SSUP-Growth: A Novel Mining High Utility Algorithm Itemset with Single-Scan of Database

Naji Alhusaini, Li Jing, Ammar Hawbani and Abdullah Alhusaini
University of Science and Technology of China Hefei, China
School of Management, Hefei University of Technology Hefei, China
E-mail: naji@mail.ustc.edu.cn

Abstract. High Utility Itemset Mining (HUIM) alludes to the identification of itemsets of high utility in the value-based database UP-Growth algorithm is a standout amongst the best algorithms for overcome the challenge of candidate generation and scan database reputedly of previous algorithms. However, it needs scan database twice to actualize the UP tree. Regarding of the updating existing data with new information, UP-growth needs for twofold scanning of new information and existing information. The fundamental motivation behind this work is to build up another algorithm, Single-Scan Utility Pattern Tree (SSUP-tree), for mining high utility itemsets from transaction database through only single-scan of database. In our algorithm, the details of high-utility itemsets is preserved in a particular data structure of the SSUP-Tree after a single-scan of database. Consequently, it can retrieve the identical UP-tree with a fixed minimum utility threshold. The proposed algorithm required to scan the new data only to update SSUP-tree. In this regard, in order to estimate the execution of the proposed algorithm, the SSUP-tree algorithm has been implemented on synthetic and real datasets. The results of this study revealed that SSUP-tree shows a significant enhancement in the execution in terms of runtime since it keeps the huge databases details in a compact format and it avoids repetition of database scanning.

1. Introduction
A framework of HUIM has emerged to address the limitation of Frequent Itemset Mining (FIM), which only considers the frequency of the items without regard to the utility of items such as profit and quantity. HUIM has many scientific and commercial applications and is one of the topics of recent active research [1] [2] [3] [4] [5]. The importance of HUIM framework is to provide the decision makers with greater flexibility to reflect the utility of items, such as profit and quantity. This allows more interesting and feasible patterns to be generated In HUIM framework, each item has two values to determine its utility, external utility which represents the significance of the items that given separately by the users and the internal utility which represents the significance of the item in the transaction. Several methods have been applied in previous research [2, 4, 5, 6, 8] to enhance the performance of utility mining. Arguably, Regarding the method used, the HUIM framework approaches can be further classified as follows: level-wise approaches [1, 6, 7], Tree-based (pattern-growth) approaches [8, 9, 10, 11, 2, 12], Projection-based approaches [4, 13], and utility list based methods [3, 14, 5, 15].

Tree-based methods such as Utility Pattern Growth (UP-Growth) [11] seeks to overcome dilemma of the candidates generation. Nonetheless, this algorithm still needs to scan the
database double time. Additionally, while the database has been updated, UP-Growth [11] needs to re-implement the whole process for new and old database together once more to update the rules. In other words, it ignores the past double scan of database and start the whole process again. In this paper, we propose a new method, SSUP-Growth, to generate High Utility Itemset with only single-scan of database. In case of updating the database, our proposed method needs a single-scan of the new data only.

Based on the literature aforementioned and the best knowledge of the authors, UP-growth approach is one of the best approaches that avoids the costly candidates generation. However, it still needs to scan database double time. Furthermore, it handles the update of the database by rescan the old and the new data double time again. This motives the current work to create a novel algorithm that needs single-scan of database and handles the update of database by single-scan of the new data only.

The rest of this paper is structured in such a way: in Section 2, we present utility itemset mining related work. In Section 3, we explain the relative definitions and the problem statement. The proposed algorithm and the data structure are comprehensively introduced in Section 4. Finally, section 5 concludes this work.

2. Preliminaries and Problem Statement

2.1. Problem Definition

Let \( I = \{ y_1, y_2, ..., y_m \} \) represent a limited set of distinct items, and \( D = \{ T_1, T_2, ..., T_n \} \) depict a set of transactions, where each transaction \( T_i (1 \leq i \leq n) \) portrays a subset of \( I \) and has a distinctive identifier \( i \) known as Tid. Any subset of \( I \) includes \( K \) items \( \{ y_1, y_2, ..., y_k \} \) where \( y_j \in I, 1 \leq j \leq k \) is named as the \( k \)-itemset. Table 1 provides an example transaction database \( D \), and Table 2 lists the items of external profit details.

**Table 1. An example database**

| TID | Transaction | TU |
|-----|-------------|----|
| 1   | (A,1) (C,1) (D,1) | 8  |
| 2   | (A,2) (C,6) (E,2) (G,5) | 27 |
| 3   | (A,1) (B,2) (C,1) (D,6) (E,1) (F,5) | 30 |
| 4   | (B,4) (C,3) (D,3) (E,1) | 20 |
| 5   | (B,2) (C,2) (E,1) (G,2) | 11 |

**Table 2. Items Profit table**

| Item | A | B | C | D | E | F | G |
|------|---|---|---|---|---|---|---|
| Profit | 5 | 2 | 1 | 2 | 3 | 1 | 1 |

**Definition 1.** Item internal utility refers to the significance of item \( y_j \) in the transaction \( T_i \) such as quantity provided in the transaction and denoted as \( Q(y_j, T_i) \). For example in Table 2, \( Q(A, T_2) = 2 \).

**Definition 2.** Item external utility presents the prominence of item \( y_i \) provided in the utility table, such as profit. For example, in Table 2, \( P(A) = 5 \).

**Definition 3.** The utility of an item \( y_j \in T_i \) remains distinct as \( U(y_j, T_i) \) is calculated as the product of the internal and external profit of items in the transaction, \( U(y_j, T_i) = P(y_j) * Q(y_j, T_i) \).

For example, in Table 1, \( U(D, T_1) = 2 * 1 = 2 \).

**Definition 4.** The utility of a set \( Y \) in \( T_i \) is denoted by \( U(Y, T_i) \) and defined by, \( \sum_{Y_i \in Y \land Y_i \subseteq T_i} U(Y_i, T_i) \).

For example, in Table 1, \( U(AC, T_1) = U(A, T_1) + U(C, T_2) = 5 + 1 = 6 \).

**Definition 5.** The utility of the itemset \( Y \) in \( D \) is signified by \( U(Y) \) and defined by, \( \sum_{T_i \in D} \sum_{Y_i \in Y \land Y_i \subseteq T_i} U(y_i, T_i) \).

For example, in Table 1, \( U(AC) = U(AC, T_1) + U(AC, T_2) + U(AC, T_3) = 6 + 16 + 7 = 29 \).
Definition 6. The user-specified minimum utility percentage value is denoted by $\beta$ while the absolute minimum utility value, signified by $(\text{min}_\text{util})$, is calculated as $\text{min}_\text{util} = \beta \times \sum_{T_i \in D} \text{TU}(T_i)$

For the illustrative example in Table 1, when $\beta = 20\%$, the absolute minimum utility value is $\text{min}_\text{util} = 0.20 \times 71 = 42$.

Definition 7. The HUI depicts the itemset with the utility value greater than or equal to the user-defined minimum utility threshold $(\text{min}_\text{util})$. Otherwise, the itemset reveals an LUI.

Definition 8. The transaction utility of a transaction $T_i$ is portrayed by $\text{TU}(T_i)$ and specifies the total utility of all items it contains. It is defined by, $\text{TU}(T_i) = \sum_{Y \subseteq T_i \land y_i \in Y} \text{U}(y_i, T_i)$

For example, in Table 1, $\text{TU}(T_5) = U(B, T_5) + U(C, T_5) + U(E, T_5) = 4 + 2 + 3 = 9$.

Definition 9. The transaction-weighted utilization of the itemset $Y$ represents the sum of the transaction utilities of all the transactions including $Y$, indicated as $\text{TWU}(Y)$ and displays as $\text{TWU}(Y) = \sum_{Y \subseteq T_i \land T_i \in D} \text{TU}(T_i)$

For example, $\text{TWU}(AC) = \text{TU}(T_1) + \text{TU}(T_2) + \text{TU}(T_3) = 8 + 27 + 30 = 65$.

If $\text{TWU}(Y)$ is not smaller than the minimum utility threshold, $Y$ is considered as a high transaction-weighted utilization itemset (HTWUI).

Definition 10. The transaction-weighted downward closure, ($\text{TWDC}$ for short), is declared as follows: For any itemset $Y$, if $Y$ does not equal an HTWUI, any superset of $Y$ is not HTWUI, too. By this definition, we preserve the downward closure by employing transaction-weighted utilization. For example, in Table 1, since $\text{TWU}(AC) > \text{min}_\text{util}$, any superset of AC depicts a high utility itemset.

3. Proposed Approach

3.1. The Components of SSUP-Tree

In SSUP-Tree, the root node is labelled as Null. Each other node $N$ contains five components, $N$.parent, $N$.name, $N$.nu, $N$.count, and a set of children nodes, where $N$.parent designates the parent of $N$, $N$.name denotes the original item in the transaction database, $N$.nu refers to the utility value, and $N$.count represents the support count. In SSUP-Tree, each path represents one or more transaction(s), and its corresponding utility(s) is considered as the utility of its least high utility item(s). The utility of a given node is greater than the utility of its children or descendants.

A prefix in SSUP-tree represents the common identical patterns appeared in several transactions.

Instead of the Header Table in UP-Growth, in our proposed method there is an Item Utility List (IUL) to record the actual utility value for each item. IUL contains the names of items and their estimated utility values. Moreover, there is a High Utility Item List (HUIL), which includes the items with high utility only.

3.2. Building the SSUP-Tree

Our proposed approach constructs a compressed tree structure by taking advantage of the best usage of the appearance of the common prefixes in the transactions and the similarity of the transactions. As mention above, the primary SSUP-tree contains all of the items occur in the database. The SSUP-tree is built in two steps (see Algorithm 1):

- In the initial (and only) scan, the transactions will be inserted based on predefined item order (such as lexicographical order) one by one into the empty SSUP-Tree. Meanwhile, recording the actual utility value $\text{TWU}$ of each item into the IUL.
- IUL is ascendingly sorted according to the items utility. Next, the nodes in each path are descendingly resorted according to their utility values recorded in IUL. Afterward, the sorted paths are inserted into a new tree (NSSUP-tree).
3.3. Achieving the UP-tree from the SSUP-tree
Based on the characteristics of the SSUP-tree, it is worth to mention that the SSUP-tree is included the high and low utility items. Thus, the UP-tree is a sub-tree of the SSUP-tree with a fixed \((min\_util)\) threshold, which involves the high utility items that meet the \((min\_util)\) threshold. In this regard, the mechanism of the UP-tree generation from SSUP-tree is proposed and analyzed in this section.

After constructing the primary SSUP-tree, the High Utility Items list (HUIL) is simply obtained according to a given specific \((min\_util)\). All we need to do is discard the unpromised (low utility) items from IUL. Afterward, we examine the utility of each node in the paths, from the root until the leaves, in order to trim the primary SSUP-tree by eliminating the unpromised nodes. Since the children or its descendant have less utilities than their parents, the unpromised nodes will be eliminated with its subtree.

3.4. Update the SSUP tree with new data
Since SSUP-tree contains all the items in the database. Therefore, SSUP tree can be updated by scanning new data without additionally scanning old data. The update process runs in two steps. First, the new transactions are inserted into the primary SSUP-tree based on IUL, and meanwhile the IUL is updated. Second, the updated SSUP-tree is rebuilt based on the updated IUL. In the case of a new transaction containing new items, which do not appear in the existing database, it can be handled as a new branch of the main SSUP tree with its actual utility values.

4. Experimental Evaluation
4.1. Settings
This section shows our experiments results after examining the execution of our algorithm and comparing it to the UP-Growth algorithm [11]. The experiments have been executed on a 2.66 GHz Intel Core 4 Quad Processor with 4G memory, on operation system Windows 7. The code of the algorithms are developed in Java. Furthermore, the algorithms performance is estimated by utilizing synthetic and real datasets. Moreover, T10I6D100K and chess datasets published in [16] have been considered in this work. Both of the datasets used in this study already contained unit profits and purchase quantities. Table 3 shows the properties of the datasets.

| Dataset      | Items | Avg. Len | Transaction |
|--------------|-------|----------|-------------|
| T10I6D100K   | 1,000 | 10       | 100,000     |
| Chess        | 76    | 37       | 3,196       |

4.2. Evaluation on Synthetic and Real Datasets
First, we use the synthetic dataset T10I6D100K to show the evaluation results. We employed five minimum utility thresholds, 40%, 50%, 60%, 70%, and 80%. The evaluation results of the phase I (i.e., tree construction) on the dataset T10I6D100K are plotted in Figure 1 the results showed that our algorithm takes less execution time as it needs single-scan of dataset. On the other hand, the execution time of phase II (i.e., get the corresponding UP-tree) is shown in Figure 2. At the beginning, both algorithms generate the corresponding UP-tree for the first minimum utility threshold (40%). After that, when the minimum utility threshold increases from 40% to 50% or 70%, the corresponding UP-tree can easily obtained through trimming the UP-tree with the minimum utility threshold of 40% with no need to scan the dataset again. However, when the minimum utility threshold has diminished, UP-Growth algorithm ignored the past double
scan of dataset and re-scanned dataset double time again that requires more time. As shown in Figure 2, when the \((\text{min} \_\text{util})\) threshold decreased from 80% to 70%, UP-Growth spends more time to get the corresponding UP-tree. On the other hand, our algorithm gets the corresponding UP-tree by trimming the primary SSUP-tree, regardless how the minimum utility thresholds are modified, without having to re-scan the dataset. Second, we evaluated the execution time of UP-growth and SSUP-Tree for Phase I and Phase II on the chess dataset. Figure 3 and Figure 4 show the runtime of both algorithms for Phase I and Phase II on the dataset chess using different minimum utility thresholds. It is clear that our proposed algorithm spends less time in each phase for the same reason explain in the first experiment.

4.3. Scalability

In this subsection, we provide experiments of evaluate scalability of our proposed algorithm. The dataset size of the T10I6 dataset has been varied to evaluate the scalability of UP-Tree and SSUP-tree algorithms. Figure 5 and 6 show the performance of both methods on different size of dataset and with \((\text{min} \_\text{util})\) is 85%. As shown in Figure 5 and 6, the execution time of the SSUP-Tree is less than the UP-tree. In case of the database is size increased, the execution time for distinguish high-utility itemsets also increases. Consequently, the UP-Growth algorithm requires more processing time than the SSUP-tree.
5. Conclusions
In this paper, we present an algorithm called SSUP-tree for mining efficient itemsets from transactional databases. In addition, we discussed how the SSUP-tree is obtained by a single scan of the database and how it is updated by a single scan of the new data. In order to evaluate the runtime of the proposed algorithm, SSUP-tree algorithm and UP-growth implemented on real data sets and synthetic datasets. The experimental results show that our proposed algorithm, SSUP-tree, continuously exceeds the UP-tree algorithm in the term of runtime because SSUP-tree preserves the information of items in a compact structure and avoids scanning the database again, especially when the database contains a large number of transactions with shared prefixes. However, the limitation of the proposed method is that it based on a single minimum threshold, which has an adverse effect on the mining results. Therefore, the future work is to improve this approach to mine high utility itemsets based on multiple minimum thresholds.

Acknowledgement
This paper is supported by the “Strategic Priority Research Program of Chinese Academy of Sciences (A Class) NO. XDA19020102 ”.

References
[1] Ying Liu, Wei-keng Liao, and Alok Choudhary. A two-phase algorithm for fast discovery of high utility itemsets. In Pacific-Asia Conference on Knowledge Discovery and Data Mining, pages 689–695. Springer, 2005.
[2] Vincent S Tseng, Bai-En Shie, Cheng-Wei Wu, and S Yu Philip. Efficient algorithms for mining high utility itemsets from transactional databases. IEEE transactions on knowledge and data engineering, 25(8):1772–1786, 2013.
[3] Philippe Fournier-Viger, Cheng-Wei Wu, Souleymane Zida, and Vincent S Tseng. Fhm: Faster high-utility itemset mining using estimated utility co-occurrence pruning. In International symposium on methodologies for intelligent systems, pages 83–92. Springer, 2014.
[4] Guo-Cheng Lan, Tzung-Pei Hong, and Vincent S Tseng. An efficient projection-based indexing approach for mining high utility itemsets. Knowledge and information systems, 38(1):85–107, 2014.
[5] Souleymane Zida, Philippe Fournier-Viger, Jerry Chun-Wei Lin, Cheng-Wei Wu, and Vincent S Tseng. Efim: A fast and memory efficient algorithm for high-utility itemset mining. Knowledge and Information Systems, 51(2):595–625, 2017.
[6] Hong Yao, Howard J Hamilton, and Cory J Butz. A foundational approach to mining itemset utilities from databases. In Proceedings of the 2004 SIAM International Conference on Data Mining, pages 482–486. SIAM, 2004.
[7] Yu-Chiang Li, Jieh-Shan Yeh, and Chin-Chen Chang. Isolated items discarding strategy for discovering high utility itemsets. Data & Knowledge Engineering, 64(1):198–217, 2008.
[8] Chowdhury Farhan Ahmed, Syed Khairuzzaman Tanbeer, Byeong-Soo Jeong, and Young-Koo Lee. Efficient tree structures for high utility pattern mining in incremental databases. IEEE Transactions on Knowledge and Data Engineering, 21(12):1708–1721, 2009.
[9] Jianying Hu and Aleksandra Mojsilovic. High-utility pattern mining: A method for discovery of high-utility item sets. Pattern Recognition, 40(11):3317–3324, 2007.
[10] Chowdhury Farhan Ahmed, Syed Khairuzzaman Tanbeer, Byeong-Soo Jeong, and Young-Koo Lee. Huc-prune: an efficient candidate pruning technique to mine high utility patterns. Applied Intelligence, 34(2):181–198, 2011.
[11] Vincent S Tseng, Cheng-Wei Wu, Bai-En Shie, and Philip S Yu. Up-growth: an efficient algorithm for high utility itemset mining. In Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 253–262. ACM, 2010.
[12] Wei Song, Yu Liu, and Jinhong Li. Mining high utility itemsets by dynamically pruning the tree structure. Applied intelligence, 40(1):29–43, 2014.
[13] Alva Erwin, Raj P Gopalan, and NR Achuthan. Efficient mining of high utility itemsets from large databases. In Pacific-Asia Conference on Knowledge Discovery and Data Mining, pages 554–561. Springer, 2008.
[14] Junqiang Liu, Ke Wang, and Benjamin CM Fung. Direct discovery of high utility itemsets without candidate generation. In 2012 IEEE 12th international conference on data mining, pages 984–989. IEEE, 2012.
[15] Heungmo Ryang and Unil Yun. Indexed list-based high utility pattern mining with utility upper-bound reduction and pattern combination techniques. Knowledge and Information Systems, 51(2):627–659, 2017.
Appendix: Algorithms Pseudocode

ALGORITHM 1: SSUP-Tree Generation
Input: (1) A transaction database DB (2) A minimum utility threshold min_util
Output: (1) A Single-Scan Utility Pattern tree (SSUP-tree) (2) Item utility list (IUL)

1. Create SSUP-tree with “null” root node
2. Create empty Item Utility List IUL
3. FOREACH transaction T in the transaction database DB
   1. Insert (T[i], SSUP) into utility list of the first item in T
   2. Update utility list with items in T[i]
ENDFOR

FOREACH SSUP’s child node N
   IF T[j] has children nodes N such that T[j] == N
      THEN
         N = utility - N. utility + T[j]. utility
      ELSE
         Insert (T[j], N)
      ENDIF
RETURN

ALGORITHM 2: Achieve UP-tree from the SSUP-tree
Input: An SSUP-tree, the item utility list, min_util
Output: An UP-tree

1. Create Initial Item Utility List (IUL) set as empty
2. FOREACH item x in IUL
   IF x utility >= min_util
      Insert x in IUL
   ENDIF

ENDFOR

3. Sort IUL in utility descending order
4. Call Test(T). The description of function test as follows.
   Function Test(T)
   FOREACH child node ci of the node N
      IF ci in IUL
         THEN
            Test(ci)
         ELSE
            ci (and the potential subtree beginning from ci)
         ENDIF
   ENDIF
   Return

ALGORITHM 3: Update SSUP-Tree
Input: (1) A new database DB (2) Primary SSUP-Tree (3) Item Utility List IUL
Output: (1) An Updated One Non Utility Pattern tree (SSUP-Tree) (2) An Updated Item utility list IUL

Procedure 1: Update the primary SSUP-tree with new data as well as update IUL
   FOREACH transaction T in DB
      1. Insert (T[i], SSUP)
      2. Update IUL with items in T[i]
ENDFOR

Procedure 2: Restructure the initial SSUP-tree
Create new SSUP with “null” root node of tree
FOREACH path p from the root until a leaf in the primary SSUP-tree SSUP
   IF p is not empty then
      1. Sort p according to IUL
      2. Call function (p[i], NSSUP) to insert the sorted p[i] into the new SSUP-tree.
ENDIF

ENDF