Mining Association Rules on Time-sensitive Data

Dang-Ngo Huu[1,3], Nghia Le[2,3], Khac-Chien Nguyen[4]
[1]University of Science, Ho Chi Minh City, Viet Nam, Vietnam
[2]University of Information Technology, Ho Chi Minh City, Viet Nam, Vietnam
[3]Vietnam National University, Ho Chi Minh City, Vietnam
[4]People's Police University, Ho Chi Minh City, Vietnam

Correspondence: Khac-Chien Nguyen, nkchienster@gmail.com
Communication: received 13 Feb 2024, revised 15 Apr 2024, accepted 26 May 2024
Digital Object Identifier: 10.32913/mic-ict-research.v2024.n2.1256

Abstract: Real-world data is continuously produced and collected quickly; hence, mining association rules are used to find relationships between item sets. One of these combined rule mining techniques is Rare Association Rule Mining (RARM), a method that extracts rare, combined rules with low support but high confidence from the database. Additionally, a large amount of time-sensitive data is created in many fields, and combining valid and outdated data leads to low efficiency in extracting combined rules. This paper explores a mining method that increases rare, combined rules on time-sensitive data to extract common and rare patterns in the database. The primary approach of the research proposes a new tree structure called ILC-tree, inspired by the LC-tree structure published in 2022. Specifically, our approach includes a new tree structure and a fast reconstruction method. The proposed approach reduces execution time and memory consumption by approximately 2 times compared to the old tree structure.

Keywords: *Association rules, rare association rules, rare patterns, common patterns, damped window model*

I. INTRODUCTION

Nowadays, data, most of which is non-static, is generated and collected daily. Among these, time-sensitive data is continuously generated and frequently changes over time, characterized by being associated with specific time intervals. Applying methods to optimize such time-sensitive data in terms of factors like memory, speed, etc., can bring many benefits.

Many association rule mining (ARM) methods [1] have been applied to discover relationships between itemsets in databases, and rare association rule mining (RARM) [2], has been proposed. However, when published throughout this process, each method tends to address only part of the problem or still has existing limitations. In 2022, a group of researchers in China proposed a tree structure called LC-tree to mine rare and common patterns in the association rule mining problem [3]. The LC-tree structure combines features resembling the lifecycle of a real tree in data management and has addressed most of the limitations listed. However, due to some inappropriate data structures used in LC-tree, the tree-building time and memory consumption have not been optimized. This research will introduce new approaches, namely a new tree structure improved from the old LC-tree structure, called ILC-tree, and a new restructuring strategy. Overall, with this approach, we set three main goals:

- The proposed tree structure must ensure the advantages of the old LC-tree structure, solving the limitations of previous methods that the LC-tree has already addressed.

- The tree-building time must be faster than LC-tree and previous methods.

- Memory consumption must be less than LC-tree and previous methods.

To achieve these goals, we proposed a new approach to replace the hash table with a single linked list in the child node list, where the parent node will only point to the beginning of the linked list instead of having to point to the entire child node as in the old structure. Next, we propose a method to adjust new nodes in tree restructuring, allowing the permutation of child nodes with a group of nodes from its parent to its predecessors. Our approach helps save memory and reduce the complexity of the node adjustment process from $O\left(n^2\right)$ to $O(n)$.

II. RELATED ISSUES

1. Basic concepts

This section will introduce critical definitions of RARM and the data structure employed in RARM methods.

Let $I = \{x_1, x_2 \ldots, x_n\}$ be a set containing n items, and each subset of I is called an itemset. Let D be a database containing a list of m transactions $T = \{t_1, t_2, \ldots, t_m\}$, where each transaction $t_i \in T$ is an itemset $\{i_1, i_2, \ldots, i_k\}$. A combination rule between two sets X and Y is denoted as $X \rightarrow Y$, where $X \subset I, Y \subset I$ and $X \cap Y \neq \varnothing$.

Definition 1: Support of X, denoted as $Supp(X)$ is the fraction of transactions containing itemsets X in the database D.

$$Supp(X) = \frac{|X|}{|D|} \tag{1}$$

(where $|D|$ is the total number of transactions and $|X|$ is the total number of transactions containing itemsets X)

Definition 2: Support of the rule, denoted as $Supp(X \rightarrow Y)$ is the fraction of transactions containing both itemsets X and Y in the database D.

$$Supp(X \rightarrow Y) = \frac{|X \cup Y|}{|D|} \tag{2}$$

(where $|X \cup Y|$ is the total number of transactions containing both itemsets X and Y)

Definition 3: Confidence of the rule, denoted as $Conf(X \rightarrow Y)$ is the fraction of transactions that contain both itemsets X and Y in transactions that contain itemsets X.

$$Conf(X \rightarrow Y) = \frac{|X \cup Y|}{|X|} \tag{3}$$

Definition 4: (Rare Pattern) A rare pattern is a pattern with a count greater than or equal to the minimum rare threshold *minRare* and less than the minimum frequent threshold *minFre*, expressed as:

$$minFre > Count(X) \geq minRare \tag{4}$$

Definition 5: (Frequent Pattern) A frequent pattern is a pattern with a count greater than or equal to the minimum frequent threshold *minFre*, expressed as:

$$Count(X) \geq minFre \tag{5}$$

Current RARM methods often apply two data structures: 1 tree structure and 1 header table. For example, in the case of SSP-Tree, its tree structure stores item names and frequencies of occurrence. They include tree nodes and links, with tree nodes divided into root nodes, leaf nodes, and branch nodes, defined as follows:

Definition 6: (Root Node) A tree structure has only 1 unique root node, from which the tree path to other nodes begins. The tree node does not contain item information.

Definition 7: (Leaf Node) A leaf node is a node following the tree path, storing the name and frequency of the item.

Definition 8: (Branch Node) A branch node is a node within the tree path connecting the root node and the leaf node. The branch node also stores the name and frequency of the item.

2. Related works

Srikant and Agrawal proposed the first RARM method in 1997, using the Apriori algorithm to generate rare association rules with a minimum support level (min-support) [4]. Based on this foundation, Haglin and Manning applied min-support and max-support to generate rare association rules [5]. Subsequently, improved algorithms with similar mechanisms were developed, such as RSAA [6], Apriori-Inverse [7], and Apriori-Rare [8]. However, Apriori-based RARM algorithms face two significant issues: they must traverse the database multiple times and generate many candidate patterns. Therefore, FP-growth-based RARM algorithms were proposed based on the original FP-growth algorithm [9], applying a tree structure that only requires two passes through the database to extract patterns without generating candidates. However, these methods are designed for static databases, and in practice, continuous data is generated and changes over time. Therefore, these methods may not be well-suited as they must start from scratch for newly updated databases.

To improve the performance of RARM on dynamic databases, researchers have made enhancements to FP-growth-based RARM algorithms. AFPIM [10], FUFP [11], and IFP-tree [12] are algorithms designed for dynamic databases. They all use the FP-tree structure to maintain the compactness of information and avoid scanning as much data as possible to enhance efficiency. Additionally, some methods can update the tree structure without rescanning the initial database. For example, in 2019, Borah and Nath proposed the SSP-Tree algorithm [13] to mine rare association rules in medical diagnosis. The ISSP-tree algorithm [14] is an improved version of the SSP-tree algorithm. Mahdi and colleagues proposed the FR-tree algorithm [15], which can create, filter, and classify association rules without needing an additional threshold. Furthermore, some rare association rule mining methods have been further improved by considering the characteristics of items and databases. However, due to limitations in inefficient data structures, the current rare association rule mining process still consumes considerable time and memory for storing and searching data.

In 2022, a group of authors in China proposed a new tree structure based on the idea of the life cycle of a real tree in nature called the LC-tree. The tree structure is illustrated in Figure 1. Adding the ”null” item to the header table to store a list of pure leaf nodes helps optimize the tree to become faster and easier. This tree structure addresses the shortcomings of previous methods, such as its applicability

to dynamic databases, not requiring multiple scans of the database, and making changes to optimize pattern mining time and consume less resources than existing methods.

However, let's consider the aspects of execution time and memory consumption when building the tree. This structure is still not fully optimized because some applied data structures are unreasonable. Specifically, LC-tree uses a hash table structure, increasing memory costs due to storing crucial information in the table. Additionally, this tree structure can only perform a 1-level permutation between 2 adjacent nodes and continue recursive swapping until adjustment is complete, increasing the cost of both memory and time.

III. PROPOSED METHOD

To improve memory consumption and execution time, we propose a new tree data structure called ILC-tree, illustrated in Figure 2. We change the link between parent nodes and the list of child nodes from a hash table structure to a single linked list. By adopting this new tree structure, the child nodes are linked together into a single linked list, and the parent node points to the first child node in the list. This change helps reduce memory costs during the tree construction process. The following enhancement involves proposing a method to adjust new nodes by permuting the node that needs adjustment with its group of predecessor nodes. In the worst case, when adjusting a branch with a depth of n itemsets, it only requires $n - 1$ comparisons and a single permutation, compared to the previous method that required $n - 1$ permutations. This proposal helps reduce complexity from $O\left(n^2\right)$ to $O(n)$. In addition to the transition to a linked list structure, our approach incorporates the utilization of the damped window model [16]. This model is instrumental in optimizing the tree structure by gradually removing invalid transactions over time during pattern mining tasks.The changes significantly enhance ILC-Tree efficiency compared to the old LC tree, and will be elaborated on in detail in section 3.1 of this paper.

1. Construction of ILC-Tree

To better understand visually, an example database containing 7 transactions is introduced, as shown in Table I. The TID stores the ID of each transaction, Itemset contains the items in the corresponding transaction, Sum_vitality is the total vitality value of the transaction, Vitality contains information about the vitality value of each item, and Timestamp is the time when the transaction occurs. The larger the timestamp value of a transaction, the newer the transaction is. The larger the vitality value of each item in the tree branch, the higher the vitality of that branch. Regarding the process of building the tree, we divide it into two subtasks: tree restructuring and structure updating. These two tasks will be described respectively in the fast restructuring strategy and the tree structure updating.

TABLE I
EXAMPLE DATASET

TID	Itemset	Sum_vitality	Vitality	Timestamp
1	A, B, C, E, F, G	6	1, 1, 1, 1, 1, 1	50
2	A, C, D, G	4	1, 1, 1, 1	53
3	A, D, H	3	1, 1, 1	55
4	E, F, H, I	4	1, 1, 1, 1	60
5	A, C, E, G	4	1, 1, 1, 1	70
6	I, C, E, F	4	1, 1, 1, 1	75
7	D, A, H, B	4	1, 1, 1, 1	80

a) Fast restructuring strategy

Before each structure update operation, the header table is updated to obtain the order of the most recent item frequencies. The ILC-tree will automatically restructure when the items on the path are not in the order of the current item frequency. Instead of creating a change list of limited items, we will directly compare the items in the current header table with the previous header table to determine the items that need adjustment. Our method will swap the relevant nodes through parent and child node links for each item until all nodes are linked to the traversed list. In summary, the fast ILC-tree restructuring steps are automatically performed in the alternating sequence as follows:

Step 1: Compare the current header table with the previous header table. Starting from the high-frequency item in the list, the Q tree nodes related to the item are accessed through node links.

Step 2: If a parent node R exists with an order after the Q node, move the Q node above the parent node R.

Step 3: Use nodeLink to move to the node P, which has the same name as Q.

Step 4: If P exists, perform step 2. If not, proceed to the next step.

Step 5: Perform steps 2 to 4 until the header table is fully traversed.

During the exchange process, the following properties must be observed:

Property 1: Temporary transaction information is stored in the new leaf node of the transaction. [3]

Property 2: The number of child nodes must not exceed the number of its parent node.[3]

Property 3: A node cannot have a sibling node with the same name.[3]

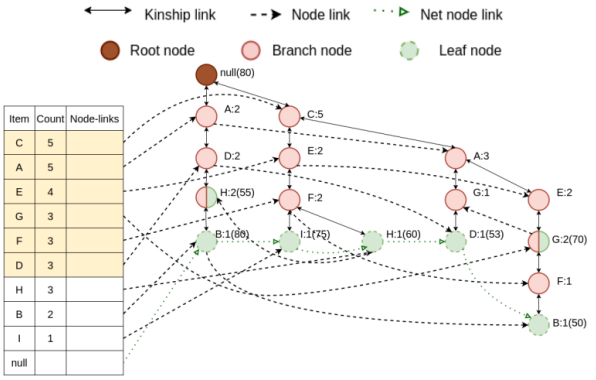


Figure 1. ILC-Tree Structure.

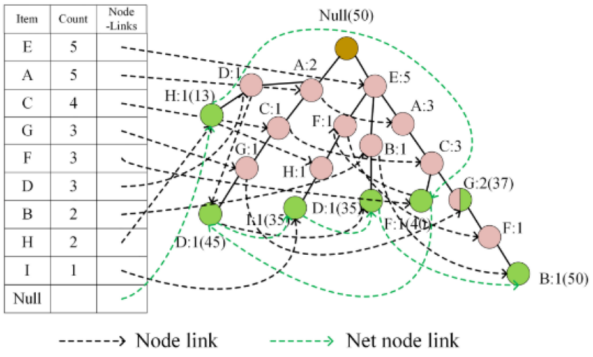


Figure 2. LC-Tree Structure[3]

To understand the node permutation process, we divide it into 3 parts: Separation, permutation, and merging of duplicate nodes. The permutation operation between nodes is only performed when the number of Q nodes and the parent R nodes of Q are equal, meaning $R.count = Q.count$. Therefore, if the number of Q nodes is less than the parent R node, it is necessary to separate from the parent node of Q to the node R. Suppose there is a branch consisting of a group of nodes $P = \{P_0, P_1, P_2, \ldots, P_n\}$, with $n > 0$, where P_0 is the node that needs adjustment, in this case, P_0 is the Q node. P_1 is the parent node of P_0, P_2 is the parent node of P_1, ..., P_n is the parent node of P_{n-1}, and R is P_n. In that case, it is necessary to split the group of nodes P into two branches $P = \{P_0, P_1, P_2, \ldots, P_n\}$ and $P^* = \{P_0^*, P_1^*, P_2^*, \ldots, P_n^*\}$. The quantity of the P^* node group will be updated as follows:

$$P_1^* = P_i.count - P_0.count, \quad for\ i : 1 \rightarrow n$$

The child nodes of the nodes in $P \setminus \{P_0\}$ that do not belong to this node branch will be sequentially assigned as child nodes of the nodes in P^*. For example, with the tree structure in Figure 2, assuming the node that needs adjustment is node B:1(50), its correct position is above the predecessor node E:2, as illustrated in Figure 3.

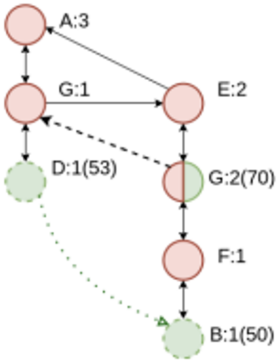


Figure 3. Branch EGFB before splitting

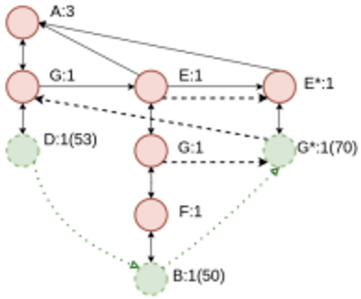


Figure 4. Branch EGFB after splitting

Before splitting, we have the set $P = \{E, G, F, B\}$. After separating, we have the set $P = \{E, G, F, B\}$ and the set $P^* = \{E^*, G^*\}$. The number of nodes in the P node group is equal, so the conditions for permutation are met. It is important to note that if any node P_i is a leaf node of another path, its temporary information will be transferred and stored in the node P_i^* when splitting. In this case, the node G^* is a leaf node, so its temporary information will be transferred from node G to node G^*, and since this leaf node has no child nodes, it will become a pure leaf node and will be linked to other pure leaf nodes.

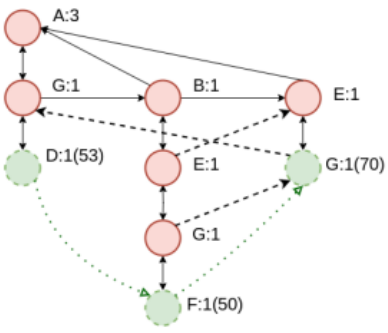


Figure 5. Tree after restructuring

Permutation: Based on the example in Figure 3(b), we will perform the permutation of node B:1(50) with the node group E:1, G:1, F:1. In this case, the order of the branch will change from $A \rightarrow E \rightarrow G \rightarrow F \rightarrow B$ *to* $A \rightarrow B \rightarrow E \rightarrow G \rightarrow F$. After the exchange, the result will be as shown in Figure 4. Since B is a pure leaf node, its old parent node F will become a new pure leaf node, and the temporary information at node B will be transferred to node F.

Merge Duplicate Nodes: If there exists a sibling node Q^* representing the same item as node Q and having the same parent node during the node exchange process, then the quantity and child links of node Q^* are merged into node Q. If Q^* is a leaf node, its temporary information will be updated to Q. If Q^* is a pure leaf node and Q is not, then Q^* will be removed from the pure leaf node linked list. Similarly, if Q is a pure leaf node and Q^* is not, then Q will be removed from the pure leaf node linked list. Afterward, Q^* is removed from the tree. This process is recursively applied to the children of Q until there are no more duplicate nodes.

b) Update Tree Structure

Updating the structure involves adding, deleting, and modifying the existing tree structure. Except for the first transaction data, adding the remaining data can be seen as an update to the existing structure. Modification operations can be considered a combination of addition and removal. This section details the process of adding transactions and deleting transactions.

- Adding Transaction: The process of adding a transaction is illustrated in Algorithm 1. The steps of adding a transaction are automatically performed in an interleaved manner as follows:

- Reordering the header table and transaction: Information about the frequency of items in the scanned transactions is store in a header table. When a new transaction is examined, each item is checked to see if it is already in the header table. If it is, its quantity in the header table is increased by 1. Otherwise, the item is inserted into the header table. After scanning, the header table and transactions are reordered based on the decreasing order of item frequencies.

- Restructuring the ILC-tree: Based on the order of the newly sorted header table, the ILC-tree is updated using the fast-restructuring strategy. It allows traversing through as few nodes as possible during the restructuring process.

- Adding New Transaction to ILC-tree: After restructuring the ILC-tree, the transactions items are rearranged and inserted into the ILC-tree in order. If a node containing an existing item is encountered, the quantity of that node is incremented by 1. Otherwise, a new node containing the new item is created in the ILC-tree and linked to the item with the same name through a nodeLink. The transaction time is stored in the leaf node of the path. If the timestamp value of the root node is less than the timestamp value of the new transaction, then the timestamp value of the root node is updated to the timestamp value of the new transaction.

- Linking Pure Leaf Nodes: When a data operation involves a leaf node, that node is identified as a pure leaf node based on its number of children. If a leaf node has no children, it is identified as a pure leaf node, and the new pure leaf node is linked to the ”null”

Entry in the header table and other pure leaf nodes. If a pure leaf node gains additional children during restructuring or adding a new transaction, its link is removed. Finally, the header table points to the most recently inserted pure leaf node.

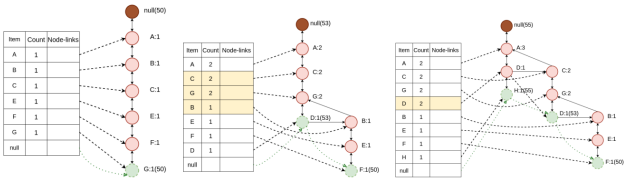


Figure 6. Process of inserting transactions 1 to 3

For example, based on the dataset in Table I, the process of adding a new transaction is illustrated in Figure 6. The set of items covered by the yellow-shaded area in the header table represents the differences between the old and new header tables. These nodes need to be adjusted before inserting new transactions. After adjustment, the last node in the path becomes the new leaf node, and the temporary information of the transaction is stored in the leaf node. The header table points to the newly inserted nodes and the pure leaf node when transactions are inserted. For example, when inserting the second transaction <A, C, D, G>, the header table updates the count of items 'A', 'C', 'G', and creates a new entry for item 'D', as shown in Figure 6. By comparing the updated and original frequencies in the header table, we identify the nodes that need adjustment, which are the nodes containing items 'C', 'G', and 'B'. After adjusting the path, node 'F' becomes the new leaf node containing the timestamp of the first transaction, and the items of the second transaction are inserted into the current ILC-tree.

The 'D' and 'Null' entries in the header table point to the new pure leaf node D and other nodes with the same item name in the tree structure are linked together. Similar processes for inserting transactions follow this example.

Algorithm 1: Add Transaction to ILC-Tree

Input: Transaction dataset D

Output: ILC-tree

Step 1: Create a header table with a key *null* and a root node labeled *null*;

Step 2: Procedure *SortHeaderTableAndTransactions*;

begin

  Read a transaction T from D;

  for *each item I in T* **do**

    if *I exists in the header table* **then**

      Increase the count of I by 1;

    end

    else

      Add I to the header table;

    end

  end

  Sort items in the header table in descending order of frequency;

  Sort each transaction T based on the order of the header table;

end

Step 3: Procedure *Restructure ILC-Tree*;

begin

  Compare the order of items between the old and new header tables;

  Rearrange the paths of the tree based on the links between different items in the two header tables;

  for *each path $P = \{P_0, P_1, \ldots, P_n\}$ belonging to a node branch in the ILC-tree* **do**

    if *the quantity of node P_i is greater than node P_0* **then**

      Split the P node group into two branches $P = \{P_0, P_1, \ldots, P_n\}$ and $P^* = \{P_0^*, P_1^*, \ldots, P_n^*\}$;

      Update the quantity of the P^* node group as $P_j^*.count = P_j.count - P_0.count$ for $j = 1 \rightarrow n$;

    end

  end

  Permute node P_0 and the node group $\{P_1, \ldots, P_i\}$;

  if *the child node W of P_{i+1} has the same name as P_0* **then**

    Merge node W with node P_0, including quantity and temporary information;

  end

end

Step 4: Procedure *Add New Transaction to ILC-Tree*;

begin

  After restructuring, insert the items of T into the ILC-tree;

  if *a node Y containing an item in T already exists* **then**

    Increase the quantity of Y by 1;

  end

  else

    Initialize node Y with a quantity of 1;

    Link it to the header table and linked nodes with the same item;

  end

  Add netNodeLink if it is a pure leaf node;

  Continue until all transactions are inserted into the tree;

end

Deletion of transaction: The ILC-tree algorithm performs the operation of deleting a transaction at a specific time and applies a top-down traversal mechanism to access nodes. The process of deleting a transaction is illustrated in Algorithm 2. For each deleted transaction, the first step is to sort it in descending order of item frequency in the current header table. Then, starting from the first item of the transaction, the count of each item in the header table is reduced by m, where m is the quantity to be deleted, as well as the count of the corresponding node. Nodes with a count of 0 after reduction will be deleted. After all transactions are deleted, the items in the header table are sorted in descending order of frequency, and the paths of the ILC-tree are rearranged based on the updated frequency decreasing order of items.

An example is provided to illustrate the transaction deletion process for the initial tree structure of Table I. Table II is the set of transactions to be deleted, and the process is illustrated in Figure 7. Nodes with a quantity reduced to 0 after reduction are marked with gray dots. After processing all nodes of transactions, the algorithm performs a fast restructuring to maintain the properties of the ILC-tree and improve the compactness of the tree structure.

Algorithm 2: Delete transaction and update the tree

Require: Transaction dataset D, ILC-tree root node, deleted transaction dataset db^- from D

Ensure : Updated ILC-tree $(D - db^-)$

From db^-, read a transaction T at a certain moment, decrease the quantity by 1 for each item i of T;
Sort the items in the header table in descending order, sort each transaction t in db^- based on the order of the updated header table. Set the ordered transaction list as $[m|M]$, where m is the first element of the ordered list, and M is the rest of the list;
Set X as the root node;
for *each child node Y of node X such that* $Y.ItemID = m.ItemID$ **do**
    Decrease the quantity of Y by 1;
    Update X to be Y, the first item of M will be m, and the rest of the items will be M;
    if *Y is a leaf node* **then**
        Remove the temporary information of transaction T;
    end
end
Continue until all transactions in db^- are deleted;
Execute steps 11-20 of Algorithm 1;

TABLE II
THE TRANSACTION SET HAS BEEN DELETED

TID	Itemset	Sum_vitality	Vitality	Timestamp
3	A, D, H	3	1, 1, 1	55
7	D, A, H, B	4	1, 1, 1, 1	80

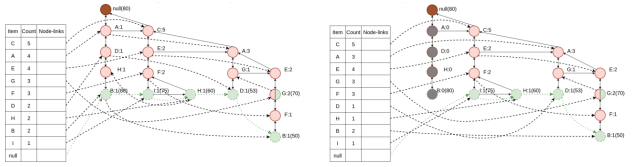


Figure 7. Process of deleting transactions 3 and 7

c) Damped window model for tree optimization

The damped window [16] model is used to optimize the tree structure by removing invalid transactions over time in pattern mining. The model is combined with our ILC-tree structure. First, the model accesses each pure leaf node through the node links linked to the ”null” entry in the header table. Each pure leaf node is analyzed and evaluated based on its time and frequency. When a pure leaf node is identified for deletion, its corresponding tree path will also be deleted. If a new pure leaf node appears during the deletion process, its decayed vitality value needs to be calculated and evaluated. If the decayed vitality value is greater than the $vit_{\min}$ value, then proceed to the next pure leaf node; otherwise, continue deleting the branch with this leaf node. After evaluating all pure leaf nodes in the tree and adjusting the tree structure, the tree structure is optimized. In this process, the model also addresses two main issues:

Determining when a leaf node needs to be deleted: A leaf node may be deleted if it infrequently occurs, and its last occurrence was a long time ago. To quantify this predictive process, the following definitions of node vitality are introduced. Firstly, we determine a value to quantify the timeliness of transactions. The lifespan of an item in a transaction $v(i, t_d)$ is calculated as: $v\,(i, t_d) = u_i$ (6)[16] In which i is an item of the transaction t_d, $i \in t_d$, and u_i is the relative importance, utility, of i. Based on this, the vitality value of the transaction (the vitality of the tree branch) $v(t_d)$ is calculated as: $v\,(t_d) = \sum\ v(i, t_d)/|t_d|$ (7)[16]

Where $|t_d|$ is the number of items in t_d. For example, for the transaction <A, C, D, G> in Table I, where all items have an importance of 1, the vitality value of the transaction is 1.

For each transaction, the value of decayed vitality $v\,(l_d)$ is calculated as follows:

$$v\,(l_d) = \sum\ v(t_d) * f^{T-d} \quad (8)[16]$$

where d is the time of occurrence of each transaction t_d, T is the current time contained in the root node, and f is the decay factor. For example, for the leaf node at transaction 2, the transaction occurrence time is $T = 80$, decay factor $f = 0.99$, then the decayed vitality value of this leaf node is calculated as: $v\left(l_{d(2)}\right) = 1 * 0.99^{80-53} \approx 0.76$. If the decayed vitality value at the leaf node falls below the $vit_{\min}$ threshold, the leaf node is identified as weak, and the related paths and nodes in the tree will be deleted.

Performing the deletion operation and adjusting the ILC-tree: The deletion operation starts from the pure leaf node to be deleted and follows the transmission mechanism from bottom to top. The quantity of the leaf node is marked as N_i. When the pure leaf node is deleted, the corresponding item quantity in the header table will decrease by N_i.

For the remaining nodes of the transaction connected to the deleted node, their quantity will reduce similarly by N_i. The amount of the corresponding item in the header table will decrease by N_i. Nodes with a quantity of 0 after the reduction will be deleted. After the deletion operations are completed, the ILC-tree performs rapid restructuring to adjust the tree structure.

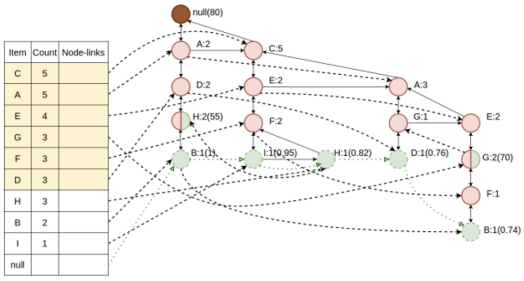
Algorithm 3: Delete transaction and update the tree

Require: Table of headers, root node
Ensure : Updated ILC-tree

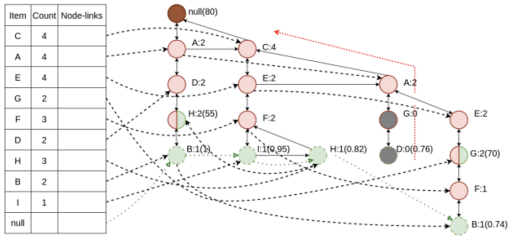
foreach *leaf node X of the ILC-tree* **do**
  Calculate $v\left(l_{d(x)}\right)$ for node X;
  if $v\left(l_{d(x)}\right) < vit_{\min}$ **then**
    Delete node X and set N_i to the number of nodes X;
    foreach *parent node Y of node X* **do**
      Decrease N_i by the number of nodes Y;
      if *the number of nodes* $Y = 0$ **then**
        Delete node Y;
      end
    end
    if *Y is a pure leaf node after deletion* **then**
      Repeat steps 2-14 to evaluate the new pure leaf node;
    end
  end
end

2. Pattern Mining from ILC-tree Structure

After the construction of the ILC, the process of creating subtrees is based on two support thresholds: $minFreq$ and $minRare$. In mining, the ILC-tree applies a backward propagation mechanism, starting from the lowest node



(a) Tree after calculating the decayed vitality value of each pure leaf node



(b) Branch t1 is deleted

Figure 8. Tree optimization process

and moving gradually up to the root node. Subsequently, frequent, and rare patterns for each item are collected based on the configured thresholds. Rare patterns need to satisfy condition 4 in Section 2, and frequent patterns need to satisfy condition 5 in Section 2. The entire pattern mining process is illustrated in Algorithm 4.

Algorithm 4: Pattern Mining from ILC-tree

Require: Header table H, minFreq, minRare
Ensure : Frequent patterns, rare patterns

$result \leftarrow \varnothing$;
for *each item I in the header table H with* $I.support > minRare$ **do**
  Create a set of prefix paths in the ILC-tree that occur with I;
  Build $Tree_{Pattern}$ containing frequent and rare items satisfying minFreq, minRare;
  Create FrequentPattern and RarePattern by appending I to frequent and rare items in $Tree_{Pattern}$;
  $result \leftarrow [FrequentPattern \cup RarePattern]$;
end
return $result$;

With the final tree structure obtained in Figure 7(f), the results for frequent and rare patterns are presented in Table

TABLE III
FREQUENT AND RARE PATTERNS EXTRACTED FROM THE ILC-TREE

Item	Base sample	Subtree	Common sample	Rare sample
A	{C:1}	{C:1}	-	-
E	{C:2}, {C, A:1}	{C:3}, {A:1}	{C, E:3}	-
H	{A:2}, {C, E:1}	{A:2}, {C, E:1}	-	{A, H:2}
F	{C, E:1}, {C, E, H:1}	{C, E:2}, {H:1}	-	{C, E, F:2}
D	{A, H:2}	{A, H:2}	-	{A, H, D:2}

III, where $minFreq = 3$ and $minRare = 2$. In this table, Item E has an empty base pattern, so it is impossible to build a subtree for E. Items G, B, and I do not satisfy the support thresholds, so no subtrees can be constructed for them, resulting in no patterns generated for these items. For Item E, the base patterns include {C: 2} and {C, A: 1}. According to the support thresholds, a frequent pattern is {C, E: 3}, and no rare patterns are generated for Item E. Frequent and rare patterns for the remaining items in the header table are generated similarly.

3. Time Complexity Analysis

The proposed ILC-tree method comprises two stages: updating and optimizing the tree structure. Therefore, the time complexity analysis is performed separately for each phase.

a) Tree Structure Update

Let D be the database with T transactions, I be the total number of items in the database, and L be the length of the most extended transaction. Updating the tree structure is divided into adding transactions and deleting transactions. Adding a transaction involves four steps: rearranging the header table and the transaction, fast restructuring, adding the new transaction to the ILC-tree, and linking the pure leaf nodes.

Adding transactions: The worst-case scenario for the first step is when the length of each transaction equals the number of items I, meaning $L = I$. However, this is the best case for rearranging the header table because there will be no changes in the order. Therefore, the time complexity of the first step is $O(TI^2)$. In the second step, the worst case is when all tree paths need adjustment when the data is inserted. In other words, the total number of swaps equals the total number of items I. Hence, the overall time complexity of the second step is $O(TI^2)$. In the third step, the worst case is when the tree has a maximum depth of $L = I$, and the ILC-tree requires traversing a maximum of I nodes to insert a node. Therefore, the total time complexity of inserting transactions is $O(TI^2)$. In the final step, the worst case is when all T transactions require linking node information, and the total time complexity is $O(T)$.

Deleting transactions: The worst-case scenario for deleting transactions is when all transactions are deleted, and each deletion requires traversing through nodes, resulting in a total time complexity of $O(TI^2)$. Hence, the overall time complexity of updating the tree structure can be asymptotically expressed as $O(TI^2)$.

b) Tree Optimization

Tree optimization includes calculating the decayed vitality value of each pure leaf node, deleting paths, and quick restructuring. The worst case for calculating the degraded vitality value is when the number of paths is T, and the time complexity is $O(T)$. For deleting paths, the time complexity depends on the number of paths in the tree and the depth of the tree. The maximum depth of the tree is $L = I$, and the maximum number of tree paths is T. Therefore, the time complexity is $O(TI^2)$. The total time complexity of quick restructuring is $O(TI^2)$, as the maximum number of swaps required is I. Thus, the overall time complexity of tree optimization can be asymptotically expressed as $O(TI^2)$. In summary, ILC-tree requires $O(TI^2)$ for updating the tree structure and $O(TI^2)$ for optimizing the tree. The total time complexity of building the ILC-tree can be asymptotically expressed as $O(TI^2)$.

IV. EXPERIMENTAL

1. Experimental Environment and Dataset

TABLE IV
CHARACTERISTICS OF THE DATASETS

Dataset	Category	Transaction count	Item count (I)	Average Length (A)
Retail utility timestamp	Synthetic	88,162	16,470	10.30
Ecommerce utility timestamp	Real	14,975	3,468	11.71
Mushroom utility timestamp	Synthetic	8,416	119	23

In this section, we will conduct experiments to verify the effectiveness of the proposed algorithm and compare it with SSP-tree, ISSP-tree, and LC-tree. SSP-tree and ISSP-tree are two methods for mining rare itemset patterns using their proposed structures to store data, while LC-tree utilizes a life-cycle tree structure to optimize the tree-building process compared to the two aforementioned structures. We will perform a comparison evaluating the real-time execution and memory consumption of these

methods on various datasets. We use authentic and synthetic datasets, including Retail utility timestamps, ECommerce utility timestamps, and Mushroom utility timestamps. These datasets are widely used to evaluate performance in pattern mining and can be downloaded from the SPMF - Open-Source Data Mining Library. Table IV shows the characteristics of the datasets, such as the time span, the number of transactions, the number of distinct items, the average length, and the density of an item in a transaction. Datasets with more distinct items incur higher memory and execution time costs. Additionally, datasets with lower item density result in lower chances of item repetition in different transactions, leading to longer structure update times. Therefore, we selected three datasets with high, medium, and low density (19.33%, 0.34%, 0.06%) to evaluate our proposed tree structure objectively compared to other tree structures.

To run the experiments, we divided each dataset into five smaller or equal-sized databases, with each small dataset accounting for approximately 20% of the original dataset. After each transaction is inserted into the tree structure, the algorithm will run the reconstruction function promptly. After inserting each dataset, the algorithm will use the damped window model to optimize the tree once.

TABLE V
BRANCHING DENSITY IN THE DATASET

Dataset	f	$[vit_{\min}]$	$minRare$	$[minFreq]$
Retail utility timestamp	0.9	0.28	800	1000
ECommerce utility timestamp	0.9	27.3	75	110
Mushroom utility timestamp	0.9	2.35	800	1000

All methods build an initial tree structure with temporary information and perform pattern mining. Then, the data structure is updated and optimized every time a supplementary database is inserted, and the pattern mining task is re-executed. Decay factor f, minimum $vit_{\min}$ threshold, and support thresholds are set as in Table V. All methods generate similar rare and common patterns and spend equal amounts of time on the pattern mining process due to the same input data and parameter settings. We evaluate the performance of the methods and the obtained results by taking the average values from multiple runs. All algorithms are implemented in Python and executed on a computer with the following configuration: Ubuntu 18.04.6 LTS, Intel 2.2 GHz CPU, 256 GB RAM.

2. Results

a) Execution Time

From Figure 9 (a), it is evident that our tree structure has a faster execution time compared to the old version of LC-tree and SSP-tree, ISSP-tree, with times of 72.9 hours, 81.4 hours, 87.9 hours, and 83.6 hours, respectively, in the Retail utility timestamp dataset. Figure 9 (b) shows 48.6 minutes, 59.4 minutes, 56.7 minutes, and 56.8 minutes, respectively, in the Ecommerce utility timestamps dataset. The relatively long execution time in the Retail utility timestamp dataset is due to its large and sparse nature, leading to the need to build more nodes. Our ILC-tree has a mechanism for quickly adjusting nodes, thus helping to reduce the reconstruction time.

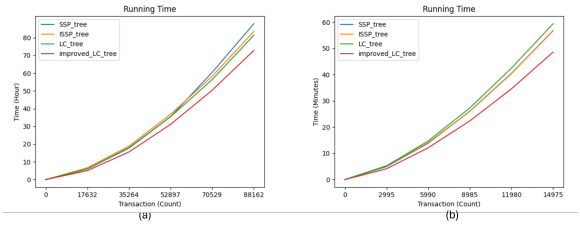


Figure 9. Real-time execution time chart of the ILC-tree structure compared with 3 structures: LC-tree, SSP-tree, and ISSP-tree, on 2 datasets

b) Memory Consumption

In terms of memory consumption, our algorithm stands out in comparison to LC-tree, SSP-tree, and ISSP-tree. In the Retail utility timestamp dataset, depicted in Figure 10 (a), our algorithm exhibits memory footprints of 158.9 MB, 321.7 MB, 540.2 MB, and 402.8 MB. Meanwhile, in the Ecommerce utility timestamps dataset illustrated in Figure 10 (b), the memory footprints are observed to be 30.0 MB, 61.9 MB, 87.3 MB, and 60.8 MB respectively. These figures underscore the superior efficiency of our approach.

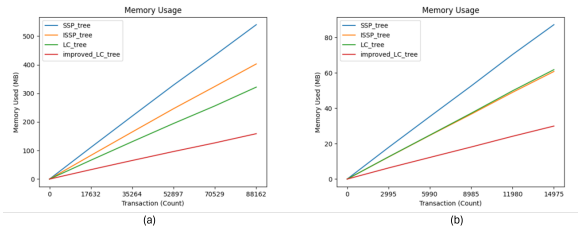


Figure 10. Memory consumption of the ILC-tree structure compared to 3 other structures: LC-tree, SSP-tree, and ISSP-tree, on 2 datasets

Moreover, when directly compared to the legacy LC-tree structure, our ILC-tree demonstrates a remarkable performance, executing approximately twice as fast while conserving more memory than its predecessors SSP-tree and ISSP-tree. This significant enhancement is primarily attributed to the implementation of our novel tree structure, which effectively reduces the memory overhead associated with the list of child nodes at each parent node.

V. CONCLUSION

In this study, we have proposed improvements to optimize execution time and memory consumption compared to the old tree structure. This includes changing the tree structure and using a method to adjust new nodes. For the tree structure, we use a singly linked list to store child nodes instead of a hash table, reducing the storage cost of keys in the hash table. For the tree reconstruction task, our method allows the node that needs adjustment to swap positions with a group of nodes rather than just the parent node, improving execution time, especially for large and densely branched datasets.

In the future, we will explore new algorithms and strategies to optimize the pattern mining process. We will consider using more efficient libraries like Dask or DataTable to handle large-sized files, improving execution time. We will also investigate methods for extracting rules from patterns and continue researching and building new tree structure to find more optimal solutions. Despite the remaining challenges, our proposed enhancements have significant potential for the future, thanks to their processing speed and memory optimization.

REFERENCES

[1] Hipp, Jochen, Ulrich Güntzer, and Gholamreza Nakhaeizadeh, "Algorithms for association rule mining—a general survey and comparison," *ACM SIGKDD Explorations Newsletter*, vol. 2, no. 1, 2000, pp. 58–64.

[2] Abbasi, Ameer Ahmed, and Mohamed Younis. "A survey on clustering algorithms for wireless sensor networks," *Computer Communications*, vol. 30, no. 14-15, 2007, pp. 2826–2841.

[3] Hu, Kerui, Lemiao Qiu, Shuyou Zhang, Zili Wang, and Naiyu Fang, "An incremental rare association rule mining approach with a life cycle tree structure considering time-sensitive data," *Applied Intelligence*, vol. 53, no. 9, 2023, pp. 10800–10824.

[4] Srikant, Ramakrishnan, and Rakesh Agrawal, "Mining generalized association rules," *Future Generation Computer Systems*, vol. 13, no. 2-3, 1997, pp. 161–180.

[5] Haglin, David J., and Anna M. Manning, "On minimal infrequent itemset mining," In *Proceedings of DMIN*, 2007, pp. 141–147.

[6] Yun, Hyunyoon, Danshim Ha, Buhyun Hwang, and Keun Ho Ryu, "Mining association rules on significant rare data using relative support," *Journal of Systems and Software*, vol. 67, no. 3, 2003, pp. 181–191.

[7] Koh, Yun Sing, and Nathan Rountree, "Finding sporadic rules using apriori-inverse," In *Proceedings of the 9th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, Springer, 2005, pp. 97–106.

[8] Szathmary, Laszlo, Amedeo Napoli, and Petko Valtchev, "Towards rare itemset mining," In *Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, vol. 1, 2007, pp. 305–312.

[9] Han, Jiawei, Jian Pei, and Yiwen Yin, "Mining frequent patterns without candidate generation," *ACM SIGMOD Record*, vol. 29, no. 2, 2000, pp. 1–12.

[10] Koh, Jia-Ling, and Shui-Feng Shieh, "An efficient approach for maintaining association rules based on adjusting FP-tree structures," In *Proceedings of the International Conference on Database Systems for Advanced Applications*, Springer, 2004, pp. 417–424.

[11] Hong, Tzung-Pei, Chun-Wei Lin, and Yu-Lung Wu, "Incrementally fast updated frequent pattern trees," *Expert Systems with Applications*, vol. 34, no. 4, 2008, pp. 2424–2435.

[12] Adnan, Muhaimenul, Reda Alhajj, and Ken Barker, "Alternative method for incrementally constructing the FP-tree," In *Proceedings of the 3rd International IEEE Conference on Intelligent Systems*, 2006, pp. 494–499.

[13] Borah, Anindita, and Bhabesh Nath, "Incremental rare pattern based approach for identifying outliers in medical data," *Applied Soft Computing*, vol. 85, 2019, pp. 105824.

[14] Ahmed, Shafiul Alom, and Bhabesh Nath, "ISSP-tree: An improved fast algorithm for constructing a complete prefix tree using single database scan," *Expert Systems with Applications*, vol. 185, 2021, pp. 115603.

[15] Mahdi, Mahmoud A., Khalid M. Hosny, and Ibrahim Elhenawy, "FR-Tree: A novel rare association rule for big data problem," *Expert Systems with Applications*, vol. 187, 2022, pp. 115898.

[16] Yun, Unil, Donggyu Kim, Eunchul Yoon, and Hamido Fujita, "Damped window based high average utility pattern mining over data streams," *Knowledge-Based Systems*, vol. 144, 2018, pp. 188–205.

Dang-Ngo Huu received the B.E. degree in Data Science from Faculty of Information Technology, University of Science in 2023. Research direction: Data Science, Privacy-preserving data mining.
Email: hudn@gmail.com

Khac-Chien Nguyen received his Bachelor's degree in Information Technology from the People's Security Academy in 2003 and received the Master degree in Computer Science from the University of Natural Sciences - Vietnam National University, Ho Chi Minh City in 2009, and his PhD in Communication Engineering from the Post and Telecommunication Institute of Technology Hanoi in 2019. His research interests include: Cloud computing and Data mining.
Email: nkchienster@gmail.com

Nghia Le received the B.E. degree from Faculty of Electronics and Telecommunications, Ho Chi Minh City University of Technology in 1995. In 1998, he received Master degree in Computer Science, Ho Chi Minh City University of Science. His research interests include: Artificial Intelligence in Embedded Systems and Data Security.
Email: nghiale@gmail.com