Journal of Computer Science and Cybernetics, V.36, N.2 (2020), 105–118
DOI 10.15625/1813-9663/36/2/14353

AN EFFICIENT ALGORITHM FOR MINING HIGH UTILITY ASSOCIATION RULES FROM LATTICE

TRINH D.D. NGUYEN[1,*], LOAN T.T. NGUYEN[2,3], QUYEN TRAN[4], BAY VO[5]

[1]*Faculty of Computer Science, University of Information Technology, Ho Chi Minh City, Vietnam*

[2]*School of Computer Science and Engineering, International University, Ho Chi Minh City, Vietnam*

[3]*Vietnam National University, Ho Chi Minh City, Vietnam*

[4]*Informatics Team, Bac Lieu Specialized High School Bac Lieu City, Vietnam*

[5]*Faculty of Information Technology, Ho Chi Minh City University of Technology, Ho Chi Minh City, Vietnam*

Abstract. In business, most of companies focus on growing their profits. Besides considering profit from each product, they also focus on the relationship among products in order to support effective decision making, gain more profits and attract their customers, e.g. shelf arrangement, product displays, or product marketing, etc. Some high utility association rules have been proposed, however, they consume much memory and require long time processing. This paper proposes LHAR (Lattice-based for mining High utility Association Rules) algorithm to mine high utility association rules based on a lattice of high utility itemsets. The LHAR algorithm aims to generate high utility association rules during the process of building lattice of high utility itemsets, and thus it needs less memory and runtime.

Keywords. High utility itemsets; High utility itemset lattice; High utility association rules.

1. INTRODUCTION

The frequent itemset mining (FIM) only supports to find frequent itemsets in transaction database. The problem only considers the appearance of items in each transaction instead of their profit, means that each item has similar utility (profit). In the real world of transaction database, the profits of items are different [18]. For example, in a transaction, customer may buy 10 bottles of water and one bottle of wine, however, the profit from a bottle of wine may be much higher than that of water even the quantity of bottles of water is higher. To solve the problem, high utility itemset mining (HUIM) has been investigated in order to consider the frequent of each item in itemsets as well as their utility value. The result of HUIM has been applied applied to many different fields, e.g. clicks on website, website marketing, retails, medical, etc. [18]. In HUIM, high utility association rules play an important part to consider the relationship among items in database. However, there have not been many researches on high utility association rules. Two algorithms, HGB-HAR (High-utility Generic Basis - High-utility Association Rule) [12] and LARM (Lattice-based Association Rules Miner) [10]

*Corresponding author.
E-mail addresses: trinhndd.ncs@grad.uit.edu.vn (T.D.D.Nguyen); nttloan@hcmiu.edu.vn (L.T.T.Nguyen); tlquyen083@gmail.com (Q.Tran); vd.bay@hutech.edu.vn (B.Vo).

have been proposed. The LARM algorithm has better performance than that of HGB-HAR. However, LARM is based on a two-stages process to generate high utility association rules (HARs), the first stage is to build high utility itemsets lattice, and the second is to generate HARs from the built lattice. Thus, LARM still has longer execution time and consumes more memory. This paper aims to improve the performance of LARM for mining HARs from high utility itemsets lattice (HUIL). The main contributions are as follows:

– Propose LHAR (Mining High utility Association Rules based on building Lattice) algorithm to mine high utility association rules during the processing of building high utility itemsets lattice.

– Carry out experiments on different databases to indicate the efficiency of LHAR algorithm comparing to LARM algorithm. The rest of the paper is organized as follows: Section 2 presents definitions and states the problem of mining high utility association rules. Section 3 collects recent related researches on mining HUIs and HARs. Section 4 discusses new algorithm, LHAR, to mine HARs based on HUIL. Section 5 presents the comparison between LHAR algorithm and LARM [10] algorithm in terms of runtime and memory usage. Section 6 concludes and discusses future works.

2. DEFINITIONS

Definition 2.1. (Transaction database) [10]. Given a finite set of items **I**. A transaction database **D** is a set of finite transactions, $\mathbf{D} = \{T_1, T_2, ..., T_n\}$, in which each transaction T_d is a subset of **I** and has a unique identifier (Transaction identifier - Tid). Each item i_p in T_d is associated to a positive number, called quantity, denoted as $q(i_p, T_d)$. Each item $i_p \in \mathbf{I}$ in T_d has a utility value, denoted as $p(i_p)$.

Table 1. Transaction Database example

TID	Transaction	Unit profit
T_1	$A(4)C(1)E(6)F(2)$	$A(4)C(5)E(1)F(1)$
T_2	$D(1)E(4)F(5)$	$D(2)E(1)F(1)$
T_3	$B(4)D(1)E(5)F(1)$	$B(4)D(2)E(1)F(1)$
T_4	$D(1)E(2)F(6)$	$D(2)E(1)F(1)$
T_5	$A(3)C(1)E(1)$	$A(4)C(5)E(1)$

Table 1 describes an example of transaction database with five transactions $T_1, T_2, ..., T_5$. Considering transaction T_2, it has three items D, E, F with corresponding quantity $1, 4, 5$ and their corresponding utility $2, 1, 1$.

Definition 2.2. (Utility of an item in a transaction) The utility of an item i in a transaction T_d is denoted as $u(i, T_d)$ and is defined as $p(i) \times q(i, T_d)$. For example, the utility of item D in transaction T_2 in the above sample database is $u(D, T_2) = 2 \times 1 = 2$.

Definition 2.3. (The utility of an itemset in a transaction) The utility of an itemset X in a transaction T_c, denoted as $u(X, T_c)$, and is defined as $u(X, T_c) = \sum_{i \in X} u(i, T_c),\ X \subseteq T_c$. For

example, the utility of itemset $X = \{D, E\}$ in T_2 from the above sample database in Table 1 is $u(\{D, E\}, T_2) = u(D, T_2) + u(E, T_2) = 2 + 4 = 6$.

Definition 2.4. (The utility of an itemset in database) The utility of an itemset X in database $\mathbf{D}$ is calculated as the sum utility of X in all transactions containing X, that is $u(X) = \sum_{X \subseteq T_d \wedge T_d \in \mathbf{D}} u(X, T_d)$. The utility of itemset $X = \{E, F\}$ in database $\mathbf{D}$ is $u(X) = 31$.

Definition 2.5. (The support of an itemset in database) The support of itemset X in database $\mathbf{D}$ indicates the frequency of availability of X in $\mathbf{D}$. The support value of X with respect to $\mathbf{D}$ is defined as the proportion of itemsets in a database containing X. The support of $X = \{A, C, E\}$ in the above database is $supp(\{A, C, E\}) = 2/5$ or $supp(\{A, C, E\}) = 2$, in short.

Definition 2.6. (High utility itemset) An itemset X is considered as a high utility itemset if its utility $u(X)$ is no less than a minimun utility threshold ($minUtil$) defined by user ($u(X) \geq minUtil$). Otherwise, X is called a low utility itemset.

Definition 2.7. (Local utility value of an item in an itemset). The local utility value of an item x_i in itemset X, denoted as $luv(x_i, X)$, and is defined by the sum of utility of x_i in all transactions containing X. The formula to calculate $luv(x_i, X)$ is $luv(x_i, X) = \sum_{X \subseteq T_d \wedge T_d \in D} u(x_i, T_d)$. For example, the local utility of $x_i = \{E\}$ in $X = \{E, F\}$ is $luv(x_i, X) = 6 + 4 + 5 + 2 = 17$.

Definition 2.8. (Local utility value of itemset in itemset) The local utility value of itemset X in itemset $Y, X \subseteq Y$, denoted as $luv(X, Y)$, and is defined by the sum of local utility of each item $x_i \in X$ in Y. The formula is described as follows $luv(X, Y) = \sum_{x_i \in X \subseteq Y} luv(x_i, Y)$. For example, $luv(X, Y)$ of X in Y where $X = \{D, E\}$ and $Y = \{D, E, F\}$ (given in Table 1) is $luv(X, Y) = (2 + 2 + 2) + (4 + 5 + 2) = 6 + 11 = 17$.

Definition 2.9. (High utility association rule). A high utility association rule R having the form of $X \rightarrow Y \setminus X$, describes the relationship of two high utility itemsets $X, Y \subseteq \mathbf{I}$, $X \subset Y$. The utility confidence of R, $uconf(R)$, is denoted as $uconf(R) = luv(X, XY)/u(X)$. The association rule R: $X \rightarrow Y$ is called the high utility association rule if $uconf(R)$ is greater than or equal to a minimum utility confidence threshold ($minUconf$) given by user. Otherwise, R is considered as low utility association rule. For instance, $X = \{F[14], E[17]\}$ and itemset $Y = \{D[6], F[12], E[11]\}$, the rule R: $FE \rightarrow D$ (which is the shortened form of R: $FE \rightarrow DFE \setminus FE$) has confident value $uconf(R) = 23/31 \times 100 = 74.19\%$. If $minUconf = 60\%$, then R is considered as high utility association rule.

3. RELATED WORK

3.1. High utility itemset mining

The HUIM problem was first introduced in 2004 by Yao et al. [15] and has since, attracted various researchers recently. HUIM addresses the realistic problem that each item can be occurred more than once in each transaction and has its own utility values. Liu et al. (2005) proposed the Two-Phase algorithm [9], one of the earliest algorithms for mining high utility itemsets. The Two-Phase algorithm presented and applied the definition

of Transaction Utility (TU) and Transaction Weighted Utility (TWU) onto the Apriori algorithm [1] to mine HUIM efficiently and accurately. However, Two-Phase generates a large number of candidates in its first phase by over-estimating the utility of candidates. Besides, it performs multiple database scans and thus consumes a large amount of memory and need long execution time.

The Two-Phase algorithm as said, can find the complete set of HUIs in transaction database, but it still is a computationally expensive algorithm. Thus, several approaches haven been proposed to increase further the performance of HUIM. Le et al. introduced two new algorithms named TWU-Mining [6] and DTWU-Mining [7]. The proposed algorithms aim to reduce the candidates generated when mining for HUI using TWU measure by using the data structures, the IT-Tree [17] and the WIT-Tree [7]. Another algorithm named UP-Growth, which was proposed by Tseng et al. [14], introduced a novel tree structure called UP-Tree, to efficiently mining HUIs. The UP-Growth algorithm consisting of two stages, is based on the FP-Growth algorithm [4] and the down-ward closure property of the Two-Phase algorithm [9]. Tseng et al. proposed four effective strategies for pruning candidates: i) Discarding global unpromising items (DGU); ii) Decreasing global node utilities (DGN); iii) Discarding local unpromising items (DLU); iv) Decreasing local node utilities (DLN). By applying these strategies during the process of building global and local UP-Tree, UP-Growth generates less candidates than the Two-Phase algorithm does. And thus, the runtime of UP-Growth has 1000 times faster than that of Two-Phase. Besides, it also requires less memory than Two-Phase. However, UP-Growth still generates a large number of candidates in its first phase by over-estimating utility of each candidates. Moreover, building and maintaining the UP-Tree structure is computationally expensive. The improved version of UP-Growth, named UP-Growth+, was also proposed by Tseng et al. in 2013 [13]. UP-Growth+ came with two new strategies to optimize further the UP-Tree, called Discarding local unpromising items and their estimated Node Utilities and Decreasing local Node utilities for the Nodes. In 2014, Yun et al. proposed the MU-Growth [16] algorithm to improve the UP-Growth+ algorithm. MU-Growth came with another tree data structure called MIQ-Tree (Maximum Quantity Item Tree). In 2014, Fournier-Viger et al. has introduced a more efficient pruning strategy, named Estimated Utility Co-occurrence Pruning (EUCP) [3], to help speeding up the process of mining HUIs. EUCP makes use of the Estimated Utility Co-occurrence Structure (EUCS) to consider item co-occurrences.

Zida et al. proposed EFIM algorithm [18] for mining HUIs effectively with two new upper bounds on utility: Revised sub-tree utility (SU) and local utility (LU). The author demonstrated that the two proposed upper bounds are tighter than TWU and remaining utility based upper bound. EFIM algorithm also introduced two new strategies, High-utility Database Projection (HDP) and High-utility Transaction Merging (HTM), to reduce the cost of scanning database. Unlike Two-Phase or UP-Growth, EFIM is a single phase algorithm. And by utilising the newly proposed upper bounds and strategies, EFIM has better execution time and consume less memory than previous approaches.

In 2017, Krishnamoorthy make use of all existing pruning techniques, such as TWU-Prune [9], EUCS-Prune [3], U-Prune [8] to develop two more pruning techniques, named LA-prune and C-prune. These pruning strategies were then incorporated into an algorithm called HMiner [5].

As in 2019, an extended version of EFIM was proposed by Nguyen et al. [11], named

iMEFIM, which utilized the P-set data structure to reduce the cost of database scans and thus boost the overall performance of the EFIM algorithm dramatically, and iMEFIM also adapted a new database format to handle dynamic utility values to be able to mine HUIs in real-world databases [11].

3.2. Mining high utility association rules from high utility itemsets

Sahoo et al. proposed the HGB-HAR algorithm [12] for mining HARs from high utility generic basic (HGB). The algorithm consists of three phases: (1) mining high utility closed itemsets (HUCI) and generators; (2) generating high utility generic basic (HGB) association rules; And (3) mining all high utility association rules based on HGB. The HGB-HAR algorithm [12] is one of the first high utility association rule mining algorithm. However, the phase 3 of this approach requires more execution time if the HGB list is large and each rule in HGB contains many items in both antecedent and consequent. In this paper, to address this issue, we propose an algorithm for mining high utility association rules using a lattice.

Mai et al. proposed LARM algorithm [10] for mining HARs from high utility itemsets lattice (HUIL). The algorithm has 2 phases: (1) building a HUIL from the discovered set of high utility itemsets; And (2) mining all high utility association rules (HARs) from HUIL. The LARM algorithm is more efficient compared to HGB-HAR in terms of memory usage and runtime. However, this algorithm has two depth scan processes through ResetLattice and InsertLattice. Besides, the algorithm is only able to generate HARs after having the complete lattice of high utility itemsets.

4. PROPOSED METHOD

Problem statement: Given a transaction database **D**, minimum utility threshold $minUtil$ and minimum confidence threshold $minUconf$. The problem of mining all high utility association rules from database **D** is to generate all association rules, formed from two high utility itemsets having utility value greater than or equal to $minUtil$, and having $uconf(R) \geq minUconf$.

4.1. LHAR (Lattice-based for mining High utility Association Rules) algorithm

In this paper, we propose an efficient approach to mine all high utility association rules based on high utility itemsets lattice. The overall process is consisted of two phases, as follows:

- Phase 1. Mine the complete set of HUIs having utility value greater than or equal to minUtil from database **D**. In this stage, the EFIM algorithm [18] is used, which is the most efficient HUIM algorithm.
- Phase 2. Construct HUIL and mine HARs during the HUIL construction process. This process only requires a single step, compared to the two steps from the LARM algorithm, and thus significantly reduces the overall execution time and memory consumption.

The main contribution of this paper is in Phase 2. In this stage, instead of performing two separated steps, which are constructing the lattice first and then scan the constructed lattice

the discover HARs as in the LARM algorithm does, we group these steps into a single stage. In which, while constructing the HUIL, we directly extract the high-utility association rules from the lattice if the rules satisfy the minUconf threshold. This help significantly reduce the runtime required to mine the complete set of HARs. Evaluation studies have shown that our approach has the execution time outperforming the original LARM algorithm over a thousand-fold and dramatically reduces memory usage, up to half of LARM.

Pseudo-code of our approach is presented in Section 4.2 and is named LHAR. The LHAR algorithm is level-wise and contains two main functions, the *BuildLattice* and the *InsertLattice* functions, where, the *BuildLattice* function is called to construct the HUIL based on the input set of HUIs and a user-specified $minUconf$ threshold. Note that the HUIs were ascending sorted by the number of items in each HUI (called *level*). The *BuildLattice* first initializes the *Root* node of the lattice and the set of discovered rules (*RuleSet*). Then at each level of the lattice, the *InsertLattice* is then called to insert an itemset X into the lattice and to recursively explore subsets of X which are HUIs to directly discover and extract HARs during the construction process, non-HARs are also pruned directly during the HUIL construction. By using this approach, we completely eliminated the need of rescanning the constructed lattice to extract HARs, which is time and memory consuming. Memory usage is now only for storing the discovered rules and the partially constructed HUIL. Section 4.2 presents the LHAR algorithm in details.

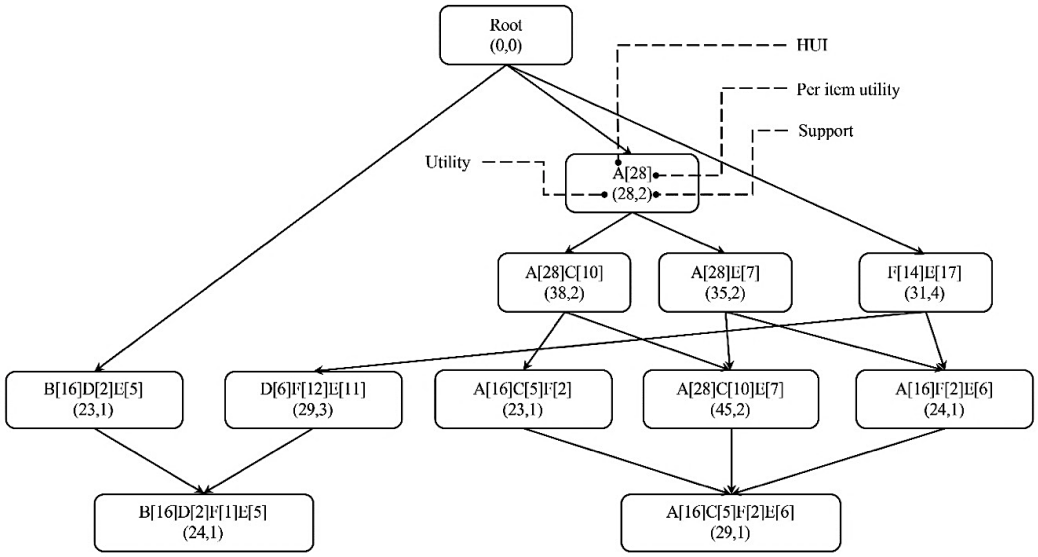

Figure 1. High utility itemsets lattice

The constructed HUI lattice of the sample database in Table 1 is presented in Figure 1. This lattice is similar to that from LARM [12] including a root node and parent-child nodes. The root node is a node containing the empty itemset and has no utility value (or utility equals to 0). Each node (non-root nodes) contains a HUI along with its utility and support value. For instance, considering node $A[28](28, 2)$, the itemset is A, its associated values are $Utility = 28$, $Support = 2$. Node $A[28](28, 2)$ is the parent of node $A[28]C[10](38, 2)$ which contains two items A and C with the corresponding utility values are

$Utility(A) = 28$, $Utility(C) = 10$. The utility value and support of AC are $Utility = 38$, $Support = 2$, respectively. In another words, node $A[28]C[10](38, 2)$ is the child of $A[28](28, 2)$. And $A[28](28, 2)$ has two children, $A[28]C[10](38, 2)$ and $A[28]E[7](35, 2)$.

Figure 1 shows the HUIL constructed from the list of HUIs mined from the sample database with $minUtil$ threshold equals to 23 (25% of the total utility of the transaction database example).

4.2. LHAR algorithm

This section presents the pseudo code of the proposed LHAR algorithm. The inputs of the algorithm are the complete set of discovered HUIs ($TableHUI$), ascending sorted by the number of items, and the user-specified $minUconf$ threshold.

The algorithm returns the complete set of mined HARs from the input and satisfied the $minUconf$ threshold.

LHAR algorithm

```
Input: TableHUI, minUconf
Output: RuleSet;
01: BuildLattice(tableHUI, minUconf)
02:  SET rootNode=∅;
03:  SET RuleSet=∅;
04:  SET Root=new Itemset(0,0);
05:  rootNode.add(Root);
06:  FOR EACH(level in tableHUI.getLevels)
07:    FOR EACH(X in level)
08:      Root.isTraversed=false;
09:      SET resetList=ArrayList of Empty Itemset;
10:      InsertLattice(X, Root, minUconf);
11:      FOR EACH(Y in resetList)
12:        Y.isTraversed=false;
13:      END FOR
14:    END FOR
15:  END FOR
16: END
17: InsertLattice(X, rNode, minUconf)
18:   IF rNode.isTraversed THEN
19:     return;
20:   END IF
21: SET Flag=true, rNode.isTraversed=true;
22: IF X.size>1 THEN
23:   FOR EACH ChildNode IN rNode.ChildNode
24:     IF ChildNode ⊂ X THEN
25:       IF ChildNode.isTraversed=false THEN
26:         resetList.add(ChildNode);
27:         Uconf=R.CalculateConfidence(ChildNode, X);
28:         IF Uconf ≥ minUconf THEN
29:           SET R: ChildNode → X\ChildNode;
30:           RuleSet.add(R);
31:         END IF
32:       END IF
33:       Set Flag=false;
```

```
34:       InsertLattice(X, ChildNode, minUconf);
35:     END IF
36:   END FOR
37: END IF
38: IF Flag THEN
39:   IF X.isTraversed=false THEN
40:     rootNode.add(X);
41:     rNode.ChildNode.add(X);
42:     resetList.add(X);
43:     X.isTraversed=true;
44:   END IF
45:   ELSE
46:     rNode.ChildNode.add(X);
47:   END ELSE
48: END IF
```

This section explains how the LHAR algorithm mines HARs from HUIs.

* Initially, the algorithm triggers $BuildLattice$ method to construct a lattice with $rootNode$: $Root(0, 0)$ and initiates the result collector $RuleSet$ (line 2, 3).

* Next, the algorithm scans HUIs, which were ascending sorted by the number of items ($level$). Considering a HUI $\{X\}$, the flag $isTraversed$ is used to track if $\{X\}$ is traversed ($true$) or not ($false$). $isTraversed$ is initiated for root node $Root(0, 0)$ as $false$. An empty $resetList$ is used at line 9 to handle HUIs which has $isTraversed = true$ during the lattice construction. The algorithm then calls $InsertLattice(X, Root, minUconf)$ to insert $\{X\}$ into $rootNode$ and generate HARs which satisfy $minUconf$ (line 10). Line 11 and 12 is called to reset the flag $isTraversed$ for each HUI in $resetList$ to false after finish processing $InsertLattice(X, Root, minUconf)$ on each node $\{X\}$.

The execution of $InsertLattice(X, rNode, minUconf)$ is as follows.

* It first checks the value of $isTraversed$ on the $rNode$ parameter. If the value is $false$, then the method will perform the following steps set $Flag$ value to $true$. The $Flag$ variable is used to check if $\{X\}$ can be inserted into $rNode$. Set $isTraversed$ of $rNode$ to $true$ to notify that $rNode$ is already processed. $InsertLattice$ is then called recursively to decide which node will be the parent of $\{X\}$.

* Next, the method checks the size of itemset $\{X\}$, if $\{X\}$ has only one item, then it adds $\{X\}$ directly into $rootNode$ (line 38). The steps to add $\{X\}$ into $rNode$ are described from line 38 to 48. If $\{X\}$ does not exist in the $rootNode$ then adds it into lattice as the child of $rootNode$. Otherwise, $\{X\}$ is added into $rNode$. If the size of $\{X\}$ is greater than one, it scans each child node $ChildNode$ of $rNode$. If $ChildNode$ is the child of $\{X\}$ ($ChildNode \subset \{X\}$) then (i) it checks if $isTraversed$ of $ChildNode$ is $false$ in order to add $ChildNode$ into $resetList$ (line 24, 25); (ii) it then considers the rule R: $ChildNode \rightarrow X \setminus ChildNode$ (line 27) and calculate the confidence value $Uconf$ of R, and then add R into $RuleSet$ if $Uconf \geq minUconf$ (line 28); (iii) it recursively calls $InsertLattice$ method to process the insertion of $\{X\}$ into $ChildNode$ (line 34).

4.3. LHAR algorithm illustrations

Consider the sample database given in Table 1, using $minUtil = 23$ and $minUconf = 60\%$. The list of HUIs generated by the EFIM algorithm [18], sorted by levels, are as follows:

- Level-1:
  $\{A[28](28, 2)\}$, denoted as $\{A\}$.
- Level-2:
  $\{A[28]C[10](38, 2)$,
  $A[28]E[7](35, 2)$,
  $F[14]E[17](31, 4)\}$, denoted as $\{AC, AE, FE\}$.
- Level-3:
  $\{B[16]D[2]E[5](23, 1)$,
  $D[6]F[12]E[11](29, 3)$,
  $A[16]C[5]F[2](23, 1)$,
  $A[28]C[10]E[7](45, 2)$,
  $A[16]F[2]E[6](24, 1)\}$ denoted as $\{BDE, DFE, ACF, ACE, AFE\}$.
- Level-4:
  $\{B[16]D[2]F[1]E[5](24, 1)$,
  $A[16]C[5]F[2]E[6](29, 1)\}$, denoted as $\{BDFE, ACFE\}$.

The LHAR algorithm processes the list of HUIs generated by EFIM to construct HUI lattice and mine for HARs:

* Initially, this algorithm declares a lattice with $rootNode$, and defines an empty $RuleSet$.
* It then processes level1 HUIs. Consider $\{X\} = \{A\} \in$ level1. $\{X\}$ is added into $rootNode$. The $RuleSet$ is still empty since no rules were generated.
* Next, considering level2 HUIs. For each $\{X\} \in$ level2, $\{AC\}$ and $\{AE\}$ is then added into $\{A\}$ as children. $\{FE\}$ is added directly into $Root(0, 0)$ since it has no parent which are 1-itemsets. Considering the itemset $\{AC\}$, in which $ChildNode = \{A\}, X = \{AC\}$, and $ChildNode \subset X$, we have found a rule R: $A \rightarrow AC \setminus A \Leftrightarrow R$: $A \rightarrow C$, R has $Uconf(R) = 100\% \geq minUconf$, R is then added into $RuleSet$. Similarly, with $X = \{AE\}$ and $ChildNode = \{A\}$, R: $A \rightarrow AE \setminus A \Leftrightarrow R$: $A \rightarrow E$ is then added into $RuleSet$.
* At level3, considering $X = \{BDE\}, \{DFE\}, \{ACE\}, \{ACF\}$ and $\{AFE\}$, no rules were generated for $X = \{BDE\}$.
  - With $X = \{DFE\}$ we have $ChildNode = \{FE\}$, thus R: $FE \rightarrow DFE \setminus FE \Leftrightarrow R$: $FE \rightarrow D$ is added into the $RuleSet$ since its $Uconf(R) = 74.19\% \geq minUconf$.
  - With $X = \{ACE\}$, $ChildNode = \{A\}$, we have R: $A \rightarrow ACE \setminus A \Leftrightarrow R$: $A \rightarrow CE$, $Uconf(R) = 100\% \geq minUconf$, R is added into $RuleSet$. $InsertLattice$ then recursively processes $ChildNode = \{AC\}$ and $\{AE\}$, we have R: $AC \rightarrow ACE \setminus AC \Leftrightarrow R$: $AC \rightarrow E$, $Uconf(R) = 100\% \geq minUconf$, R is added into $RuleSet$. We also have R: $AE \rightarrow ACE \setminus AE \Leftrightarrow R$: $AE \rightarrow C$, $Uconf(R) = 100\% \geq minUconf$, R is added into $RuleSet$.

Table 2. Discovered HARs from **D** using $minUtil = 23, minUconf = 60\%$

Rules	$Uconf(\%)$	Rules	$Uconf(\%)$	Rules	$Uconf(\%)$
1. $A \rightarrow C$	100	5. $AC \rightarrow E$	100	9. $ACF \rightarrow E$	100
2. $A \rightarrow E$	100	6. $AE \rightarrow C$	100	10. $ACE \rightarrow F$	60
3. $FE \rightarrow D$	74.19	7. $AE \rightarrow F$	62.86	11. $AE \rightarrow CF$	62.86
4. $A \rightarrow CE$	100	8. $BDE \rightarrow F$	100	12. $AFE \rightarrow C$	100

- With $X = \{ACF\}$, $ChildNode = \{A\}$, we have R: $A \rightarrow ACF \setminus A \Leftrightarrow R$: $A \rightarrow CF$, $Uconf(R) = 57.14\% < minUconf$, thus we discard this rule. At this itemset, $InsertLattice$ is then called recursively to process $ChildNode = \{AC\}$, we have R: $AC \rightarrow ACF \setminus AC \Leftrightarrow R$: $AC \rightarrow F$, $Uconf(R) = 55.26\% < minUconf$, thus we discard this rule.

- The remaining itemset is $X = \{AFE\}$, $ChildNode = \{A\}$, we have R: $A \rightarrow AFE \setminus A \Leftrightarrow R$: $A \rightarrow FE$, $Uconf(R) = 57.14\% < minUconf$, R is discarded. $InsertLattice$ then processes recursively to $ChildNode = \{AE\}$ and $\{FE\}$. With $ChildNode = \{AE\}$, we have R: $AE \rightarrow AFE \setminus AE \Leftrightarrow R$: $AE \rightarrow F$, $Uconf(R) = 62.86\% \geq minUconf$, R is added into $RuleSet$. With $ChildNode = \{FE\}$, we have R: $FE \rightarrow AFE \setminus FE \Leftrightarrow R$: $FE \rightarrow C$, $Uconf(R) = 25.81\% < minUconf$, R is then discarded.

* The process continues similarly with level-4 HUIs, which are $\{BDFE\}$ and $\{ACFE\}$. The HARs found at this level are $BDE \rightarrow F, ACF \rightarrow E, ACE \rightarrow F, AE \rightarrow CF$ and $AFE \rightarrow C$. The discarded rules are $DFE \rightarrow B, AC \rightarrow FE$ and $FE \rightarrow AC$ with the $Uconf(R) = \{27.59\%, 55.25\%, 25.81\%\}$, respectively.

The results of the algorithm are presented in Table 2 in the order of discovery, including the discovered rules and the associated $Uconf(R)$ values.

4.4. The advantages of LHAR algorithm

LHAR algorithm has the following improvements compared to the LARM algorithm [10], which helps increase the performance of the algorithm in terms of runtime and memory usage.

* LHAR constructs a lattice of high utility itemsets with $rootNode$ then apply a single depth scan by $InsertLattice$, while LARM does the process through two separated methods $ResetLattice$ and $InsertLattice$. The method $ResetLattice$ requires a similar amount of execution time to $InsertLattice$.

* LHAR combines the process of building lattice and generating HARs into one process. It bypasses the method $FindHuiRulesFromLattice$ from the LARM algorithm. As a result, LHAR has better runtime and consumes less memory.

Table 3. Test datasets and their characteristics

Dataset	$N°$ trans	$N°$ items	Total utility	Size (KB)
Chess	3,196	75	2,156,659	642
Mushroom	8,124	119	3,413,720	1,064
Accidents	340,183	468	196,141,636	64,686

Table 4. The number of HUIs and HARs discovered from test datasets

			$N°$ HARs $minUConf$		
Dataset	$minUtil\%$	$N°$ HUIs	40%	60%	80%
Chess	24.5	9,740	1,803,478	1,691,473	593,668
	25.5	4,226	490,292	476,465	200,900
	26.5	1,911	132,873	132,250	703,86
	27.5	791	30,726	30,726	22,211
Mushroom	10	707,250	700,455	679,987	594,178
	11	5,800	281,150	279,574	255,553
	12	2,726	78,308	78,308	74,688
	13	1,152	19,606	19,606	19,474
Accidents	10	7,479	729,209	422,415	100,614
	11	2,367	131,644	88,388	23,911
	12	728	22,510	17,778	5,568
	13	189	2,623	2,453	1,024

5. EXPERIMENTAL STUDIES

5.1. Datasets and experimental environment

We used the datasets from an open-source website SPMF by Fournier [2]: https://bit.ly/2y77RGI. These datasets have been used in many publications in the fields of data mining, high utility itemset mining and high utility association rule mining. The attributes of these datasets are described in UCI Machine Learning Repository at: https://bit.ly/39lh4YX. Table 3 shows characteristics of the datasets used in our tests.

The LARM and LHAR algorithm were all developed using Java. The algorithms were experimented on a computer with the configuration as follows: Intel® Core™ i7-8550U processor, clocked at 1.80GHz, 8 GB of RAM, and running Windows 10 Professional 64-bit. The number of HUIs and HARs mined from relevant datasets are presented in Table 4.

5.2. Comparison on runtime and memory usage between LARM algorithm and LHAR algorithm

We thoroughly analyze the performance between of LHAR algorithm and LARM algorithm on different datasets, and the minUconf threshold was fixed at 60% on all the datasets. In general, the running time and memory consumption of the LHAR algorithm are significantly better than those of the LARM algorithm [10] (Figures 2 to 4).

In the Chess dataset evaluations, it can be seen that the execution time of LHAR has

a major speed boost (Figure 2), which is up to 1400 times faster than LARM, it took only almost 4 seconds for LHAR to finish the task at $minUtil = 24.5\%$ while LARM needs an hour and a half on the test computer to complete. This is the biggest difference in runtime between LHAR and LARM in our studies. For memory usage on the Chess dataset (Figure 2), LHAR reduces the memory needed by half on all minUtil thresholds, LHAR requires the maximum amount 550MB of memory at $minUtil = 24.5\%$ while LARM needed over 1GB of memory.

The execution time of LHAR on the Mushroom dataset (Figure 3) is also lower than that of LARM with the speed up factor is approximately 33 times at $minUtil = 10\%$. As the minimum utility threshold decrease from 13% down to 10%, the increasing in the runtime of LHAR is almost linear while LARM has a sharp increase here. And for the memory usage comparison, the same thing as on the Chess dataset, the memory utilization of LHAR on Mushroom is better than LARM (Figure 3) on all thresholds tested.

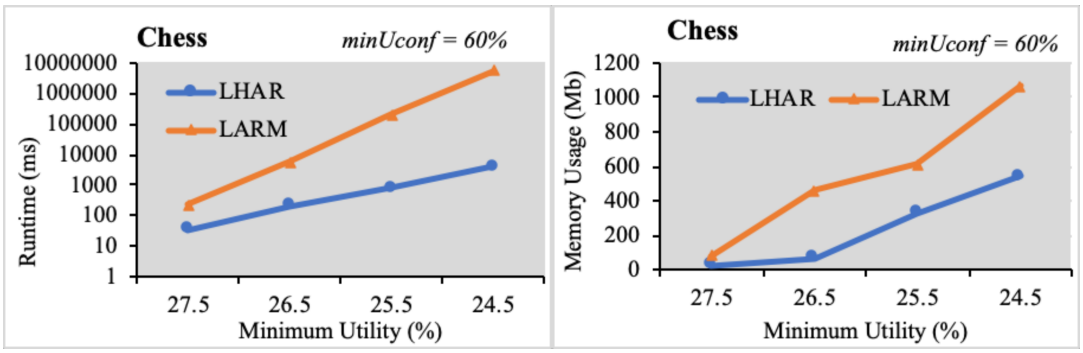

Figure 2. Runtime and memory comparison on Chess dataset

We repeated our process, this time on the Accidents dataset. In this test, the speed up factor is approximately 520 times at $minUtil = 10\%$ (Figure 4) and is also almost linear. For memory consumption, which is also shown in Figure 4, LHAR is still a winner here with twice the times lower memory usage than LARM, with the maximum value at 320MB when compared to almost 640MB of LARM at the same $minUtil$.

Through out the evaluation studies, it can be seen that the LHAR algorithm has superior performance in both runtime and memory utilization when compared to that of LARM, with the speed up factor is up to 1400 times and memory usage is two times lower than LARM.

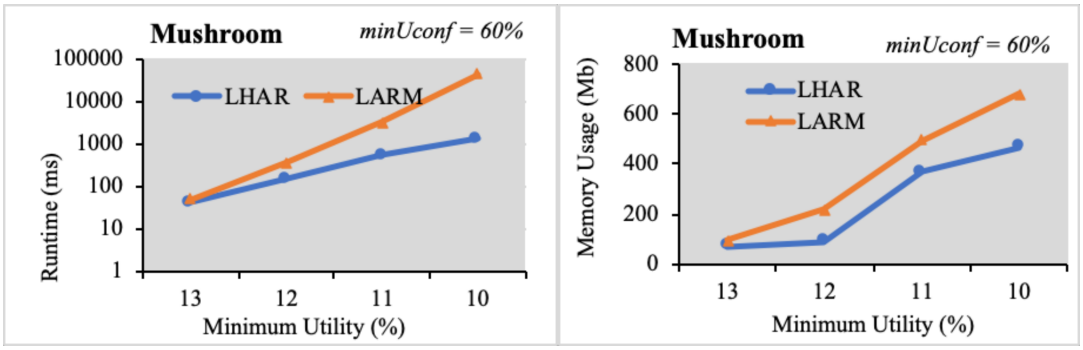

Figure 3. Runtime and memory comparison on Mushroom dataset

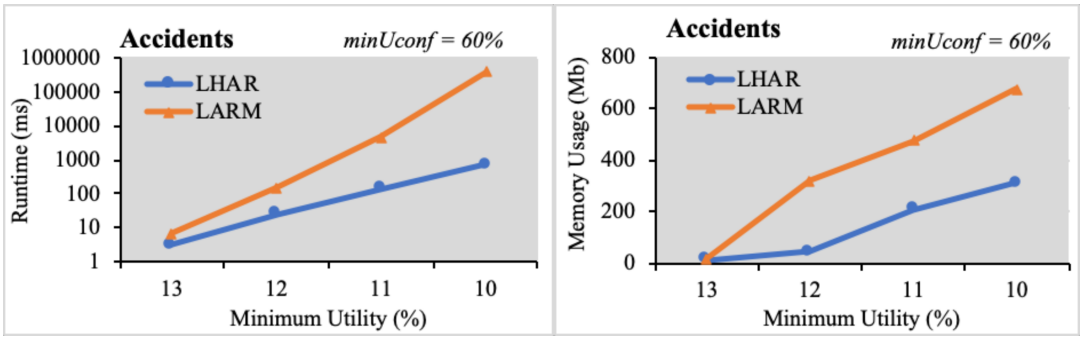

Figure 4. Runtime and memory comparison on Accidents dataset

The lower minUtil threshold, the higher speed up factor. This also shows that the increasing in execution time of LHAR is almost linear when we dropped the *minUtil* threshold on all the tests.

6. CONCLUSIONS

Based on the research of mining HARs from HUIL in LARM algorithm [10], we proposed an improvement of LARM via our algorithm LHAR, in which mines HARs during HUIL construction progress, aims to reduce the algorithm execution time and memory consumption. We conducted variety of experiments on standard databases to firm that LHAR is more efficient than LARM in terms of runtime and memory usage. LHAR algorithm is useful for decision systems and management boards in many fields, e.g., business, education, medical, stocks, etc. This approach can be extended further to mine low high utility association rules, which has tentative support for organization to improve their business activities.

ACKNOWLEDGMENT

This research is funded by Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.05-2018.01.

REFERENCES

[1] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994, pp. 487–499.

[2] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, C.-W. Wu, and V. S. Tseng, "SPMF: A java open-source pattern mining library," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3389–3393, jan 2014.

[3] P. Fournier-Viger, C. W. Wu, S. Zida, and V. S. Tseng, "FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning," *International Symposium on Methodologies for Intelligent Systems*, vol. 8502 LNAI, pp. 83–92, 2014.

[4] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," *Data Mining and Knowledge Discovery*, vol. 8, no. 1, pp. 53–87, 2004.

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

[6] B. Le, H. Nguyen, T. Cao, and B. Vo, "A novel algorithm for mining high utility itemsets," in *Proceedings of 2009 1st Asian Conference on Intelligent Information and Database Systems, ACIIDS 2009*, 2009, pp. 13–17.

[7] B. Le, H. Nguyen, and B. Vo, "An efficient strategy for mining high utility itemsets," *Proceedings of International Journal of Intelligent Information and Database Systems*, vol. 5, pp. 164–176, 2011.

[8] M. Liu and J.-F. Qu, "Mining high utility itemsets without candidate generation," in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*. ACM, 2012, pp. 55–64.

[9] Y. Liu, W.-k. Liao, and A. Choudhary, "A two-phase algorithm for fast discovery of high utility itemsets," in *Proceedings of the 9th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, ser. PAKDD'05. Springer-Verlag, 2005, pp. 689–695.

[10] T. Mai, B. Vo, and L. T. Nguyen, "A lattice-based approach for mining high utility association rules," *Information Sciences*, vol. 399, 2017.

[11] L. T. Nguyen, P. Nguyen, T. D. Nguyen, B. Vo, P. Fournier-Viger, and V. S. Tseng, "Mining high-utility itemsets in dynamic profit databases," *Knowledge-Based Systems*, vol. 175, pp. 130–144, 2019.

[12] J. Sahoo, A. K. Das, and A. Goswami, "An efficient approach for mining association rules from high utility itemsets," *Expert Systems with Applications*, vol. 42, 2015.

[13] V. S. Tseng, B.-E. Shie, C.-W. Wu, and P. S. Yu, "Efficient algorithms for mining high utility itemsets from transactional databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 8, pp. 1772–1786, 2013.

[14] V. S. Tseng, C.-W. Wu, B.-E. Shie, and P. S. Yu, "UP-Growth: An efficient algorithm for high utility itemset mining," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2010, pp. 253–262.

[15] H. Yao, H. Hamilton, and C. Butz, "A foundational approach to mining itemset utilities from databases," in *Proceedings of the Fourth SIAM International Conference on Data Mining*, vol. 4, 2004, pp. 22–24.

[16] U. Yun, H. Ryang, and K. Ryu, "High utility itemset mining with techniques for reducing overestimated utilities and pruning candidates," *Expert Systems with Applications*, vol. 41, pp. 3861–3878, 2014.

[17] M. Zaki, "Scalable algorithms for association mining," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 3, pp. 372–390, may 2000. [Online]. Available: http://dx.doi.org/10.1109/69.846291

[18] S. Zida, P. Fournier-Viger, 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, 2016.

Received on August 25, 2019
Revised on March 18, 2020