An efficient approach for mining closed high utility itemsets and generators

Thang Mai and Loan T. T. Nguyen

ABSTRACT
Mining closed high utility itemsets (CHUIs) serves as a compact and lossless representation of high utility itemsets (HUIs). CHUIs and their generators are useful in analytical and recommendation systems. In this paper, we introduce a lattice approach to extract CHUIs and their generators from a set of HUIs quickly. The experimental results show that mining CHUIs and their generators from a lattice of HUIs is efficient in both runtime and memory usage.

ARTICLE HISTORY
Received 1 November 2016
Accepted 28 February 2017

KEYWORDS
High utility itemset; closed high utility itemset; high utility generator; data mining; lattices

Introduction
The traditional association rule mining (Agrawal, Imielinski, & Swami, 1993) only considers the presence of items in a transaction, and the utility of items is not observed. Many investigations of high utility itemset mining (HUIM) have been published, such as the incremental mining algorithm based on the concept of the fast-update approach for efficiently extracting high utility itemsets (HUIs) (Lin, Lan, Hong, 2012), mining HUIs from vertically distributed databases (Vo, Nguyen, & Le, 2009), the HUI-Miner algorithm with a utility-list structure to store both utility and heuristic information for pruning the search space (Liu & Qu, 2012), the Efficient high-utility Itemset Mining (EFIM) algorithm using two upper bounds called sub-tree utility and local utility for search-space pruning (Zida, Fournier-Viger, Lin, Wu, & Tseng, 2016), mining top-k HUIs (Tseng, Wu, Fournier-Viger, & Yu, 2016), the UP-Growth algorithm using the UP-Tree data structure for candidate pruning (Tseng, Wu, Shie, & Yu, 2010), the UP-Growth+ algorithm (Tseng, Wu, Shie, & Yu, 2013), mining HUIs with multiple minimum utility thresholds (Gan, Lin, Fournier-Viger, & Chao, 2016) and the Two-Phase algorithm for mining HUIs with effective performance (Liu, Liao, & Choudhary, 2005).

The HUIM tasks are more difficult than that of frequent itemset mining (FIM) (Agrawal & Srikant, 1994). In HUIM, the utility of an itemset is neither monotonic nor anti-monotonic, and thus a HUI may have a superset or subset with lower, equal or higher utility. Therefore, we cannot apply some techniques on pruning the search space which have been developed in FIM into HUIM, and thus many recent algorithms have been developed focusing...
on mining HUIs, especially on candidate elimination (Liu et al., 2005; Liu & Qu, 2012; Tseng et al., 2013; Zida et al., 2016).

The concept of closed HUIs (CHUIs) extends the concept of closed patterns from FIM presented by Wang, Han, and Li (2007). A CHUI is a HUI having no proper supersets which are HUIs and exists in the same transactions set (Tseng, Wu, Fournier-Viger, & Yu, 2015). In HUIM, there have been many investigations on extracting CHUIs and generators (Fournier-Viger, Gomariz, Soltani, & Gueniche, 2014; Sahoo, Das, & Goswami, 2015; Wu, Fournier-Viger, Gu, & Tseng, 2015).

In this paper, we address the need of CHUIs and their high utility generators (HUGs) as well as propose an approach to mine CHUIs and their HUGs from HUIs by using a HUIs semi-lattice structure. The main contributions in this paper are as follows:

- We propose an algorithm for building a semi-lattice of high utility itemsets (HUIL) from mined HUIs.
- We propose an algorithm to generate all CHUIs and their generators from a HUIL.
- We carry out experiments to examine the efficiency of the proposed method.

In the rest of this paper, we present some current works related to mining HUIs, CHUIs and generators. We also introduce some basic definitions and the problem statement. We then propose the HUIL algorithm to construct a HUIL, which includes IsClosed and IsGenerator flags from a HUI list. Based on the HUIL structure, we propose the LHUCI-Miner algorithm to extract all CHUIs and their HUGs. To examine the proposed algorithm's problem, we carry out the experimental results with regard to both runtime and memory usage. Finally, we discuss conclusions and some directions for future works.

Related works

High utility itemset mining

The traditional association rules (Agrawal et al., 1993) do not consider the combination between the weight of an item and its quantity. HUIM helps to solve these problems. In HUIM, a transaction can contain multiple same items; moreover, each item has its weight (e.g. profit or utility).

Several studies on HUIM have been carried out. Liu et al. (2005) proposed a Two-Phase algorithm to discover all HUIs efficiently with search-space optimization. However, candidates are generated and it needs to scan database multiple times.

To solve the issue of huge number of candidates being generated, Li, Yeh, and Chang (2008) proposed an isolated items discarding strategy. To improve the process of mining HUIs and reduce the number of database scans, Ahmed, Tanbeer, Jeong, and Lee (2009) proposed a tree-based algorithm named IHUP. Beside that, Tseng et al. (2010) also proposed the UP-Growth algorithm for mining HUIs. Then, Tseng et al. (2013) improved the UP-Growth algorithm and proposed UP-Growth+ algorithm to reduce the overestimated utilities.

HUI-Miner was proposed by Liu and Qu (2012) to discover HUIs with a new utility-list data structure. For mining HUIs, each utility list for an itemset handles the information of transaction identifications for all of the transactions containing the itemset, the utility values of the itemset in the transactions and the sum of the utilities of remaining items that can be included in super itemsets of the itemset in the transactions.
To speed up the runtime and reduce memory consumption in the HUIM process, Lan, Hong, and Tseng (2014) introduced an efficient approach that adopts a projection-based indexing mechanism that directly extracts the result from the given database.

Fournier-Viger, Wu, Zida, and Tseng (2014) proposed the Fast High-Utility Miner algorithm to extract all HUIs from a transaction database. While this is considered as an efficient approach, it generates a large number of HUIs, and this algorithm requires much space for processing as well as fails to terminate.

Sahoo et al. (2015) proposed a Fast High Utility Itemset Miner (FHIM) and defined the promising utility co-occurrence structure to further reduce the number of candidates. Zida et al. (2016) proposed the EFIM algorithm for fast HUIM. EFIM calculates the remaining utility (called the sub-tree utility) at the parent node rather than at child nodes in the depth-first search, and thus it can prune more nodes in the search space.

**CHUIs and generators mining**

A CHUI is a HUI having no proper supersets that are HUIs and appear in the same number of transactions. A generator is also an itemset that has no proper subset having the same support. Fournier-Viger, Wu, and Tseng (2014) investigated the concept of applying generators to HUIM and proposed two new concise representations of HUIs, called HUGs and generator of HUIs (GHUIs). Two new algorithms, HUG-Miner and GHUI-Miner, were also proposed for mining generators. Overall, these are efficient approaches for extracting CHUIs and generators directly from a database.

Wu et al. (2015) provided a solution to mine Closed+ High Utility Itemsets, compact and lossless representations of HUIs. This approach helps to resolve the issue of large numbers of candidates being generated, and thus has better performance in both runtime and memory consumption.

Sahoo et al. (2015) investigated integrating the closure property into HUIM and proposed the HUCI-Miner algorithm to mine all CHUIs and their HUGs from HUIs. This is one of the first studies on extracting CHUIs and HUGs on the mined HUIs from a database. However, the overall process still has performance issues when mining CHUIs and generators from a huge set of HUIs.

**Lattice-based approach in data mining**

Davey and Priestley (1990) introduced the concepts of lattice, order to this context as well as the concept of the join and the meet. An ordered set \((L, \leq)\) is a lattice if the join \(x \lor y\) and the meet \(x \land y\) always exist for any two elements \(x\) and \(y\) in \(L\). \(L\) is called a join semi-lattice if only the join exists. \(L\) is called a meet semi-lattice if there is only the meet available.

In FIM, the set of all frequent itemsets is only a meet semi-lattice. For any two itemsets, only their meet is guaranteed to be frequent, while their join may or may not be. Similarly, in HUIM, the set of itemsets is also a meet semi-lattice. For any pair of itemsets, their join and meet may or may not be HUIs.

In FIM, there are many studies based on the concept of lattice. Zaki and Hsiao (2005) proposed a lattice approach to mine closed itemset. Vo and Le (2011) proposed the Lattice-FI algorithm for quickly mining closed frequent itemsets and generators, and
MNAR_Lattice for fast generating minimal non-redundant association rules from the lattice. Vo, Hong, and Le (2013) proposed a lattice-based approach for mining most generalization association rules.

Definitions and problem statement

Definitions

Definition 1: Given a finite set of items \( I = \{i_1, i_2, \ldots, i_m\} \). Each item \( i_p \) (\( 1 \leq p \leq m \)) has a unit profit \( p(i_p) \). An itemset \( X \) is a set of distinct \( k \) items \( \{i_1, i_2, \ldots, i_k\} \), where \( i_j \in I, 1 \leq j \leq k \), \( k \) is the size of itemset \( X \). A transaction database \( D = \{T_1, T_2, \ldots, T_n\} \) is a set of transactions where each transaction \( T_d \) (\( 1 \leq d \leq n \)) has an unique identifier \( id \), called \( Tid \). Each item \( i_p \) in the transaction \( T_d \) is associated with a weight indicator called quantity \( q(i_p, T_d) \), which is the purchased number of item \( i_p \). An example transaction database is shown in Tables 1 and 2.

Definition 2: The utility of an item \( i \) in a transaction \( T_d \) is denoted as \( u(i, T_d) \) and defined as \( p(i) \times q(i, T_d) \) if \( i \in T_d \). For example, the utility of item \( A \) in \( t_5 \) from Tables 1 and 2 is \( u(A, t_5) = 3 \times 3 = 9 \).

Definition 3: The utility of an itemset \( X \) in a transaction \( T_d \) is denoted as \( u(X, T_d) \) and defined as

\[
u(X, T_d) = \sum_{x_i \in X} u(x_i, T_d).
\]

Definition 4: The utility of an itemset \( X \) in database \( D \) is calculated by the total utility of \( X \) in all transactions that \( X \) belongs to.

\[
u(X) = \sum_{X \subseteq T_d \wedge T_d \in D} u(X, T_d).
\]

Definition 5: The support of an itemset \( X \) is an indication of how frequently that \( X \) appears in database \( D \). The support value of \( X \) with respect to \( T \) is defined as the proportion of itemsets in a database containing \( X \), denoted as \( supp(X) \). For example, the support of \( X = \{A, C, E\} \) in database \( D \) from Table 1 is \( 2/9 \).

Definition 6: The transaction utility (TU) of a transaction \( T_d \) is the sum of the utility of all the items in \( T_d \). For example, \( TU(t_2) = u(D) + u(E) + u(F) = 2 + 4 + 5 = 11 \).

\[
TU(T_d) = \sum_{x \in T_d} u(x, T_d).
\]

Table 1. An example of transaction database \( D \).

| \( Tid \) | Transaction     |
|----------|----------------|
| \( t_1 \) | A(4), C(1), E(6), F(2) |
| \( t_2 \) | D(1), E(4), F(5) |
| \( t_3 \) | B(4), D(1), E(5), F(1) |
| \( t_4 \) | D(1), E(2), F(6) |
| \( t_5 \) | A(3), C(1), E(1) |
| \( t_6 \) | B(1), F(2), H(1) |
| \( t_7 \) | D(1), E(1), F(4), G(1), H(1) |
| \( t_8 \) | D(7), E(3) |
| \( t_9 \) | G(10) |
Definition 7: The transaction-weighted utilization (TWU) of an itemset \( X \) is defined as the sum of the transaction utility of transactions containing \( X \). For example, \( \text{TWU} (A) = TU (t_1) + TU (t_3) = 26 + 15 = 41 \).

\[
\text{TWU} (X) = \sum_{X \subseteq T_d} TU (T_d).
\]

Definition 8: An itemset \( X \) is a HUI if it has a utility equal or larger than the minimum user-specified utility threshold. If an itemset \( X \) has a utility lower than the minimum utility threshold, it is not a HUI; in other words, it is a low utility itemset.

Definition 9: An itemset \( Y \) is called as closure of itemset \( X \) if there are no greater supersets of \( X \) than \( Y \) that \( \text{supp}(X) = \text{supp}(Y) \), denoted as \( \gamma(X) \). An itemset \( X \) is a CHUI if \( X = \gamma(X) \) and \( u(X) \geq \text{min-util} \).

Definition 10: An itemset \( X \) is called a HUI generator or HUG or generator if it is a HUI and no any subset \( Z \) of \( X \) has \( \text{supp}(X) = \text{supp}(Z) \).

Definition 11: The local utility value of an item \( x_i \) in itemset \( X \), which is denoted as \( \text{luv}(x_i, X) \), is calculated by the total utility of \( x_i \) in all records containing \( X \):

\[
\text{luv}(x_i, X) = \sum_{X \subseteq T_d} u(x_i, T_d).
\]

Definition 12: Let \( X = \{i_1, i_2, \ldots, i_n\} \) be an itemset, then the utility unit array of \( X \) is defined as \( U(X) = \{u_1, u_2, \ldots, u_n\} \), where \( u_i = \text{luv}(i_j, X) \), \( j \in \{1, 2, \ldots, n\} \).

Definition 13: The local utility value of an itemset \( X \) in itemset \( Y \), \( X \subseteq Y \), is denoted as \( \text{luv}(X, Y) \) and defined by the sum of local utility values of each item \( x_i \in X \) in itemset \( Y \), as follows:

\[
\text{luv}(X, Y) = \sum_{X \subseteq Y} \text{luv}(x_i, Y).
\]

**Problem statement**

Given a set of HUIs which are mined from transaction database \( D \) with a user-specified minimum utility threshold (\( \text{min-util} \)), the problem statement is to discover all CHUIs and their HUGs from HUIs.
**Mining CHUIs and generators**

*Constructing a lattice of HUIs*

In this section, we refer to a semi-lattice as a lattice. We propose an approach of using a lattice structure to build a lattice of HUIs where each node contains a HUI, IsClosed flag and IsGenerator flag, and we call this lattice HUIL. From this HUIL, we can quickly extract useful information about CHUIs and HUGs.

**HUIL structure**

The HUIL structure contains a root node, which is an empty itemset with utility and support equal to 0, child nodes and connections between each pair of nodes. The connections between each pair of nodes are used to specify their parent–child relationships. Each node contains information about the itemset: utility, support, IsClosed flag and IsGenerator flag. The IsClosed flag in a lattice node indicates that the itemset in this node is a CHUI if its value is true. The IsGenerator flag indicates that the itemset in a lattice node is a generator if its value is true. The name of each node is formed based on the collection of items in an itemset.

For example, in Figure 1, Root is the root node, B is a node with utility = 20 and support = 2, and B node has two child nodes {BE, BF}, B is not a CHUI but a generator.

![Figure 1. HUIL from the database in Tables 1 and 2.](image-url)
Algorithm for constructing HUIL

Algorithm 1. HUIL (HUIs)
Input: Set of HUIs sorted by number of items (level) ascending (HUIs)
Output: HUIL with root node (rootNode)

SetupLattice()
1. Set rootNode = ∅
2. For j = 1 to HUIs.Levels.Count do
3. For each X in HUIs.Levels[j] do
4. For each childNode in rootNode.Children do
5. If childNode.Itemset ⊆ X then
6. Set childNode.IsTraversed = False
7. ResetFlagsOnLatticeNodes(childNode)
8. End
9. End
10. PushNodeToLattice (HUI, rootNode)
11. End
12. End

ResetFlagsOnLatticeNode (Ln)
13. Set Ln.IsTraversed = False
14. For each Lc in Ln.Children do
15. ResetFlagsOnLatticeNodes(Lc)
16. End
17. PushNodesToLattice (X, rootNode)
18. If rootNode.IsTraversed then
19. Return
20. End
21. Set Flag = True, rootNode.IsTraversed = True
22. For each childNode in rootNode.Children do
23. If childNode ⊆ X then
24. Set Flag = False
25. PushNodeToLattice (X, childNode)
26. End
27. End
28. If Flag = True then
29. nodeRoot.Children.Add(X)
30. If rootNode.Support = X.Support then
31. Set X.IsGenerator = False
32. Set rootNode.IsClosed = False
33. End
34. End

Firstly, the algorithm calls the SetupLattice procedure to set up a lattice with an empty root node. Secondly, it scans all HUIs (each set of HUIs is sorted by the quantity of items). For each HUI, the IsTraversed flags of root and child nodes are reset, then the algorithm calls the PushNodeToLattice method to add a HUI into the lattice. With regard to the PushNodeToLattice method, the variable flag is used to determine if node {X} can be added directly into the current node. If the current rootNode has child nodes where each childNode ⊆ X (line 23), the PushNodeToLattice method will be called recursively (line 25) to insert the {X} node into the lattice with each childNode as the root node. If there does not exist any childNode ∈ rootNode.Children ∧ Lc ⊆ X, X will be a child node of the current rootNode (line 29). To prepare information for mining CHUIs and generators, there are two flags, IsGenerator and IsClosed, which are attached to the HUI when it is inserted into the lattice (lines 25–29). From the results of the lattice, we can easily determine which nodes are CHUIs or generators. We can then propose an algorithm to extract all CHUIs and associated generators based on the HUIL, as described in the fifth section.
The HUIs extracted using the FHIM algorithm (Sahoo et al., 2015) from the given database in Tables 1 and 2 are shown in Table 3. The FHIM algorithm is then applied as follows. Let \( \text{level-}i \) be the set of HUIs in which each itemset has \( i \) items (\( i > 0 \)). The results in Table 3 will contain four levels, as follows: \( \text{level-1} = \{G, B, A, D, F, E\} \), \( \text{level-2} = \{BE, BF, AE, AC, DE, DF, EF\} \), \( \text{level-3} = \{BEF, BDE, AEF, ACE, DEF\} \) and \( \text{level-4} = \{BDEF, ACEF\} \). Firstly, an empty node will be initialized and added into the lattice as root node \( \text{rootNode} \). Secondly, the \( \text{level-1} \) set of HUI is processed, initially with \( G \), and it is connected directly to \( \text{rootNode} \), after that \( B, A, D, F \) and \( E \) are processed and also inserted directly into \( \text{rootNode} \) as child nodes. Thirdly, considering \( \text{level-2} \) of HUI, \( BE \) is processed; as the \( \text{rootNode} \) has a child node \( B \), it will call the \text{PushNodeToLattice} method recursively with \( B \) as the root node, and then add a connection between \( B \) and \( BE \). Similarly, with regard to child node \( E \), \( E \) and \( BE \) also have a connection. This process then continues with the rest of the HUIs. The IsClosed and IsGenerator flags are also updated if two nodes have a connection and the same support. The results of the lattice are shown in Figure 1.

**Extracting CHUIs and generators from HUIL**

Based on the results of Algorithm 1, each node in a lattice carries IsClosed and IsGenerator flags, indicating that the itemset is a CHUI and/or generator. In this section, we propose Algorithm 2, named LHUCI-Miner, to extract CHUIs and the list of generators of each CHUI.

**Algorithm for mining CHUIs and generators**

Algorithm 2. LHUCI-MINER (HUIL)
Input: HUIL with the \( \text{rootNode} \)
Output: CHUIs and their HUGs (\( \text{ChuiList} \))

\( \text{ExtractChuiAndGenerator()} \)

1. For each \( \text{Lc in rootNode.Children do} \)
   
   \( \text{ExtendFindingChuiAndGenerator(Lc)} \)

\( \text{ExtendFindingChuiAndGenerator(Lc)} \)

4. If \( \text{Lc.Flag = False then} \)
   
   \( \text{Set latticeNode.IsFlag = True} \)
   
   \( \text{If (Lc.IsClosed = True) then} \)
   
   \( \text{ChuiList.AddHighUtilityClosedItemset (Lc)} \)

5. \( \text{End} \)

6. Else if \( \text{(Lc.IsGenerator = True and Lc.IsClosed = False) then} \)

7. \( \text{FindingChuiAndGenerator (Lc)} \)

8. \( \text{End} \)

9. For each \( \text{Ls in Lc.Children do} \)

10. \( \text{ChuiList.AddGenerator (Lc, Lc)} \)

11. \( \text{End} \)

12. \( \text{End} \)

13. \( \text{End} \)

14. \( \text{End} \)

Table 3. Extracted HUIs from the database in Tables 1 and 2 with \( \text{min-util} = 20 \).  

| Itemset | Utility | Itemset | Utility | Itemset | Utility |
|---------|---------|---------|---------|---------|---------|
| A       | 21      | AE      | 28      | AFE     | 20      |
| B       | 20      | BE      | 21      | BDE     | 23      |
| D       | 22      | BF      | 23      | BFE     | 22      |
| G       | 22      | DE      | 37      | DFE     | 36      |
| E       | 22      | DF      | 24      | ACFE    | 25      |
| F       | 20      | FE      | 36      | BDFE    | 24      |
| AC      | 31      | ACE     | 38      |         |         |

Illustration

The HUIs extracted using the FHIM algorithm (Sahoo et al., 2015) from the given database in Tables 1 and 2 are shown in Table 3. The FHIM algorithm is then applied as follows. Let \( \text{level-}i \) be the set of HUIs in which each itemset has \( i \) items (\( i > 0 \)). The results in Table 3 will contain four levels, as follows: \( \text{level-1} = \{G, B, A, D, F, E\} \), \( \text{level-2} = \{BE, BF, AE, AC, DE, DF, EF\} \), \( \text{level-3} = \{BEF, BDE, AEF, ACE, DEF\} \) and \( \text{level-4} = \{BDEF, ACEF\} \). Firstly, an empty node will be initialized and added into the lattice as root node \( \text{rootNode} \). Secondly, the \( \text{level-1} \) set of HUI is processed, initially with \( G \), and it is connected directly to \( \text{rootNode} \), after that \( B, A, D, F \) and \( E \) are processed and also inserted directly into \( \text{rootNode} \) as child nodes. Thirdly, considering \( \text{level-2} \) of HUI, \( BE \) is processed; as the \( \text{rootNode} \) has a child node \( B \), it will call the \text{PushNodeToLattice} method recursively with \( B \) as the root node, and then add a connection between \( B \) and \( BE \). Similarly, with regard to child node \( E \), \( E \) and \( BE \) also have a connection. This process then continues with the rest of the HUIs. The IsClosed and IsGenerator flags are also updated if two nodes have a connection and the same support. The results of the lattice are shown in Figure 1.
Initially, the algorithm traverses all of child nodes from the root of the lattice. For each child node, it calls `ExtendFindingChuiAndGenerator` at line 2. `ExtendFindingChuiAndGenerator (Lc)` will add `Lc` to CHUIs list if `Lc.IsClosed` equals to True (lines 6–8). The `Lc` itemset can be both a CHUI and a generator if it is a HUCI and its IsGenerator flag is True (lines 11–13). If `Lc` is a generator and not a CHUI, `FindingChuiAndGenerator (Lc)` is called to find which CHUI that `Lc` belongs to (lines 9, 10). In this method, a queue structure and a list structure are used, with all child nodes of `Lc` as initial values (lines 18, 19). If a queue has items, it processes each itemset `Li` in the queue and adds `Lc` to be the generator of `Li` if `Li` is a CHUI and has the same support as `Lc` (lines 22–24). If `Li` has child nodes, then the algorithm continues to add all these into the queue (lines 25–30). Example CHUIs and generators for the sample database shown above are presented in Tables 4 and 5.

### Experimental results

In this section, we executed our algorithms, LHUCI-Miner and HUCI-Miner, to evaluate their performance with regard to time and memory usage when mining CHUIs and HUGs.

The experiments were implemented and tested on a system with the following configuration: Intel Core i7–6500U 2.5 GHz (4 CPUs), 16 GB of RAM and running Windows 10, 64 bit version. The source code was created in C# using Visual Studio 2015 Community, .Net framework 4.5. The data sets (Fournier-Viger, Gomariz, et al., 2014) for testing have the features shown in Table 6.

| Itemset | Utility | Support |
|---------|---------|---------|
| G       | 22      | 2       |
| F       | 20      | 2       |
| E       | 22      | 7       |
| BF      | 23      | 2       |
| DE      | 37      | 5       |
| EF      | 36      | 5       |
| ACE     | 38      | 2       |
| DEF     | 36      | 4       |
| BDEF    | 24      | 1       |
| ACEF    | 25      | 1       |
For each test data set in Table 6, we carried out the tests with various min-util (%) values to compare performance with regard to runtime and memory usage when mining CHUIs and generators for both HUCI-Miner and LHUCI-Miner. For any further references, we also reported some sample results of HUIs, CHUIs and HUGs for the test data sets (Table 7). For selecting min-util values mentioned in Table 7, we obtained references from the publication of Sahoo et al. (2015). Moreover, we also executed our proposed algorithm and High Utility Generic Basic algorithm with many other different min-util values to evaluate their performance; therefore, we adjusted to use some other values which could support and return more results on HUIs, CHUIs as well as HUGs. On the Chainstore data set, we used a very small value of min-util (0.005%) to have larger numbers of HUIs so that we could emphasize the effectiveness of the proposed LHUCI-Miner algorithm at runtime (Figure 5).

**Runtime for mining CHUIs and generators**

We executed our proposed algorithms, LHUCI-Miner and HUCI-Miner, on many different data sets (Table 6) with various min-util values. The results shown in Figures 2–6 indicate that using a lattice to extract CHUIs and their HUGs is more efficient than using the HUCI-Miner algorithm. The axes in Figures 2–6 are in the base-10 logarithmic scale.

We also obtained similar results for the time needed to mine CHUIs and generators from the Chess data set (Figure 3). This is a rapid process, and we can extract the information quickly within 8 seconds with min-util = 25% using HUCI-Miner. However, if we apply LHUCI-Miner with the same min-util = 25%, we can have results within 4 seconds. The average runtime ratio for mining CHUIs and generators on this data set is 51%.

The experimental results at runtime for extracting CHUIs and generators from the Retail data set (Figure 3) show that LHUCI-Miner is always faster than HUCI-Miner.

**Table 5.** Extracted HUGs.

| Itemset | Utility | Support |
|---------|---------|---------|
| G       | 22      | 2       |
| F       | 20      | 2       |
| E       | 22      | 7       |
| B       | 20      | 2       |
| D       | 22      | 5       |
| EF      | 36      | 5       |
| A       | 21      | 2       |
| DF      | 24      | 4       |
| BE      | 21      | 1       |
| ACE     | 25      | 1       |

**Table 6.** Characteristics of the test data sets.

| Data set  | # of transactions | # of items | Size (MB) |
|-----------|-------------------|------------|-----------|
| Chess     | 3196              | 75         | 0.63      |
| Mushroom  | 8124              | 119        | 1.03      |
| Retail    | 88,162            | 16,470     | 6.42      |
| Chainstore| 1,112,949         | 46,086     | 79.2      |
| Accidents | 340,183           | 468        | 63.1      |
Similarly, we also had the same runtime performance evaluation between LHUCI-Miner and HUCI-Miner algorithms on mining CHUIs and generators from the Mushrooms data set. On the Mushrooms data set, we also used various \textit{min-util} values decreasing from

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Data set & \textit{min-util} (%) & \# HUI & \# CHUI & \# HUG \\
\hline
Chess & 25 & 6406 & 3550 & 4074 \\
 & 26 & 2875 & 1727 & 1873 \\
 & 27 & 1246 & 816 & 874 \\
 & 28 & 493 & 339 & 358 \\
Mushroom & 10 & 9594 & 119 & 1623 \\
 & 11 & 5801 & 52 & 1080 \\
 & 12 & 2726 & 22 & 606 \\
 & 13 & 1152 & 6 & 317 \\
Retail & 0.01 & 22,479 & 21,935 & 21,959 \\
 & 0.02 & 7375 & 7312 & 7312 \\
 & 0.03 & 3765 & 3749 & 3749 \\
 & 0.04 & 2272 & 2266 & 2266 \\
Chainstore & 0.005 & 12,347 & 12,275 & 12,289 \\
 & 0.01 & 3884 & 3870 & 3873 \\
 & 0.02 & 1165 & 1165 & 1165 \\
 & 0.03 & 593 & 593 & 593 \\
Accidents & 10 & 7479 & 7479 & 7479 \\
 & 11 & 2367 & 2367 & 2367 \\
 & 12 & 728 & 728 & 728 \\
 & 13 & 189 & 189 & 189 \\
\hline
\end{tabular}
\caption{Results of HUIs, CHUIs and generators with the test data sets.}
\end{table}

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{Chess}
\caption{Runtime for mining CHUIs and generators from the Chess data set.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{Retail}
\caption{Runtime for mining CHUIs and generators from the Retail data set.}
\end{figure}
14% to 10% to observe the runtime of LHUCI-Miner versus that of HUCI-Miner on the same set of HUIs. In general, Figure 4 indicates that the LHUCI-Miner algorithm is more efficient than HUCI-Miner.

To compare LHUCI-Miner runtime and HUCI-Miner runtime with various numbers of HUIs, we evaluated the performance of LHUCI-Miner by adjusting the min-util. As can be seen in Figure 5, with the Chainstore data set, we decreased min-util from 0.03% to 0.004%. The result indicated that the runtime of LHUCI-Miner is always better than the runtime of HUCI-Miner.
Figure 6 shows that mining CHUIs and generators from the lattice using LHUCI-Miner is effective. In our experimental results for the Accidents data set with various min-util values from 11% to 15%, we can observe that LHUCI-Miner has a better runtime than HUCI-Miner with the input of huge HUIs.

Table 8. Memory usage for mining CHUIs and generators from the test data sets.

| Data set   | min-util (%) | HUCI-Miner memory usage (MB) | LHUCI-Miner memory usage (MB) |
|------------|--------------|------------------------------|-------------------------------|
| Chess      | 25           | 106.53                       | 106.81                        |
|            | 26           | 67.89                        | 65.82                         |
|            | 27           | 59.62                        | 60.40                         |
|            | 28           | 50.39                        | 49.43                         |
| Mushrooms  | 10           | 160.47                       | 172                           |
|            | 11           | 107.2                        | 113                           |
|            | 12           | 75.97                        | 76.06                         |
|            | 13           | 66.15                        | 66.29                         |
| Retail     | 0.01         | 418.25                       | 434.66                        |
|            | 0.02         | 310.15                       | 315.40                        |
|            | 0.03         | 224.54                       | 227.29                        |
|            | 0.04         | 191.46                       | 192.39                        |
| Chainstore | 0.005        | 1048.39                      | 1058.98                       |
|            | 0.01         | 896.39                       | 896.04                        |
|            | 0.02         | 762.38                       | 771.65                        |
|            | 0.03         | 671.68                       | 671.67                        |
| Accidents  | 10           | 374.56                       | 349.57                        |
|            | 11           | 322.13                       | 319.18                        |
|            | 12           | 311.38                       | 308.38                        |
|            | 13           | 305.73                       | 305.61                        |

Memory usage for mining CHUIs and generators

Mining CHUIs and generators using both HUCI-Miner and LHUCI-Miner algorithms is mainly based on HUIs, and the memory usage for the test data sets remains about the same in each case. Moreover, the use of a lattice makes it necessary to store the connections between parent and child nodes, and thus mining CHUIs and generators using a lattice approach can require more memory than when using HUCI-Miner.

In our experimental results, the difference in memory consumption between both algorithms is not too much (Table 8). In real applications, runtime is much more important than memory usage. Therefore, although the lattice approach for mining CHUIs and generators does not use less memory than HUCI-Miner, it is more efficient because it needs less execution time.

Conclusions and future works

In this work, we used the utility-confidence framework and the lattice concept to mine CHUIs and generators. We proposed an algorithm, called HUIL, to construct the lattice of a HUI set including Closed and Generator flags. Based on the constructed HUIL, we also propose the LHUCI-Miner algorithm to extract all CHUIs and generators. The process of this work requires less execution time. The results of the experiments in this study show that the proposed algorithm can be used effectively in various recommendation systems. In future work, we are going to investigating the use of the HUIL structure for quickly generating all high utility association rules.
Acknowledgements
This work was carried out during the tenure of an ERCIM ‘Alain Bensoussan’ Fellowship Programme.

Disclosure statement
No potential conflict of interest was reported by the authors.

Funding
This research is funded by NTTU Foundation for Science and Technology Development under grant number [2017.01.75].

Notes on contributors
Thang Mai graduated Master of Computer Science from University Of Information Technology, Vietnam, 2017. He has 10 years’ working experiences in software development. His research interests include data mining, database performance evaluation and reporting techniques.

Loan T.T. Nguyen is a postdoctoral fellow (ERCIM) at University of Warsaw, Warsaw, Poland. She received a Ph.D. degree at Wroclaw University of Science and Technology, Wroclaw, Poland in 2015, and M.Sc. degree in 2007 from the University of Information Technology, Vietnam National University of Ho Chi Minh, Vietnam. Her research interests include association rules, classification, and incremental mining.

References
Agrawal, R., Imielinski, T., & Swami, A. (1993). Mining association rules between sets of items in large databases. Proceedings of the 1993 ACM SIGMOD international conference on management of data, Washington, DC, USA, pp. 207–216.

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. VLDB’94, San Francisco, CA, USA, pp. 487–499.

Ahmed, C. F., Tanbeer, S. K., Jeong, B. S., & Lee, Y. K. (2009). Efficient tree structures for high utility pattern mining in incremental databases. IEEE Transactions on Knowledge and Data Engineering, 21(12), 1708–1721.

Davey, B. A., & Priestley, H. A. (1990). Introduction to lattices and order. Cambridge: Cambridge University Press.

Fournier-Viger, P., Gomariz, A., Soltani, A., & Gueniche, T. (2014). SPMF: Open-source data mining library. SPMF: A Java open-source pattern mining library. Journal of Machine Learning Research, 15(1), 3389–3393.

Fournier-Viger, P., Wu, C. W., & Tseng, V. S. (2014). Novel concise representations of high utility itemsets using generator patterns. Advanced data mining and applications – 10th international conference, Roskilde, Denmark, pp. 30–43.

Fournier-Viger, P., Wu, C. W., Zida, S., & Tseng, V. S. (2014). Faster high utility itemset mining using estimated utility co-occurrence pruning. Proceedings 21st international symposium on methodologies for intelligent systems, Porto, Portugal, pp. 83–92.

Gan, W., Lin, J. C., Fournier-Viger, P., & Chao, H. (2016). More efficient algorithms for mining high-utility itemsets with multiple minimum utility thresholds. DEXA, 1, 71–87.

Lan, G. C., Hong, T. P., & Tseng, V. S. (2014). An efficient projection-based indexing approach for mining high utility itemsets. Knowledge and Information Systems, 38(1), 85–107.

Li, Y., Yeh, J., & Chang, C. (2008). Isolated items discarding strategy for discovering high utility itemsets. Data & Knowledge Engineering, 64(1), 198–217.
Lin, C. W., Lan, G. C., & Hong, T. P. (2012). An incremental mining algorithm for high utility itemsets. *Expert Systems with Application, 39*(8), 7173–7180.

Liu, M., & Qu, J. (2012). *Mining high utility itemsets without candidate generation*. Proceedings of the 21st ACM international conference on information and knowledge management, Maui, Hawaii, USA, pp. 55–64.

Liu, Y., Liao, W., & Choudhary, A. (2005). *A two-phase algorithm for fast discovery of high utility itemsets*. Proceedings of the 9th Pacific-Asia conference on advances in knowledge discovery and data mining, Hanoi, Vietnam, pp. 689–695.

Sahoo, J., Das, A. K., & Goswami, A. (2015). An efficient approach for mining association rules from high utility itemsets. *Expert Systems with Applications, 42*(13), 5754–5778.

Tseng, V. S., Wu, C. W., Fournier-Viger, P., & Yu, P. (2015). Efficient algorithms for mining the concise and lossless representation of high utility itemsets. *IEEE Transactions on Knowledge and Data Engineering, 27*(3), 726–739.

Tseng, V. S., Wu, C. W., Fournier-Viger, P., & Yu, P. S. (2016). Efficient algorithms for mining top-K high utility itemsets. *IEEE Transactions on Knowledge and Data Engineering, 28*(1), 54–67.

Tseng, V. S., Wu, C. W., Shie, B., & Yu, P. S. (2010). UP-Growth: *An efficient algorithm for high utility itemset mining*. Proceedings of the 16th ACM SIGKDD international conference on knowledge discovery and data mining, Washington, DC, USA, pp. 253–262.

Tseng, V. S., Wu, C. W., Shie, B., & Yu, P. S. (2013). Efficient algorithms for mining high utility itemsets from transactional databases. *IEEE Transactions on Knowledge and Data Engineering, 25*(8), 1772–1786.

Vo, B., Hong, T., & Le, B. (2013). A lattice-based approach for mining most generalization association rules. *Knowledge-Based Systems, 45*, 20–30.

Vo, B., & Le, B. (2011). Mining minimal non-redundant association rules using frequent itemsets lattice. *International Journal of Intelligent Systems Technologies and Applications, 10*(1), 92–106.

Vo, B., Nguyen, H., & Le, B. (2009). *Mining high utility itemsets from vertical distributed databases*. International conference computing and communication technologies, RIVF’09, Da Nang, Vietnam, pp. 1–4.

Wang, J., Han, J., & Li, C. (2007). Frequent closed sequence mining without candidate maintenance. *IEEE Transactions on Knowledge and Data Engineering, 19*(8), 1042–1056.

Wu, C. W., Fournier-Viger, P., Gu, J., & Tseng, V. S. (2015). *Mining closed+ high utility itemsets without candidate generation*. Conference on technologies and applications of artificial intelligence, Tainan, Taiwan, pp. 187–194.

Zaki, M. J., & Hsiao, C. J. (2005). Efficient algorithms for mining closed itemsets and their lattice structure. *IEEE Transactions on Knowledge and Data Engineering, 17*(4), 462–478.

Zida, S., Fournier-Viger, P., Lin, J. W., Wu, C. W., & Tseng, V. S. (2016). EFIM: A fast and memory efficient algorithm for high-utility itemset mining. *Knowledge and Information Systems, 51*, 595–625.