Method for Mining High-utility Patterns in Transaction Stream Data based on Linked List Structure

Minh-Thai Tran[1], Anh-Duy Tran[1], Duc-Thanh Pham[1], Minh-Nguyen Le[1]

[1] Faculty of Information Technology, Ho Chi Minh City University of Foreign Languages - Information Technology, Vietnam

Correspondence: Minh-Thai Tran, thaitm@huflit.edu.vn,
Communication: received 15 Dec 2023, revised 18 Apr 2024, accepted 23 May 2024
Digital Object Identifier: 10.32913/mic-ict-research.v2024.n2.1249

Abstract: Mining valuable patterns in data streams presents a significant challenge in the field of data mining. This task is crucial as it allows for the identification of highly profitable item sets within transaction databases. However, as new transactions are continually added, new valuable patterns emerge, thus changing the usefulness of previously analyzed data. It is essential to promptly update information regarding these changes to enable effective business decision-making. Consequently, existing mining methods applied to transaction flow datasets require considerable time to identify new patterns and update information related to new transactions. This article focuses on the research and proposal of a new transaction stream data mining method called High-Utility Stream Linked-List Mining. The method utilizes a linked list structure known as the High-Utility Stream Linked List (HUSLL) to store information about patterns in the database. Mining and updating transaction information are directly performed on the HUSLL structure. Experimental results demonstrate that this novel mining method exhibits more efficient execution times compared to previous solutions.

Keywords: *transaction stream data, data mining, high-utility item set.*

I. INTRODUCTION

Mining association rules [1] is one of the important topics in the field of data mining. Typically, association rules in mining are bifurcated into two stages: frequent pattern mining, also known as frequent item sets, followed by the generation of rules from these frequent patterns. However, some patterns, although not frequent, carry a higher value, such as instances where users purchase a few items with each item holding greater value than the remaining items. Consequently, frequent pattern mining encounters limitations when not considering both the number of items (internal utility value) and the value of the items (external utility value) purchased. To address these limitations, high-utility pattern mining is introduced, which considers both internal and external utility values to determine high-utility patterns within the transaction database [2]. Various high-utility pattern mining algorithms are proposed, including HMiner [3], EFIM [4], MAHI [5] and UBP-Miner [6].

However, existing algorithms are designed for static datasets, where data remains unchanged over time. Yet, the utility of a pattern or set of items may vary over time, such as beach tourism-related items being highly useful in the summer but rarely purchased in the winter. Consequently, methods for mining high-utility patterns over time are proposed to extract useful patterns in continuously updated databases. These methods include mining high-utility patterns on growing databases, such as HUI-list-INS algorithms [7], EIHI [8], LIHUP [9], and others. Additionally, transactional data stream mining methods, such as MHUI-BIT [10], MHUI-TID [10], and HUPMS [11] are introduced.

The current limitation of transaction stream data mining methods lies in the generation of considerable intermediate data when using prefix tree or condition tree structures for mining. Furthermore, during data update operations, specifically sliding window movement operations, updating the state of intermediate data structures incurs significant time and memory costs. This article proposes a list structure called High-Utility Stream Linked List (HUSLL), designed for efficient storage of data related to highly useful patterns in the database, enabling quick updates whenever a sliding window shift occurs. Specifically, the article introduces the High-Utility Stream Linked-List Mining (HSLM) algorithm, exploiting the HUSLL structure to detect high-utility patterns in the transaction stream database.

By using a list structure, pattern mining can be ap-

proached differently than with the HUPMS algorithm. The list structure can provide more flexible mining methods, optimizing the process of extracting pattern information. This can ensure that data is efficiently and accurately mined. Furthermore, with the list structure, updating the pattern information storage structure when there is a window shift can be optimized. When the sliding window moves, the list can be updated flexibly and efficiently. This helps save time and resources in updating the pattern information. Additionally, using a list structure can also change the way information is stored for parallel mining operations. The list structure can enhance the efficiency of parallel mining processes by partitioning and managing pattern information better.

The remainder of the article is structured as follows. Section II presents related research, followed by concepts and definitions in Section III. Section IV details the proposed method, while Section V showcases experimental results. Finally, conclusions and future research directions are discussed in Section VI.

II. RELATED WORKS

Frequent pattern mining has received significant attention and research since the introduction of the Apriori algorithm by Agrawal and Srikant in 1994 [1]. This algorithm utilizes the Apriori property, which states that "All nonempty subsets of a frequent itemset are frequent." In the pattern generation process, if an infrequent pattern emerges, there is no need to generate a parent pattern for that candidate, allowing for the early elimination of candidates to reduce space and exclude non-frequent patterns. However, a notable drawback is the algorithm's tendency to generate a large number of candidates, requiring multiple database scans to verify pattern popularity and thus increasing the overall mining process cost. In 2000, Han et al. proposed the FP-Growth algorithm [12] to address this limitation by exploiting frequent patterns without generating candidate patterns. This algorithm utilizes a tree structure called FP-Tree to store pattern information and employs a divide-and-conquer strategy for frequent pattern generation. Initially, the algorithm scans the database to identify sets with a length of 1 (1-itemset). Subsequently, the FP-Tree structure is constructed to identify frequent patterns within the 1-item set. While finding frequent patterns in trees incurs lower costs, the time required to generate FP-Trees may increase with larger databases. Following this, several other algorithms have been proposed to enhance the efficiency of frequent pattern mining, including LP-Growth [13], PMRARIM-IEG [14], and based on a linear table [15].

Frequent pattern mining algorithms are known for their focus on examining the occurrence frequency of patterns. However, many real-world problems require considering additional characteristics of the data, such as the internal utility value and external utility value of each item within a transaction. Frequent pattern mining algorithms do not meet this requirement. As a result, high-utility pattern mining algorithms, such as those proposed by [3], [4], [16], and [17], have been introduced. In 2005, Liu et al. [18] presented the Two-Phase algorithm for mining high-utility patterns, which builds upon the Apriori algorithm. To leverage the characteristics of the Apriori algorithm, Liu introduced the concept of transaction utility weights (TWUs) to estimate high-utility patterns based on utility weights. Currently, other mining methods such as FHM [16] and D2HUP [17] have been proposed for mining high-utility patterns using data structures like lists. The effectiveness of applying these list structures has been demonstrated on various experimental datasets.

Business analytics and decision support applications face the challenge of handling large and continuously growing volumes of data over time. A prime example is sales data, which is consistently updated with each customer purchase. While there exist high-utility pattern mining algorithms such as HMiner [3], EFIM [4], D2HUP [17], MAHI [5] and UBP-Miner [6]. these algorithms are designed to operate on static data. In other words, they do not accommodate data updates during the mining process, making them unsuitable for addressing the real-world mining needs of continuously evolving datasets.

To address the limitations of static data mining algorithms, several approaches have been proposed for incremental databases. In 2008, Yeh et al. [19] introduced two algorithms based on the Apriori approach: Incremental Utility Mining and Fast Incremental Utility Mining. These algorithms focus on generating candidates and testing to identify high-utility patterns. In 2009, Ahmed et al. [20] presented the IHUP algorithm, which utilizes a tree structure for mining high-utility patterns in incremental data. Several tree-based mining algorithms have since been proposed, such as iCHUM [21] and HUPID-Growth [22]. In 2015, Lin et al. [7] introduced a list-based mining algorithm named HUI-list-INS. Additionally, Fournier-Viger et al. [8] proposed EIHI, another list-based algorithm in the same year. In 2017, Yun et al. [9] presented the LIHUP algorithm, which stands out for its ability to mine high-utility patterns in a dynamic environment without the need to generate candidates.

Furthermore, research has extended to mining algorithms for incremental data and transaction stream data. Transaction stream data, characterized by continuous updates and item ordering over time, presents unique challenges. Items or sets of items may exhibit varying levels of utility

over different periods. Sliding window-based algorithms, such as MHUI-BIT [10] and MHUI-TID [10], have been proposed to handle the dynamic nature of continuously changing datasets. However, these algorithms are noted for generating a significant number of candidates, which poses a limitation.

Three main models are currently being used to mine high-utility itemsets in data streams. The first model is the landmark model: It mines all high-utility itemsets from the past to the present at a given time point from the data stream. The MAHUSP [23] algorithm that was proposed in 2017 is included among them. To store the necessary patterns, this algorithm utilizes a tree structure called MAS-Tree. MAHUSP remains limited in its memory capacity when adding new potential patterns to the tree. The second model is time decay models: This model is designed to find the latest relevant information from the data by distinguishing the importance of old and new transactions through a weight. In this model, GENHUI [24] and HAUPM [25] are two typical algorithms that are included. Lastly, the sliding window model: Separate the stream data into multiple batches, each batch having multiple transactions. The window will only keep a specific number of the latest batches in the stream data. Once there is new stream data, the data in the old batches will be excluded and the new transactions will be added to the window as the last batch. In this paper, we focus on the approach of mining high-utility itemsets from stream data using the sliding window model.

In 2010, Ahmed et al. [11] introduced a high-utility pattern mining algorithm designed for stream data known as HUP Mining over Stream Data (HUPMS). This algorithm utilizes a tree structure named HUS-Tree to store data stream information and conducts high-utility pattern mining based on this structure. Each time the sliding window changes, the algorithm updates the information of the nodes in the tree. The HUPMS algorithm involves generating prefix trees and condition trees as intermediate structures during the mining process. High-utility patterns are then derived from the conditional tree. However, a limitation of this algorithm is the necessity to generate numerous prefix trees and condition trees each time a pattern is mined, resulting in significant time consumption. Additionally, employing a storage tree structure requires updating the state of each node whenever the sliding window moves, especially if non-leaf nodes must be deleted, leading to increased processing time. Also, approaching from the perspective of a sliding window, the SHU-Growth [26] algorithm used a tree structure similar to HUS-Tree to store highly useful pattern information in the current sliding window. However, the strength of SHU-Growth lies in the addition of two techniques, Reducing Global Estimated Utilities and Reducing Local Estimated Utilities, aimed at reducing the search space and pruning out unpromising candidate patterns. In some datasets, experimental results have indicated that it is more effective than HUPMS.

Most recently, in 2022, Reddy et al. proposed an improved algorithm based on a tree structure called EGUI-Tree [27] to mine high utility itemsets from streaming data under the sliding window model. In this algorithm, each node in the tree stores information about the candidate patterns, including the transaction utility value and the remaining utility value. The authors proposed a formula to recalculate the utility value of patterns after each information update to reduce search space. The limitation of this algorithm is that the update operation involves updating the entire tree structure, including modifying the pattern values in each node, using batch information update operations, and shifting batch positions, which requires a significant amount of time and memory to execute. In addition, the deletion operation still has the limitation of the HUPMS algorithm when it needs to check all the information of the child nodes of the node to be deleted.

To overcome these limitations, this article proposes a high-utility pattern mining method based on a linked list structure called HUSLL. Unlike the HUPMS algorithm, this method eliminates the need to construct prefix trees or condition trees while still effectively identifying high-utility patterns in the transaction stream database. Utilizing a list structure simplifies node updates, thereby reducing processing time when handling data updates.

III. PROBLEM DEFINITION

This article adopts and builds upon definitions from previous studies [28]. Let $I = \{i_1, i_2, \ldots, i_n\}$ represent the set of items (or events), and $D = \{T_1, T_2, \ldots, T_m\}$ represent the transaction database, where each $T_j \in D$ is a children's set of items from I.

Definition 1: The internal utility of each item, denoted as $iu(i, T_j)$, is the frequency of item i appearing in transaction T_j. For instance, in Table I, item a appears twice in transaction T_1, so the internal utility of item a in transaction T_1 is denoted as $iu(< a >, T_1) = 2$.

Definition 2: External utility of each item, denoted as $eu(i)$, represents the value of item i. For example, in Table II, the external utility of item a is $eu(< a >) = 5$.

Definition 3: The utility of item i in transaction T_j, denoted as $u(i, T_j)$, is calculated by the formula (1):

$$u(i, T_j) = eu(i) \times iu(i, T_j) \tag{1}$$

For example $u(< a >, T_1) = 2 \times 5 = 10$, using data from Table I and Table II.

Definition 4: The utility of itemset X in transaction T_j, denoted as $u(X, T_j)$, is calculated by the formula (2):

$$u(X, T_j) = \sum_{i \in X} u(i, T_j) \quad (2)$$

Where $X = \{i_1, i_2, \ldots, i_k\}$ is the set of k items, $X \subseteq T_j$ and $1 \leq k \leq n$. For example, the item set $< ac >$ in transaction T_1 has a utility of $u(< ac >, T_1) = 2 \times 5 + 3 \times 10 = 40$, using data from Table I and Table II.

Definition 5: The utility of itemset X in the database, denoted as $u(X)$, is calculated by the formula (3):

$$u(X) = \sum_{X \subseteq T_j, T_j \in D} u(X, T_j) \quad (3)$$

For example, the utility of $< ac >$ in Table I is $u(< ac >) = u(< ac >, T_1) + u(< ac >, T_5) + u(< ac >, T_6) + u(< ac >, T_7) = 235$.

Definition 6: The utility of transaction T_j, denoted as $tu(T_j)$, is the total utility value of the items included in transaction T_j, calculated by formula (4):

$$tu(T_j) = \sum_{i \in T_j} u(i, T_j) \quad (4)$$

For example, the utility of transaction T_1 in Table I is $tu(T_1) = u(< a >, T_1) + u(< c >, T_1) + u(< d >, T_1) = 2 \times 5 + 3 \times 10 + 4 \times 7 = 68$.

Definition 7: The minimum utility threshold δ is a predetermined percentage value based on the total utility value of all transactions in the database. In Table I, the total utility of all transactions is 734. The minimum utility value can be defined using formula (5):

$$minutil = \delta \times \sum_{T_j \in D} tu(T_j) \quad (5)$$

For example, when $\delta = 20\%$, the minimum utility value is calculated as $minutil = 20\% \times 734 = 146.8$. A set of items X is deemed a high-utility pattern if $u(X) \geq minutil$. The problem of mining high-utility patterns is framed as identifying all sets X that satisfy the condition $u(X) \geq minutil$.

In the frequent pattern mining problem, the Apriori property proves valuable for the early elimination of infrequent patterns by showing that all extensions of an infrequent event are also infrequent. However, this property does not apply to the problem of mining high-utility patterns. For instance, if $minutil$ is set at 146.8, then the pattern $< a >$ in Table I is not considered a high-utility pattern because $u(< a >) = 115$. Nevertheless, the extended set of $< a >$ is $< ac >$, which qualifies as a high-utility pattern since $u(< ac >) = 235$. Consequently, a measure called utility weight has been introduced to address the high-utility pattern mining problem.

Definition 8: The transaction weight utility of a set of items X, denoted as $twu(X)$, is calculated as the sum of the utility values of transactions containing X, defined by formula (6):

$$twu(X) = \sum_{X \subseteq T_j, T_j \in D} tu(T_j) \quad (6)$$

For example, in Table I the transaction weight utility of the item $< d >$ is calculated as $twu(< d >) = 68 + 43 = 111$. The Apriori property can be applied to this measure. For example, with $minutil = 146.8$, then $twu(< d >) < minutil$. Subsequently, the expanded pattern of $< d >$ is $< dc >$, which also exhibits a transaction weight utility smaller than minutil or $twu(< dc >) < minutil$.

A pattern X is considered a high transaction-weighted utility pattern if $twu(X) \geq minutil$.

Transaction stream data is a prevalent type of data in practical applications, encompassing sales data, website operations data, weather data, etc. In this type of data, the volume is unlimited and can grow over time. A batch of transactions comprises numerous individual transactions, while a window encompasses several batches. Figure 1 illustrates an example of transaction stream data, with the database divided into four equal-sized batches and two windows. Each batch contains two transactions, and each window comprises three batches. Window W_1 includes batches B_1, B_2, and B_3, while window W_2 includes batches B_2, B_3, and B_4.

Definition 9: A transaction T is said to be reorganized, if the items present in T are arranged in the increasing order of their twu values [27]. For example, reorganized transaction for $T_1 = \{< a >, < c >, < d >\}$ is $\{< d >, < a >, < c >\}$ because $twu(< d >) = 111 < twu(< a >) = 439 < twu(< c >) = 647$.

Definition 10: Given an itemset X and a reorganized transaction T with $X \subseteq T$, the set of items after X in T is represented as $A(T/X)$ and are available in T followed by X. For example, set of items after itemset $(< d >, < a >)$ in T_1 given by $A(T_1/\{< d >, < a >\}) = \{< c >\}$.

Definition 11: The remaining utility [27] of itemset X in a reorganized transaction T_j, denoted as $ru(X, T_j)$, is calculated by the formula (7):

$$ru(X, T_j) = \sum_{i_k \subseteq A(T_j/X)} u(i_k, T_j) \quad (7)$$

For example, $ru(< d >, T_1) = u(< a >, T_1) + u(< c >, T_1) = 2 \times 5 + 3 \times 10 = 40$.

Definition 12: The utility of itemset X in a batch B_k is defined by formula (8):

$$u_{B_k}(X) = \sum_{T_j \in B_k} u(X, T_j) \quad (8)$$

TABLE I
TRANSACTION DATABASE

TID	$< a >$	$< b >$	$< c >$	$< d >$	$< e >$	tu
T_1	2	0	3	4	0	68
T_2	4	3	0	0	0	44
T_3	0	5	7	0	0	110
T_4	0	2	0	1	5	43
T_5	6	7	2	0	0	106
T_6	3	0	5	0	1	69
T_7	8	8	4	0	2	152
T_8	0	9	5	0	5	142

TABLE II
EXTERNAL UTILITY OF ITEMS

Item	eu(i)
$< a >$	5
$< b >$	8
$< c >$	10
$< d >$	7
$< e >$	4

For example, in Figure 1, and considering data in Table II, $u_{B_3}(< ac >) = u(< ac >, T_5) + u(< ac >, T_6) = 45 + 70 = 115$.

Definition 13: The utility of itemset X in a window W_l is defined by formula (9):

$$u_{W_l}(X) = \sum_{B_k \in W_l} u_{B_k}(X) \tag{9}$$

For example, in Figure 1, and considering data in Table II, $u_{W_2}(ac) = u_{B_2}(ac)+u_{B_3}(ac)+u_{B_4}(ac) = 0+115+80 = 195$.

Definition 14: The minimum utility threshold in a window W_l, denoted by δ_{W_l}, is a given percentage value based on the total utility value of all transactions in the W_l window. In Figure 1, the total utility of all transactions in window W_2 is 622. If $\delta_{W_l} = 20\%$, then the minimum utility in window W_l can be defined by formula (10):

$$minutil_{W_l} = \delta_{W_l} \times \sum_{T_j \in W_l} tu(T_j) \tag{10}$$

For example, with $\delta_{W_2} = 20\%$, the minimum utility value is $minutil_{W_2} = 20\% \times 622 = 124.4$.

An item set X is called a high-utility pattern in window W_l if $u_{W_l}(X) \geq minutil_{W_l}$. The problem of finding a high-utility pattern is defined as identifying all item sets X that satisfy the condition $u_{W_l}(X) \geq minutil_{W_l}$.

For example, for $\delta_{W_2} = 20\%$, the pattern $< ac >$ is a high-utility pattern in window W_2 because $u_{W_2}(< ac >) = 195$.

Definition 15: The transaction weight utility of an item set X in window W_l, denoted by $twu_{W_l}(X)$, is calculated as the total utility value of the transactions in window W_l containing X. For example, $twu_{W_2}(< ac >) = twu_{B_2}(< ac >)+twu_{B_3}(< ac >)+twu_{B_4}(< ac >) = 106+69+152 = 327$.

X is said to be a high transaction weight utility pattern in window W_l if $twu_{W_l}(X) \geq minutil_{W_l}$. For example, with $minutil_{W_2} = 124.4$, the pattern $< ac >$ is a high-weighted utility pattern because $twu_{W_2}(< ac >) = 327$.

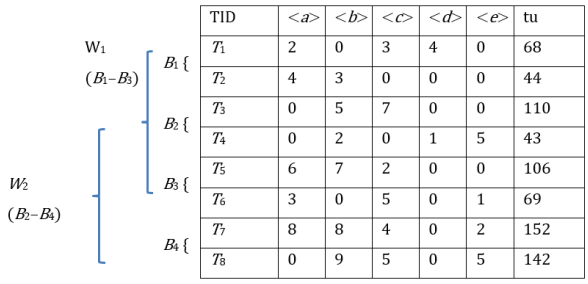
Figure 1 illustrates a transaction stream database. In this representation, the symbols T_i, with $1 \leq i \leq 8$, represent each transaction occurring the ith time. The letters in the set $\{< a >, < b >, < c >, < d >, < e >\}$ denote distinct items in the database. The digits in each row T_i in the column of letters in the set $\{< a >, < b >, < c >, < d >, < e >\}$ represent the internal utility of the item in transaction T_i. For example, in transaction T_1, item $< a >$ has an internal utility of 2. The symbol tu represents the utility of each transaction. The symbol B_j, with $1 \leq j \leq 4$, represents trading batches, each containing a certain number of transactions. Different batches have the same number of transactions, and transactions in each batch do not overlap. The notation W_k, with $1 \leq k \leq 2$, represents sliding windows, each containing a certain number of batches. Batches in different sliding windows may overlap or two different sliding windows may contain the same batches. As shown in Figure 1, sliding window W_2 can contain batches B_2 and B_3, which are also present in sliding window W_1. Window W_1 is the first window created, and it can only contain a maximum of three trading batches. After inserting batches B_1, B_2, and B_3 into window W_1, the window is full. Upon examining transaction T_7 in transaction batch B_4, sliding window W_1 moves to sliding window W_2. In the sliding window model, the algorithm only performs pattern mining on a sliding window containing a certain number of transaction batches.

The problem of mining high-utility patterns on transaction stream data is stated as follows: Given a transaction stream database D and a user-defined minimum utility threshold ($minutil$), mining high-utility patterns in the sliding window W is to find the set of all patterns X whose utility weight $twu(X)$ is greater than or equal to minutil in the sliding window W.

IV. THE PROPOSED ALGORITHM

a) High-Untility Stream Linked List Data

In this section, the article introduces a linked-list data structure called HUSLL, designed for storing information about transactions in the database.



TID	<a>	<b>	<c>	<d>	<e>	tu
T_1	2	0	3	4	0	68
T_2	4	3	0	0	0	44
T_3	0	5	7	0	0	110
T_4	0	2	0	1	5	43
T_5	6	7	2	0	0	106
T_6	3	0	5	0	1	69
T_7	8	8	4	0	2	152
T_8	0	9	5	0	5	142

Figure 1. Example of transaction stream data

The most apparent difference between the HUSLL structure and the Prefix Tree structure is the transformation of organizing nodes in the tree structure into nodes organized in a linked list. A sorted list can be used to execute the mining algorithm without the necessity of storing an intermediate structure mapping to nodes with the same item id. Additionally, deleting nodes in the tree requires more processing steps compared to handling a linked list. This is because the node deletion in a tree involves checking and traversing each child node in the branch to be deleted. In contrast, with a linked list, the deletion operation occurs by simply removing the node from the linked list via the next link, without needing to consider the linked child nodes of the node being examined.

For example, suppose that in window W_1 of batch B_1, there are two items $< m >$ and $< o >$ satisfying the threshold in T_1 and T_2. However, $< m >$ and $< o >$ do not appear in the transactions of window W_2. Therefore, when transitioning from W_1 to W_2, the data in batch B_1 will be deleted. Consequently, items $< m >$ and $< o >$ will also be deleted. In the case of organizing data in a tree structure, to delete the node containing $< m >$, we need to check if $< m >$ has any child nodes because the deletion operation will be performed on the child nodes first. Additionally, to perform mining on the tree using the HUPMS algorithm, an intermediate structure is needed to store the mapping to node $< m >$ in the tree. Therefore, deleting a node $< m >$ requires updating the intermediate structure. This operation consumes resources, especially in the case of continuous updates to the transaction stream database, where batch addition and deletion occur more frequently than in a static database. In contrast, when the data is organized as a linked list, node deletion is much simpler because there is no need to consider child nodes or update intermediate mapping structures.

Each node in the HUSLL structure stores identification information about an item (or event), the value of the node, and the values of transaction batches, which consist of multiple consecutive transactions containing that item. Transaction batches are numbered according to the order

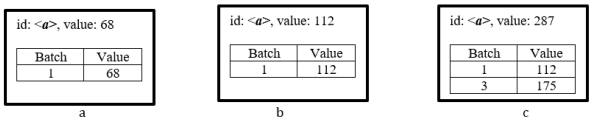


Figure 2. Structure of a node that stores information on item $< a >$

of transactions in the database. To achieve this, a hash table structure is employed to store information about the values of transaction batches. Utilizing a hash table for batch information facilitates quick updates of batch values when the mining algorithm examines transactions within a specific batch. Additionally, using a hash table helps conserve storage space by avoiding the need to store values of batches that do not contain the item under consideration. Deleting a batch within a node is also more straightforward with a hash table when there is sliding window movement, compared to using a list. Figure 2 illustrates the structure of a node in the list. Figure 2a demonstrates the structure of the node storing information about item $< a >$ with batch B_1 containing the value of transaction T_1 in Figure 1. Figure 2b illustrates the data update activity on batch B_1 when exploring transaction T_2. Figure 2c showcases the structure of the node storing information about item $< a >$ after exploring transactions in three batches: B_1, B_2, and B_3. Since item $< a >$ does not appear in batch B_2, information about batch B_2 is not stored in the list node. For ease of presentation, the node structure of item $< a >$ in Figure 2c is described as follows: $\{< a >: [1 : 112], [3 : 175]\}$. The value of a node represents the total value of the transaction batches that the node contains. In Figure 2c, the node value is 287, which is the sum of the values of batches B_1 and B_3.

To represent the relationships between nodes, links are established. An "after" link is employed to signify the sequential order of items in a transaction. For instance, in transaction T_1 in Figure 1, item $< c >$ appears after item $< a >$ in alphabetical order. Subsequently, an "after" link is created to indicate that item $< c >$ and item $< a >$ belong to the same transaction, and item $< c >$ follows item $< a >$. The "after" link plays a crucial role in constructing the HUSLL list when exploring transactions in the database. A single node can have multiple after-nodes associated with it. Once the list construction operation is completed, the after links can be removed to reduce memory size before the extraction process takes place. Figure 3 illustrates the after link connecting two nodes, $< a >$ and $< c >$.

The "before" link is employed to represent the preceding order of items within a transaction. For instance, in transaction T_1 in Figure 1, item $< a >$ precedes item $< c >$ in alphabetical order. A "before" link is then established

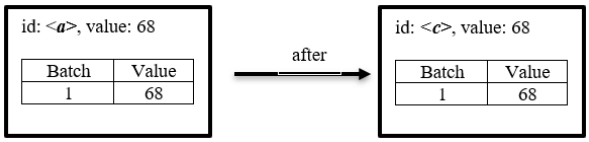


Figure 3. After link between two nodes $< a >$ and $< c >$

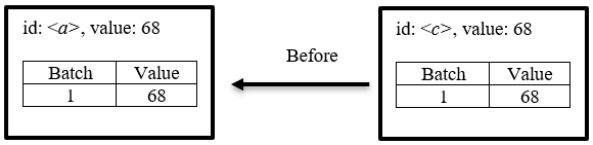


Figure 4. Before link between two nodes $< a >$ and $< c >$

to signify that item $< a >$ and item $< c >$ belong to the same transaction, and item $< a >$ precedes item $< c >$. The "before" link plays a crucial role in the high-utility pattern mining process. For example, if node $< c >$ has a "before" link with node $< a >$, a new pattern $< ac >$ can be created during the candidate generation process. Figure 4 illustrates the "before" link between node $< c >$.

A HUSLL root node is created by default with a null value. As each item in a transaction is examined, the root node establishes an after link with the node containing the first item in the transaction. These nodes also create a before link with the root node. Subsequently, when inspecting other items in the transaction, a new node is generated containing the item, along with before and after links between the nodes.

The next link is employed to connect all the nodes, forming a HUSLL structure. During the list-building process, the nodes created are linked together and sorted based on item identifier information. The next link is utilized in both the mining process and the update process of the HUSLL structure when the sliding window moves.

During mining operations, the algorithm considers items with the same identifier to generate patterns containing a specific identifier. The use of the next link facilitates the easy identification of patterns with the same identifier, as nodes are sorted by item identification. This approach avoids the necessity of using an intermediate mapping structure to locate nodes with the same identifier.

When a sliding window shift operation occurs, a loop traverses all the nodes in the HUSLL based on the next link to identify nodes with updated information about the transaction batch. Once found, a delete operation is performed to eliminate the selected batch of transactions and update the node's value. If the value of a node is 0, the node is removed from the HUSLL by deleting the before, after, and next links to that node. The deletion

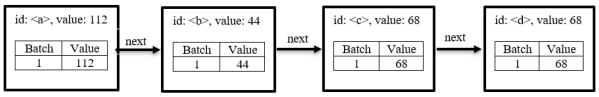


Figure 5. Next link between nodes in transactions T_1 and T_2

of such a link is simpler than updating a tree structure. Simultaneously, not using an intermediate structure to store the links mapping to the node allows update operations on HUSLL to be faster than on the HUS-Tree structure. Figure 5 illustrates the next link between the nodes in transactions T_1 and T_2 in Figure 1. In this figure, transaction T_1, the node containing item $< c >$, has a before link with the node containing item $< a >$ and has no connection with the node containing item $< b >$. However, because item $< b >$ is placed before item $< c >$ in the alphabet, when the next link is created, item $< b >$ will be positioned before item $< c >$.

The process of constructing the HUSLL structure involves creating nodes to store item information and establishing before, after, and next links connecting these nodes. Initially, an empty root node is created. Utilizing the information from Table I, the algorithm sequentially goes through each item in each transaction in alphabetical order to add each item to the HUSLL structure. For example, during the traversal operation of transaction T_1 (with a total utility value, tu, of 68 and items $< a, c, d >$), the item $< a >$ is added to the list by creating a new node with item-id $< a >$ and twu value 68. The resulting node is $\{< a >: [1 : 68]\}$. A next link is established from the root node to $\{< a >: [1 : 68]\}$, and the root node will have an after link to $\{< a >: [1 : 68]\}$. Simultaneously, $\{< a >: [1 : 68]\}$ will have a before link to the root node. Subsequently, item $< c >$ is added to the HUSLL by creating the node $\{< c >: [1 : 68]\}$. Since, in transaction T_1, item $< c >$ comes after item $< a >$, an after link is created between node $\{< c >: [1 : 68]\}$ and node $\{< a >: [1 : 68]\}$. Additionally, a before link is established between node $\{< a >: [1 : 68]\}$ and node $\{< c >: [1 : 68]\}$. Node $\{< c >: [1 : 68]\}$ is inserted at the end of the HUSLL list, creating a next link between node $\{< a >: [1 : 68]\}$ and node $\{< c >: [1 : 68]\}$. Similarly, item $< d >$ is added to the HUSLL by creating node $\{< d >: [1 : 68]\}$. As item $< d >$ follows item $< c >$ in transaction T_1, a before link is formed from node $\{< d >: [1 : 68]\}$ to node $\{< c >: [1 : 68]\}$. Two links, after and next, are generated at node $\{< c >: [1 : 68]\}$ to point to node $\{< d >: [1 : 68]\}$. Figure 6 illustrates the HUSLL structure that stores the items contained in transaction T_1.

For simplicity, the HUSLL structure is represented as

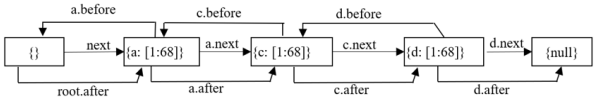


Figure 6. The HUSLL stores information on transaction T_1

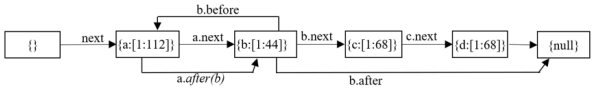


Figure 7. The HUSLL after adding transaction T_2

$\{\} \rightarrow \{< a >: [1 : 68]\} \rightarrow \{< c >: [1 : 68]\} \rightarrow \{< d >: [1 : 68]\}$, where the direction of the arrow signifies the next link.

Next, transaction T_2 is processed to add items to the HUSLL. With a *tu* value of 44, T_2 includes 2 items $< a, b >$. Item $< a >$ from T_2 is incorporated into the HUSLL, but since there is already node $< a >$ with an after link to the root node, and transactions T_1 and T_2 belong to the same batch B_1 (as shown in Figure 1), the *twu* value of item $< a >$ in T_2 is accumulated to node $\{< a >: [1 : 68]\}$, resulting in $\{< a >: [1 : 112]\}$. Subsequently, item $< b >$ is added as the next node after $\{< a >: [1 : 112]\}$ in alphabetical order. The HUSLL structure is designed to maintain alphabetical order, placing item $< b >$ before item $< c >$. Figure 7 illustrates the HUSLL structure after incorporating transaction T_2, with some details omitted for clarity.

Moving on, transaction T_3 is processed to add items to the HUSLL. With a tu value of 110, T_3 includes two items, $< b, c >$. Item $< b >$ from T_3 is introduced into the HUSLL with a *twu* value of 110. Given that there is no after link from the root node to the identifier $< b >$, a new node is created. Additionally, since T_3 belongs to transaction batch B_2, the newly added node will take the form $\{< b >: [2 : 110]\}$. As the list already contains node $\{< b >: [1 : 44]\}$, the new node will be inserted after node $\{< b >: [1 : 44]\}$. Similarly, a new node $\{< c >: [2 : 110]\}$ is generated and added after node $\{< c >: [1 : 68]\}$. Since a new transaction batch is involved, nodes $\{< b >: [2 : 110]\}$ and $\{< c >: [2 : 110]\}$ only store the value of the current batch, which is batch 2. Figure 8 illustrates the HUSLL structure after incorporating transaction T_3.

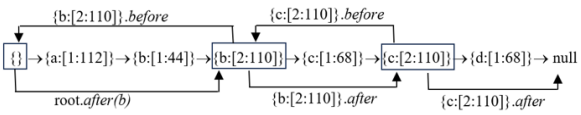


Figure 8. The HUSLL after adding transaction T_3

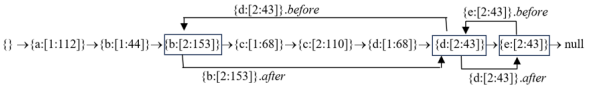


Figure 9. The HUSLL after adding transaction T_4

$\{\} \rightarrow \{< a >: [1 : 112], [3 : 106]\} \rightarrow \{< b >: [1 : 44], [3 : 106]\} \rightarrow \{< b >: [2 : 153]\} \rightarrow \{< c >: [1 : 68]\} \rightarrow \{< c >: [2 : 110]\} \rightarrow \{< c >: [3 : 106]\} \rightarrow \{< d >: [1 : 68]\} \rightarrow \{< d >: [2 : 43]\} \rightarrow \{< e >: [2 : 43]\} \rightarrow null$

Figure 10. The HUSLL after adding transaction T_5

Continuing with the process, transaction T_4 is processed to add items to the HUSLL. With a *tu* value of 43, T_4 encompasses three items: $< b, d, e >$. As the root node has an after link to node $\{< b >: [2 : 110]\}$ and transaction T_4 belongs to batch B_2, the *tu* value of T_4 should be added to node $\{< b >: [2 : 110]\}$ to result in $\{< b >: [2 : 153]\}$. Subsequently, two new nodes $\{< d >: [2 : 43]\}$ and $\{< e >: [2 : 43]\}$ are generated and appended to the list in the following order: $\{< d >: [1 : 68]\}$. Figure 9 illustrates the list structure after the addition of transaction T_4.

Moving forward, transaction T_5 is processed to add items to the HUSLL. With a *tu* value of 106, T_5 includes three items: $< a, b, c >$. Given that the root node already has an after link to node $\{< a >: [1 : 112]\}$ and transaction T_5 belongs to batch B_3, we update node $\{< a >: [1 : 112]\}$ to $\{< a >: [1 : 112], [3 : 106]\}$. The order of *twu* values for batches B_1 and B_3 in node $\{< a >: [1 : 112], [3 : 106]\}$ is represented in the hash table structure. Since there is an after link from node $\{< a >: [1 : 112], [3 : 106]\}$ to node $\{< b >: [1 : 44]\}$, it is updated to $\{< b >: [1 : 44], [3 : 106]\}$. As there is no after link after $\{< b >: [1 : 44], [3 : 106]\}$ to the node with the identifier $< c >$, a new node $\{< c >: [3 : 106]\}$ is created and connected after node $\{< c >: [2 : 110]\}$. The HUSLL structure after adding items in transaction T_5 is illustrated in Figure 10.

Figure 10 illustrates the HUSLL structure after incorporating the items from transaction T_5. In this representation, link information is omitted for simplicity.

Continuing with the process, transaction T_6 is processed to add items to the HUSLL, and the outcomes are illustrated in Figure 11.

With the HUSLL structure, when the window shifts from W_1 to W_2, the data in transaction batch B_1 is removed, and the data from transaction batch B_4 is added to the list. To delete the data of transaction batch B_1, the HUSLL structure is traversed from the first node to the last node based on the next link. At each node traversal, the data

$\{\} \rightarrow \{< a >: [1:112], [3:175]\} \rightarrow \{< b >: [1:44], [3:106]\} \rightarrow \{< b >: [2:153]\} \rightarrow \{< c >: [1:68], [3:69]\} \rightarrow \{< c >: [2:110]\} \rightarrow \{< c >: [3:106]\} \rightarrow \{< d >: [1:68]\} \rightarrow \{< d >: [2:43]\} \rightarrow \{< e >: [2:43]\} \rightarrow \{< e >: [3:69]\} \rightarrow null$

Figure 11. The HUSLL after adding transaction T_6

(twu) of batch B_1 is deleted from the hash table. Utilizing a hash table structure makes finding and deleting the value of batch B_1 in each node easier compared to using a list of batch values. Subsequently, nodes with a twu value of 0 are removed from the list. For example, from the HUSLL list in Figure 11, when the algorithm reaches transaction T_7, the sliding window will shift from window W_1 to W_2. At that point, all the values of batch 1 in each node will be deleted. A loop will iterate through each node in the HUSLL list to find and delete batch 1 data in each node. In Figure 11, we can see that the nodes $\{< a >: [1:112], [3:175]\}$, $\{< b >: [1:44], [3:106]\}$, $\{< c >: [1:68], [3:69]\}$, and $\{< d >: [1:68]\}$ contain data of batch 1. After performing the deletion operation, the values of these nodes will be $\{< a >: [3:175]\}$, $\{< b >: [3:106]\}$, $\{< c >: [3:69]\}$, and $\{< d >:\}$. At this point, the node $\{< d >:\}$ does not contain any data and will be removed from the list. The deletion operation is performed by deleting all links created to the node $\{< d >:\}$ and the next link of the node just before the node $\{< d >:\}$, which is the node $\{< c >: [3:106]\}$ in Figure 11, will be directly linked to the node just after $\{< d >:\}$, which is the node $\{< d >: [2:43]\}$. Finally, the items of transaction batch B_4 are incorporated into the HUSLL. The advantage of maintaining a HUSLL list is that traversing and deleting nodes is considerably more straightforward than with the HUS-Tree structure. The use of a hash table enables the storage of only batches with the item identifier present, without having to store zero values, as is the case in the HUS-Tree structure. Consequently, the update only occurs on nodes containing the batch under consideration, and the deletion of batch B_1 is performed solely on nodes where this batch appears, such as nodes $\{< a >: [1:112], [3:175]\}$, $\{< b >: [1:44], [3:106]\}$, $\{< c >: [1:68], [3:69]\}$, and $\{< d >: [1:68]\}$, instead of having to update all nodes as required by the HUS-Tree structure. Figure 12 illustrates the outcome of the shift operation to window W_2.

The process of constructing the HUSLL list is detailed in algorithm 1.

b) Mining High-Ultility Patterns on the HUSLL List

From the list structure in Figure 12, we can recognize the following characteristics:

Property 1. Nodes containing information about items appearing in the same transaction will possess a "before" link to a node containing information about the item whose identifier immediately precedes the item identifier of the node under consideration. For instance, in Figure 12, the node $\{< e >: [3:69]\}$ has a "before" link to node $\{< c >: [3:69]\}$, and node $\{< c >: [3:69]\}$ has a "before" link to node $\{< a >: [3:175], [4:152]\}$. Simultaneously, the items $< a, c, e >$ appear in the same transactions T_6 and T_7.

$\{\} \rightarrow \{< a >: [3:175], [4:152]\} \rightarrow \{< b >: [3:106], [4:152]\} \rightarrow \{< b >: [2:153], [4:142]\} \rightarrow \{< c >: [3:69]\} \rightarrow \{< c >: [2:110], [4:142]\} \rightarrow \{< c >: [3:106], [4:152]\} \rightarrow \{< d >: [2:43]\} \rightarrow \{< e >: [2:43]\} \rightarrow \{< e >: [3:69]\} \rightarrow \{< e >: [4:152]\} \rightarrow \{< e >: [4:142]\} \rightarrow null$

Figure 12. The HUSLL after sliding to window W_2

Algorithm 1: BuildHUSLL (D, minutil)

```
1  Inputs: Transaction database D, minimum utility
     threshold minutil.
2  Outputs: HUSLL data structure.
3  begin
4      Create root-node R;
5      Set current node C is null;
6      foreach T ∈ D do
7          i=T[0];
8          if R.hasAfter(i) = False then
9              CreateNode for i AS iN;
10             R.setAfter(iN);
11             iN.setBefore = R;
12             call AddToHUSLL (iN,R);
13             C=iN;
14         end
15         else
16             C= R.getNode(i);
17             Update twu of C;
18         end
19         k=1;
20         while k < Length(T) do
21             i=T[k];
22             if C.hasAfter(i) = TRUE then
23                 C=C.getNode(i);
24                 Update twu of C;
25             end
26             else
27                 CreateNode for i AS iN;
28                 C.setAfter(iN);
29                 iN.setBefore = C;
30                 AddToHUSLL (iN,R);
31                 C=iN;
32             end
33             k=k+1;
34         end
35     end
36 end
```

Property 2. A high-utility pattern can be generated by com-

Algorithm 2: AddToHUSLL (iN,R)

```
Inputs: considered node iN, root node R
Outputs: HUSLL
begin
    C = R;
    while c.next is not null do
        if c.next > iN.id then
            iN.next = c.next;
            c.next = iN;
            return;
        end
    end
    c.next = iN
end
```

bining items in the same branch through the "before" link, and the utility value of the pattern is the list of values of the transaction batch of the last node in the branch. For example, in Figure 12, the "before" link of the nodes includes nodes $\{< a >: [3 : 175], [4 : 152]\}$ and node $\{< c >: [3 : 69]\}$. The node $\{< e >: [3 : 69]\}$ can generate the pattern $< ec >$ by concatenating the identifiers of node $\{< c >: [3 : 69]\}$ and node $\{< e >: [3 : 69]\}$. Subsequently, the utility of the $< ec >$ pattern will be the transaction batch value of the first node in the "before" link, which is node $\{< e >: [3 : 69]\}$. The utility value of the pattern $< ec >$ is $\{< ec >: [3 : 69]\}$. Then, the pattern $< eca >$ can be generated by concatenating the identifier of node $\{< ec >: [3 : 69]\}$ and node $\{< a >: [3 : 175], [4 : 152]\}$. Since, in the "before" link, node $\{< e >: [3 : 69]\}$ is the first node, the utility of the $< eca >$ pattern is the transaction batch value of node $\{< e >: [3 : 69]\}$. The utility value of the $< eca >$ pattern is $\{< eca >: [3 : 69]\}$.

Based on the two characteristics mentioned above, the pattern mining process on HUSLL can be outlined as follows: From the existing HUSLL list, iterate through each node using the next link. For each node, determine the linked branch based on the before link. For each branch, concatenate the item identifiers of each node within the branch to create a candidate pattern. Next, assess the utility of the candidate pattern by summing the utility values in the transaction batch of that pattern and comparing it with the minutil value. Candidate patterns exceeding the minutil threshold will be retained, and the concatenation operation to create new candidate patterns will be iterated. After this browsing and merging process, all high-utility patterns will be obtained. Algorithm 3 outlines the process of mining high-utility patterns from the HUSLL list.

For instance, with *minutil* set to 260, item $< c >$ has a utility weight equal to the total utility value in the trading batches, summing up to 69 + 106 + 152 + 110 + 142 = 579. Next, retrieve the id_item_list of nodes containing item $< c >$ in HUSLL, which includes $\{< c >: [3 : 69]\}, \{< c >: [2 : 110], [4 : 142]\}$, and $\{< c >: [3 : 106], [4 : 152]\}$. Starting with node $\{< c >: [3 : 69]\}$, traverse the list based on the before link to generate a pattern $\{< ca >: [3 : 69]\}$. For node $\{< c >: [3 : 106], [4 : 152]\}$, perform a list traversal based on the before link to generate patterns $\{< cb >: [3 : 106], [4 : 152]\}$ and $\{< ca >: [3 : 106], [4 : 152]\}$. From node $\{< c >: [2 : 110], [4 : 142]\}$, traverse the list based on the before link to generate the pattern $\{< cb >: [2 : 110], [4 : 142]\}$. Then, calculate the utility weights of the patterns by summing up the utility values. With the pattern $< ca >$, comprising $\{< ca >: [3 : 69]\}$ and $\{< ca >: [3 : 106], [4 : 152]\}$, the utility is 69 + 106 + 152 = 327. For pattern $< cb >$, consisting of $\{< cb >: [3 : 106], [4 : 152]\}$ and $\{< cb >: [2 : 110], [4 : 142]\}$, the utility weight is 106 + 152 +110 + 142 = 510. Both patterns $< ca >$ and $< cb >$ qualify as high-utility patterns since their utility weights exceed *minutil*. The pattern generation process continues with $< ca >$ and $< cb >$. For the pattern $< ca >$, the node containing $< a >$ is the one with the root node as the before link, so no further patterns can emerge. However, for the pattern $< cb >$, comprising two nodes $\{< cb >: [3 : 106], [4 : 152]\}$ and $\{< cb >: [2 : 110], [4 : 142]\}$, the node $\{< cb >: [2 : 110], [4 : 142]\}$ has node $< b >$ with the before link as the root node, indicating that no additional patterns can be generated. Regarding the node $\{< cb >: [3 : 106], [4 : 152]\}$, the preceding node is $\{< a >: 0, 175, 152\}$, enabling the pattern $\{< cba >: [3 : 106], [4 : 152]\}$ to arise. However, the pattern $< cba >$ with a utility weight of 106 + 152 = 258 does not meet the criteria for a high-utility pattern. The mining process persists until all nodes in HUSLL are traversed.

The HUSLL structure shares similarities with the HUS-Tree structure, but it differs in the use of after links during the extraction process. In contrast to the HUPMS algorithm, which relies on intermediate structures like condition trees, the HSLM algorithm utilizes the previous link, rendering after links unnecessary for mining operations. As a result, after links can be removed during mining to optimize memory usage. Each node in the HUSLL structure contains information such as the item identifier, total batch value, and before and next links. To enhance performance, mining information can be stored in text files for separate mining operations on another processor. This allows for the optimization of both mining structure construction and mining activities by dividing these tasks. One processor can focus on building the HUSLL structure and continuously updating the data, while mining operations can be carried out on one or more separate processors.

Additionally, the use of a hash table to store information

Algorithm 3: HSLM (T, minutil)

```
Inputs: HUSLL T, minimum utility threshold (minutil)
Outputs: High-utility patterns
begin
    foreach id ∈ T do
        Identify the nodes belonging to the branch
          associated with the id;
        Add the nodes to the id_item_list ;
        foreach node n ∈ id_item_list do
            Browse T and combine node n with before
              nodes of n to create pattern nx;
            Save nx in K;
        end
        foreach pattern m ∈ K do
            if utility of m ≥ minutil then
                MiningSubPattern(T,m,minsup);
                Output m;
            end
        end
    end
end
```

Algorithm 4: MiningSubPattern (T,minutil,m)

```
Inputs: HUSLL T, minimum utility threshold (minutil),
  pattern m
Outputs: High-utility patten (mx)
begin
    Create L to save list;
    Browse T and combine node m with before nodes of
      m to create pattern mx;
    Save mx to L;
    foreach pattern mx∈L do
        if utility of mx ≥ minutil then
            MiningSubPattern(T,mx,minsup);
            Output mx;
        end
    end
end
```

about item batches in each node facilitates easy updating of node information when the sliding window moves. This hash table structure speeds up transaction searches and optimizes storage operations by excluding information about batches that do not contain the item identifier of the node under consideration. The next link enables straightforward node deletion operations. When a node's value reaches 0, the node is removed from the HUSLL list by deleting all before, after, and next links and updating the value to null. In contrast, the HUS-Tree structure requires more complex deletion operations, especially for branch nodes, where updating is not easily accomplished. Traversing the next link from the root node to the last node in the list simplifies state updates for the entire mining structure.

V. EXPERIMENTAL RESULTS

In this section, we present the experimental results comparing the effectiveness of the HUPMS [11] algorithm , EGUI-TREE [27] algorithm and the proposed HSLM algorithm. The study utilizes four real-life datasets for comparison: chainstore, foodmart, ecommerce, and chicago_crimes. Detailed information about each dataset is provided in Table III. The program is implemented in Java and executed on a system running Windows 10, equipped with an Intel® Core™ i7-9700 processor and 16384 MB RAM.

TABLE III
EXPERIMENTAL DATA

Dataset name	Trans No.	Item No.	AvgLen
foodmart_utility	4,141	1,559	4.42
chainstore_utility	1,112,949	46,086	7.23
ECommerce_retail_utility	14,975	3,468	11.71
Chicago_Crimes	2,662,309	35	1.795

The input data for all three algorithms is a transactional database with three parameters: (i) the number of transactions per batch, (ii) the number of batches in each window, and (iii) the minimum useful threshold value, *minutil*. In contrast to the HUPMS algorithm, the EGUI-Tree algorithm incorporates the *ru* value for every pattern to facilitate the early pruning of unpromising patterns. Specifically, in the first phase, the EGUI-Tree algorithm constructs the global utility 1-item sets to store information about the *tu* and *ru* values of each item per transaction and batch for the current window. Next, based on the *ru* value, EGUI-Tree decides whether to expand the items in the 1-item set to create candidate patterns and add these candidate patterns to the EGUI-Tree. Each node in this structure stores information about the pattern. The EGUI-Tree is subdivided into various levels. The candidate patterns at each level have the same number of items in each pattern. Finally, the algorithm traverses the EGUI-Tree to find high utility patterns by comparing the utility value of each pattern stored on the tree with a minutil. The algorithm updates the information for both the global utility 1-item sets and the EGUI-Tree whenever the window changes.

a) Comparision of execution time when performing

In this section, we present experimental results from the high-utility pattern mining process on various datasets, with a focus on adjusting the number of batches per window, ranging from 2 to 6 for different datasets. Each dataset is associated with specific parameters. For the chainstore dataset, the minimum utility threshold is set to 0.25%, and the number of transactions per batch is 100K. The foodmart dataset has a minimum utility threshold of 0.25%, with 200 transactions per batch. In the case of the ecommerce dataset, the minimum utility threshold is 0.75%, and there are 2000 transactions per batch. Lastly, for the Chicago dataset, a minimum utility threshold of 0.15% is chosen, and there are 300K transactions per batch. The article

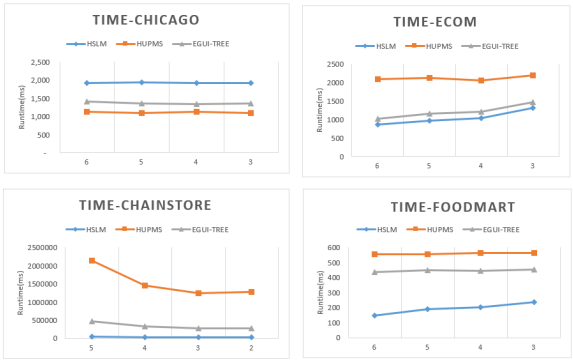


Figure 13. Comparison results according to execution time

compares the execution time to assess the effectiveness of both the proposed algorithm (HSLM) and the HUPMS algorithm. The results depicted in Figure 13 illustrate the execution times.

Figure 13 reveals that for highly dense datasets like chainstore, ecommerce, and foodmart, the HUPMS algorithm requires operations involving the construction of prefix trees, condition trees, and utility value calculations for patterns. As a result, HUPMS utilizes more execution time and memory. Conversely, HSLM can perform extraction without the need to generate prefix trees and condition trees, leading to a lower execution time for the proposed HSLM algorithm. Additionally, the EGUI-Tree algorithm performs at a faster rate than the HUPMS algorithm on these datasets. However, the update operation of the EGUI-Tree requires traversing the batch list to update patterns, which consumes more time than the HSLM algorithm. Besides, with sparse datasets like Chicago, where the density is very low, the time spent creating and deleting the condition tree and prefix tree of the HUPMS algorithm becomes negligible. In such cases, the execution time of HUPMS surpasses that of the proposed algorithm.

The above experimental results indicate that the proposed HSLM algorithm outperforms the HUPMS algorithm on dense datasets. Additionally, the results reveal that as the number of batches per window increases, the mining time also tends to increase.

b) Comparision of excution time with different batch

In this section, the article examines the experimental results by varying the number of transactions in each batch. Figure 14 illustrates the experimental outcomes on the chainstore dataset. In Figure 14a, as the number of transactions per batch increases from 50K to 100K, the HUPMS, EGUI-Tree, and HSLM algorithms exhibit a decrease in mining time. Conversely, when reducing the number of transactions per batch, the mining time

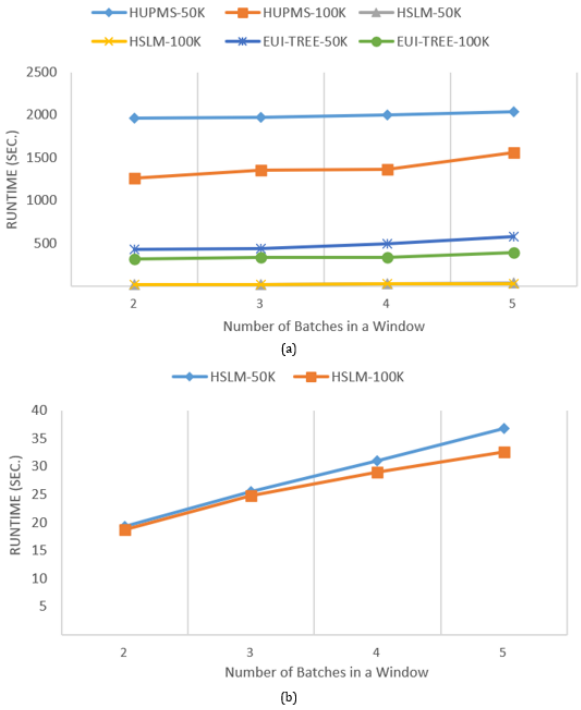


Figure 14. Comparison results based on the number of transactions per batch

increases correspondingly. Figure 14b specifically presents the mining time of the HSLM algorithm on the chainstore dataset while varying the number of transactions per batch from 50K to 100K. The results indicate that a decrease in the number of transactions per batch leads to an increase in the number of mining algorithm executions, subsequently elevating the overall mining time.

VI. CONCLUSIONS

This article introduced a high-utility pattern mining technique for transaction stream data based on the HUSLL list data structure. The experimental results demonstrated that the proposed method outperforms the HUPMS algorithm, particularly on dense datasets such as chainstore, foodmart, or ecommerce.

The HUSLL structure, resembling the HUS-Tree structure, offers the advantage of not using after links in mining operations. This feature facilitates storing HUSLL list information in text files, enabling external memory storage. Consequently, mining operations can be conducted on a different processor using the file containing the HUSLL list structure. This separation of update and exploitation steps allows efficient handling of the latest sliding window information stored in the text file without the need to rerun the algorithm from the beginning. Storing in text files also enables users to retain information about data status at

various exploitation stages, facilitating the comparison of patterns and strategic planning for subsequent stages.

Moreover, the linked list organization proves to be more effective in updating the state compared to tree data organization, particularly in a stream data mining environment where transaction information undergoes regular updates. This adaptability contributes to the practical effectiveness of the HSLM algorithm.

The proposed algorithm's performance is not particularly outstanding in sparse datasets. This is a limitation when mining high-utility sets for sparse datasets. Moreover, the process of building the linked list sequentially does not reflect parallel processing during list creation. Therefore, the mining has not been optimized for the case of a large window size. Our research will continue to focus on parallelizing the list-building process and using early candidate pruning techniques to optimize mining in the future.

The utilization of textual data opens avenues for research in parallel and simultaneous processing of multiple datasets from various sources. It also provides users with the flexibility to transform shared data while retaining valuable information gleaned from exploitation activities.

Additionally, the mining of closed high-utility sets of stream data via a sliding window [29] helps eliminate highly useful but non-repeating patterns to reduce the time and space required for mining, which also needs to be further researched.

ACKNOWLEDGEMENT

This research is funded by Ho Chi Minh City University of Foreign Languages – Information Technology under grant number H2022-05.

REFERENCES

[1] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215. Santiago, Chile, 1994, pp. 487–499.

[2] P. Fournier-Viger, J. Chun-Wei Lin, T. Truong-Chi, and R. Nkambou, "A survey of high-utility itemset mining," *High-utility pattern mining: Theory, algorithms and applications*, pp. 1–45, 2019.

[3] S. Krishnamoorthy, "Hminer: Efficiently mining high-utility itemsets," *Expert Systems with Applications*, vol. 90, pp. 168–183, 2017.

[4] S. Zida, P. Fournier-Viger, J. C.-W. Lin, C.-W. Wu, and V. S. Tseng, "Efim: a fast and memory efficient algorithm for high-utility itemset mining," *Knowledge and Information Systems*, vol. 51, no. 2, pp. 595–625, 2017.

[5] M. K. Sohrabi, "An efficient projection-based method for high utility itemset mining using a novel pruning approach on the utility matrix," *Knowledge and Information Systems*, vol. 62, pp. 4141–4167, 2020.

[6] P. Wu, X. Niu, P. Fournier-Viger, C. Huang, and B. Wang, "Ubp-miner: An efficient bit based high utility itemset mining algorithm," *Knowledge-Based Systems*, vol. 248, p. 108865, 2022.

[7] J. C.-W. Lin, W. Gan, T.-P. Hong, B. Zhang *et al.*, "An incremental high-utility mining algorithm with transaction insertion," *The Scientific World Journal*, vol. 2015, 2015.

[8] P. Fournier-Viger, J. C.-W. Lin, T. Gueniche, and P. Barhate, "Efficient incremental high-utility itemset mining," in *Proceedings of the ASE BigData & SocialInformatics 2015*, 2015, pp. 1–6.

[9] U. Yun, H. Ryang, G. Lee, and H. Fujita, "An efficient algorithm for mining high-utility patterns from incremental databases with one database scan," *Knowledge-Based Systems*, vol. 124, pp. 188–206, 2017.

[10] H.-F. Li, H.-Y. Huang, Y.-C. Chen, Y.-J. Liu, and S.-Y. Lee, "Fast and memory efficient mining of high-utility itemsets in data streams," in *2008 eighth IEEE international conference on data mining*. IEEE, 2008, pp. 881–886.

[11] C. F. Ahmed, S. K. Tanbeer, and B.-S. Jeong, "Efficient mining of high-utility patterns over data streams with a sliding window method," *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing 2010*, pp. 99–113, 2010.

[12] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM sigmod record*, vol. 29, no. 2, pp. 1–12, 2000.

[13] G. Pyun, U. Yun, and K. H. Ryu, "Efficient frequent pattern mining based on linear prefix tree," *Knowledge-Based Systems*, vol. 55, pp. 125–139, 2014.

[14] Y. Mao, B. Wu, Q. Deng, S. Mahmoodi, Z. Chen, and Y.-C. Chen, "A novel parallel frequent itemset mining algorithm for automatic enterprise," *Enterprise Information Systems*, p. 2204317, 2023.

[15] J. Lu, W. Xu, K. Zhou, and Z. Guo, "Frequent itemset mining algorithm based on linear table," *Journal of Database Management (JDM)*, vol. 34, no. 1, pp. 1–21, 2023.

[16] P. Fournier-Viger, C.-W. Wu, S. Zida, and V. S. Tseng, "Fhm: Faster high-utility itemset mining using estimated utility cooccurrence pruning," in *Foundations of Intelligent Systems: 21st International Symposium, ISMIS 2014, Roskilde, Denmark, June 25-27, 2014. Proceedings 21*. Springer, 2014, pp. 83–92.

[17] J. Liu, K. Wang, and B. CM Fung, "Direct discovery of high-utility itemsets without candidate generation," in *2012 IEEE 12th international conference on data mining*, 2012, pp. 984–989.

[18] Y. Liu, W.-k. Liao, and A. Choudhary, "A two-phase algorithm for fast discovery of high-utility itemsets," in *Advances in Knowledge Discovery and Data Mining: 9th Pacific-Asia Conference, PAKDD 2005, Hanoi, Vietnam, May 18-20, 2005. Proceedings 9*. Springer, 2005, pp. 689–695.

[19] J.-S. Yeh, C.-Y. Chang, and Y.-T. Wang, "Efficient algorithms for incremental utility mining," in *Proceedings of the 2nd international conference on Ubiquitous information management and communication*, 2008, pp. 212–217.

[20] C. F. Ahmed, S. K. Tanbeer, B.-S. Jeong, and Y.-K. Lee, "Efficient tree structures for high-utility pattern mining in incremental databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 12, pp. 1708–1721, 2009.

[21] H.-T. Zheng and Z. Li, "ichum: an efficient algorithm for high-utility mining in incremental databases," in *Knowledge Science, Engineering and Management: 8th International Conference, KSEM 2015, Chongqing, China, October 28-30, 2015, Proceedings 8*. Springer, 2015, pp. 212–223.

[22] U. Yun and H. Ryang, "Incremental high-utility pattern mining with static and dynamic databases," *Applied intelligence*, vol. 42, pp. 323–352, 2015.

[23] M. Zihayat, Y. Chen, and A. An, "Memory-adaptive high

utility sequential pattern mining over data streams," *Machine Learning*, vol. 106, pp. 799–836, 2017.
[24] D. Kim and U. Yun, "Mining high utility itemsets based on the time decaying model," *Intelligent Data Analysis*, vol. 20, no. 5, pp. 1157–1180, 2016.
[25] U. Yun, D. Kim, E. Yoon, and H. Fujita, "Damped window based high average utility pattern mining over data streams," *Knowledge-Based Systems*, vol. 144, pp. 188–205, 2018.
[26] H. Ryang and U. Yun, "High utility pattern mining over data streams with sliding window technique," *Expert Systems with Applications*, vol. 57, pp. 214–231, 2016.
[27] P. A. Reddy and M. K. Prasad, "Sliding window-based high utility item-sets mining over data stream using extended global utility item-sets tree," *International Journal of Software Innovation (IJSI)*, vol. 10, no. 1, pp. 1–16, 2022.
[28] H. Yao and H. J. Hamilton, "Mining itemset utilities from transaction databases," *Data & Knowledge Engineering*, vol. 59, no. 3, pp. 603–626, 2006.
[29] M. Li, M. Han, Z. Chen, H. Wu, and X. Zhang, "Fchm-stream: fast closed high utility itemsets mining over data streams," *Knowledge and Information Systems*, vol. 65, no. 6, pp. 2509–2539, 2023.

Dr. Tran Minh Thai received a Bachelor's degree in 2001 and a Master's degree in Computer Science in 2006, both from the University of Natural Sciences, Vietnam National University, HCMC. In 2017, he was awarded a Ph.D. in Computer Science from Vietnam National University, HCMC. From 2002 to 2015, he served as a lecturer and manager of the Information Technology Department at the Information Technology College HCMC. Since 2015, he has been a lecturer and Head of the Information Systems Department of the Faculty of Information Technology at the HCMC University of Foreign Languages - Information Technology. Dr. Thai's primary research interests lie in data mining, data privacy, big data processing, and pattern recognition.
Email: thaitm@huflit.edu.vn

Anh-Duy Tran received a Master's degree in Computer Science from the University of Natural Sciences, Vietnam National University, HCMC in 2017. He serves as a Lecturer at the Department of Information Technology at the University of Foreign Languages and Information Technology in Ho Chi Minh City. His contributions to the field of data mining aim to empower businesses and organizations to make data-driven decisions, fostering innovation and advancements in various domains.
Email: duyta@huflit.edu.vn

Duc-Thanh Pham received his Master's degree in 2006 from Ho Chi Minh City National University; Currently working as a Lecturer at the Faculty of Information Technology, Ho Chi Minh City University of Foreign Languages and Information Technology; The research field of interest is: data mining
Email: thanhpd@huflit.edu.vn

Minh-Nguyen Le received his Master's degree in 2007 from Ho Chi Minh City National University; Currently working as a Lecturer at the Faculty of Information Technology, Ho Chi Minh City University of Foreign Languages and Information Technology; The research field of interest is: data mining.
Email: nguyenltm@huflit.edu.vn