6 transactions.

TABLE I. The details of parameters used in the experiments

Parameters	**Meaning**
T_{avg}	The average size of the transactions in the dataset
|I|	The number of the items in the dataset
|D|	The number of the transactions in the dataset
Threashold	The minimum utility threshold
BatchSize	The number of the transactions deleted and inserted while the window slides
WindowSize	The number of the batches in the window

B. *Experiments Results*

In this subsection, we will compare the performance between *Subset-Lattice* algorithm and *SHU-Grow* algorithm. We compare the processing time and the number of nodes of the synthetic database. We set the size of the window as 3, the size of batch as 2, and the minimum utility threshold as 22%.

In Figure 10-(a), we show the comparison of the processing time of both algorithms for the synthetic dataset on T(5-9).I60.D1K under the change of average size of transactions. We observe that the processing time of the *SHU-Grow* algorithm increases, when the average size of the transactions increases. Because the number of the candidate patterns increases in each window as the average size of the transactions increases. However, the *Subset-Lattice* algorithm obtains the candidate patterns easily. Therefore, the *Subset-Lattice* algorithm will take the shorter time to find the high utility patterns than the *SHU-Grow* algorithm.

In Figure 10-(b), we show the comparison of the processing time of both algorithms for the synthetic dataset on T5.I(20-100).D1K under the change of the number of items. We observe that the processing time of the *Subset-Lattice* algorithm increases, when the number of items decreases. Therefore, the number of transactions with the intersection relationship increases in the *Subset-Lattice* algorithm as the number of items decreases. Because the *Subset-Lattice* algorithm obtains the candidate patterns easily. Therefore, the *Subset-Lattice* algorithm will take the shorter time to find the high utility patterns than the *SHU-Grow* algorithm.

In Figure 10-(c), we show the comparison of the processing time of both algorithms for the synthetic dataset on T8.I60.D(1-15)K under the change of the size of dataset. We observe that the processing time of the *SHU-Grow* algorithm and the *Subset-Lattice* algorithm increases, when the size of a dataset increases. The result shows that the number of the candidate patterns increases in each window as the number of transactions increases. Because the *Subset-Lattice* algorithm obtains the candidate patterns easily. Therefore, the *Subset-Lattice* algorithm will take the shorter time to find the high utility patterns than the *SHU-Grow* algorithm.

In Figure 10-(d), we show the comparison of the number of nodes of both algorithms for the synthetic dataset on T5.I(20-100).D1K under the change of the number of items. Due to that the *SHU-Grow* algorithm performs the mining process, it generates other tree structure, which keeps the information. Therefore, the *SHU-Grow* algorithm requires more number of nodes to construct their structure. We observe that the number of nodes of the *Subset-Lattice* algorithm increases, when the number of items decreases. The number of transactions with the intersection relationship increases in the *Subset-Lattice* algorithm as the number of items decreases. Therefore, the number of nodes of the *Subset-Lattice* algorithm will increases. However, the *Subset-Lattice* algorithm uses fewer nodes than the *SHU-Grow* algorithm.

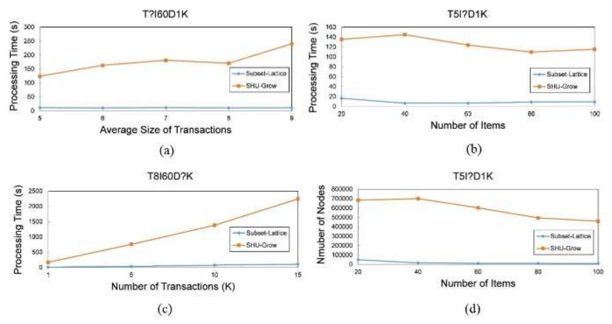


Figure 10. A comparison of the processing time of the synthetic dataset: (a) T(5-9).I60.D1K, under the change of the average size of transactions; (b) T5.I(20-100).D1K, under the change of the number of items; (c) T8.I60.D(1-15)K, under the change of the size of dataset; (d) T5.I(20-100).D1K, under the change of the number of items.

V. Conclusion

In this paper, we have proposed the *Subset-Lattice* algorithm which can mine the high utility patterns by the actual contribution value. In our algorithm, we use the lattice structure to keep the information of the transactions in each window. Moreover, our proposed algorithm uses an array, *QSRecords*, to store the processed transaction data, *QuantitySeq*. From our performance results, the *Subset-Lattice* algorithm has better performance than the *SHU-Grow* algorithm in the synthetic data.

VI. Acknowledgments

This research was supported in part by the Ministry of Science and Technology of Republic of China under Grant No. MOST-106-2221-E-110-079.

References

[1] C. F. Ahmed, S. K. Tanbeer, B. S. Jeong, and H. J. Choi, “Interactive Mining of High Utility Patterns over Data Streams,” Expert Systems with Applications, vol. 39, no. 15, pp. 11979–11991, Nov. 2012.

[2] H. Ryang and U. Yun, “High Utility Pattern Mining over Data Streams with Sliding Window Technique,” Expert Systems with Applications, vol. 57, pp. 215– 231, Sept. 2016.

[3] B. E. Shie, P. S. Yu, and V. S. Tseng, “Efficient Algorithms for Mining Maximal High Utility Itemsets from Data Streams with Different Models,” Expert Systems with Applications, vol. 39, no. 17, pp. 12947–12960, Dec. 2012.

[4] R. Agrawal and R. Srikant, “Fast Algorithms for Mining Association Rules,” Proc. of the 20th Int. Conf. on VLDB, pp. 490–501, 1994.M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

[5] H. Chen, L. C. Shu, J. L. Xia, and Q. S. Deng, “Mining Frequent Patterns in a Varying-Size Sliding Window of Online Transactional Data Streams,” Information Sciences, vol. 215, pp. 15–36, Dec. 2012.

[6] M. Deypir, M. H. Sadreddini, and M. Tarahomi, “An Efficient Sliding Window Based Algorithm for Adaptive Frequent Itemset Mining over Data Streams,” Journal of Information Science and Eng., pp. 1001–1020.

[7] G. Grahne and J. F. Zhu, “Fast Algorithms for Frequent Itemset Mining Using FP-Trees,” IEEE Trans. on Knowledge and Data Eng., vol. 17, no. 10, pp. 1347– 1362, Oct. 2005.

[8] J. L. Koh and S. N. Shin, “An Approximate Approach for Mining Recently Frequent Itemset from Data Streams,” Computer Science Data Warehousing and Knowledge Discovery, vol. 4081, no. 1, pp. 352–362, Spet. 2006.

[9] H. F. Li and H. Chen, “Mining Non-Derivable Frequent Itemsets over Data Stream,” Data and Knowledge Eng., vol. 68, No. 5, pp. 481–498, May 2009.

[10] H. F. Li and S. Y. Lee, “Mining Frequent Itemsets over Data Streams Using Efficient Window Sliding Techniques,” Expert Systems with Applications, vol. 36, no. 2, pp. 1466–1477, March 2009.

[11] C. H. Lin, D. Y. Chiu, Y. H. Wu, and A. L. P. Chen, “Mining Frequent Itemsets from Data Streams with a Time-Sensitive Sliding Window,” Proc. of the SIAM Int. Conf. on Data Mining, pp. 68–79, 2005.

[12] S. K. Tanbeer, C. F. Ahmed, B. S. Jeong, and Y. K. Lee, “Sliding Window-Based Frequent Pattern Mining over Data Streams,” Information Sciences, vol. 179, no. 22, pp. 3843–3865, Nov. 2009.

[13] J. C. W. Lin, W. Gan, P. Fournier-Viger, H. C. Chao, and T. P. Hong, “Efficiently mining frequent itemsets with weight and recency constraints”, Appl. Intell., vol. 47, no.3, pp. 769-792, Oct. 2017.

[14] Y. I. Chang, M. H. Tsai, C. E. Li, and P. Y. Lin, “A Set-Checking Algorithm for Mining Maximal Frequent Itemsets from Data Streams,” Intelligent Technologies and Eng. Systems, vol. 20, no. 2, pp. 51–63, April 2013.

[15] C. S. Hemalatha, V. Vaidehi, and R. Lakshmi, “Minimal Infrequent Pattern Based Approach for Mining Outliers in Data Streams,” Expert Systems with Applications, vol. 42, no. 4, pp. 1998–2012, March 2015.