Efficient Method for Mining High-Utility Itemsets Using High-Average Utility Measure

Loan T. T. Nguyen[1,2], Trinh D. D. Nguyen[3], Anh Nguyen[4], Phuoc-Nghia Tran[5], Cuong Trinh[6], Bao Huynh[7], and Bay Vo[7](✉)

[1] School of Computer Science and Engineering, International University, Ho Chi Minh City, Vietnam
nttloan@hcmiu.edu.vn

[2] Vietnam National University, Ho Chi Minh City, Vietnam

[3] Department of Information Technology, HCM Pre-University School, Ho Chi Minh City, Vietnam
dzutrinh@hcmpreu.edu.vn

[4] Faculty of Computer Science and Management, Wroclaw University of Science and Technology, Wroclaw, Poland
anh.nguyen@pwr.edu.pl

[5] Bac Lieu University, Bac Lieu, Vietnam
tpn_blu@yahoo.com.vn

[6] Artificial Intelligence Laboratory, Faculty of Information Technology, Ton Duc Thang University, Ho Chi Minh City, Viet Nam
trinhphicuong@tdtu.edu.vn

[7] Faculty of Information Technology, Ho Chi Minh City University of Technology (HUTECH), Ho Chi Minh City, Vietnam
{hq.bao,vd.bay}@hutech.edu.vn

Abstract. Mining high-utility itemsets (HUIs) based on high-average utility measure is an important task in the data mining field. However, many of the existing algorithms are performing the mining process sequentially and do not utilize the widely available multi-core processors, thus requiring long execution times. To address this issue, we propose an extended version of the HAUI-Miner algorithm, namely pHAUI-Miner. The algorithm applies multi-thread parallel processing to significantly reduce the mining time. Experimental evaluations on standard databases have shown the effectiveness of the proposed algorithm over the original and sequential method.

Keywords: High-average utility itemset · Data mining · Parallel computing · Multi-thread

1 Introduction

To discover the associations and the relations among the items within a transactional database, frequent itemset mining (FIM) [1] methods were applied. Companies have incorporated FIM onto their available databases to boost the executive performance. The analysis of the transactions helps put forth effective strategies in their business, such as catalog designing, marketing, customer behavior's analysis or basket analysis. FIM analyzes the customer's shopping habit, and then discover the associations among the items that were selected by the customer. Retailers use the discovered knowledge to develop effective strategies to boost their sales.

Some algorithms to perform this task are Apriori [1], AprioriTid [1], Eclat [2], FP-Growth [3], etc. Of them, the FP-Growth only requires two database scans to construct the FP-tree and directly extracts frequent itemsets from the tree. Thus, discovering the complete set of frequent itemsets (FIs). FIM only considers the existence of the items within transactions and treats all items equally. It completely ignores other important information such as the purchase quantity of items, item's profit, etc. In real-word applications, every item has its own value (unit profit), and in most cases, frequent patterns might not be the ones that yield high profit, or the usefulness to the users. Briefly, the generated profit of an item when purchased, called utility, is the product of the purchase quantity and its unit profit. Patterns or itemsets that generate high profit and satisfy a user-specified threshold are called high-utility patterns (HUPs) or itemsets (HUIs).

In HUIM, utility of an item or itemset is the sum of its utility in the database. This traditional utility calculation has a major drawback: it ignores the length of the itemset and thus the longest itemsets has higher utility value. Thus, it is not fair when applying this utility calculation on to all itemsets. To address this drawback, a new utility measure, called average utility (*au*) measure [4], was proposed to better assess the utility of itemsets. It is defined as the sum of the utilities of the itemset in transactions that contain it, divided by its length or the number of items in that itemset. If an itemset has its *au* value no less than a user-specified threshold, called the minimum average utility threshold ($minAU$), then it's called high-average utility itemset (HAUI). However, the downward closure property does not hold for this new utility measure. An itemset whose *au* value does not satisfy the threshold may be combined with one or more items to form a HAUI. This might generate a large number of candidates and the process of checking all the generated candidates is time consuming. A new upper-bound was proposed to address this issue and called average utility upper-bound (*auub*). Lan et al. has applied this new upper-bound to prune candidates [5, 6]. Lan et al. also proposed new index-table structure to speed up the mining process [7].

The rest of the paper is organized as follows: Sect. 2 surveys related works on HUIs and average utility itemsets mining. Section 3 describes definitions and propose methods on average utility itemsets mining. Experimental results will be showed in Sect. 4. Finally, Sect. 5 presents conclusion and future improvements.

2 Related Work

Extending from FIM, the task of mining the complete set of HUIs is called high-utility itemset mining (HUIM). The problem of mining HUIs was first proposed in Yao et al. 2004 [8]. The authors presented the concepts of utility and high-utility measure. HUIM is considered a challenging task since the utility measure does not satisfies the downward closure property [1], which original states in FIM that all subsets of a frequent itemset must also be frequent. In 2005, Liu et al. presented an algorithm to mine HUIs in two phases [9], namely Two-Phase. In the first phase, the algorithm computes the downward closure on utility of itemsets in transactions to prune the search space. This novel upper-bounds on utility is called Transaction Weighted Utility (TWU). Second phase scans the database again to calculate the exact utility value of itemsets to determine which one is HUI. However, TWU is not tight enough to effectively prune the search space, leaving a huge number of candidates for the algorithms to check. Thus, to reduce further the generated candidates and database scans, Lin et al. proposed a tree structure, called HUP-tree [10], to effectively mine HUIs. It first calculates the utility values for 1-itemsets and uses them to construct the HUP-tree. Then the algorithm recursively traverses the tree to extract HUIs based on a header table. The algorithm requires only two database scans to discover the complete set of HUIs in a database. In 2016, Zida et al. proposed a single phase for effectively mining HUIs, namely EFIM [11]. The algorithm using new and tighter upper-bounds to prune a large number of candidates, thus significantly reduce the mining time. To achieve better performance and to utilize the full power of the modern processors, Nguyen et al. has proposed a parallel version of EFIM, named pEFIM [12]. The algorithm partitions the search space in to separated sub-spaces and assigned each execution thread to a sub-space, thus dramatically reduce time needed to mine HUIs.

Lan et al. proposed an algorithm which incorporated index-table and the average utility upper-bound (*aub*) to mine HAUIs in 2012 [7]. The algorithm presented an effective pruning strategy using indices to reduce the execution time and memory consumption. Lin et al. proposed a single phase algorithm to efficiently mine HAUIs, named HAUI-Miner [13]. Also in 2016, Lu et al. proposed an algorithm and a tree structured, named HAUI-tree to quickly generate candidates and mine HAUIs [14]. The algorithm consists of two steps: (i) calculates the utility values of 1-itemsets in transactions to identify the maximum utility value for each transaction, then calculates the *aub* for each item. (ii) From the list of all 1-itemsets that satisfied the threshold, construct the list of 2-itemsets using the downward closure property based on the HAUI-tree. The process repeats until no new candidates generated. To mine HAUIs using multiple $minAU$ thresholds, Lin et al. proposed the algorithm HAUIM-MMAU with two pruning strategies, known as the improved EUCP strategy (IEUCP) and Prune Before Calculation strategy (PBCS) [15]. However, the HAUIM-MMAU uses the generate-and-test approach, which is time consuming. In 2018, Lin et al. proposed another algorithm named MEMU [16] and three pruning strategies to increase the performance of the mining process. Experiments show MEMU has better performance compare to HAUIM-MMAU in terms of runtime, memory usage, candidates and scalability.

3 Proposed Algorithm

This section presents preliminary concepts and problem statement. Many of the definitions and theorems were given in detail and proved in [13].

3.1 Preliminaries

Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ is the finite set of m distinct items. A transaction database $\mathcal{D}$ is a set of transactions $\mathcal{D} = \{T_1, T_2, \ldots, T_n\}$. In which, each transaction $T_q \in \mathcal{D}$ and $T_q \subseteq \mathcal{I}$ $(1 \le q \le n)$. Each transaction T_q has a unique identifier q, called its *TID*.

Definition 1. The utility of an item i_j in a transaction T_q is the product of its purchase quantity and its unit profit, denoted as $u(i_j, T_q)$ [13].

$$u(i_j, T_q) = q(i_j, T_q) \times p(i_j) \tag{1}$$

Whereas, $q(i_j, T_q)$ is the purchase quantity of item i_j in transaction T_q; positive integer $p(i_j)$ is the unit profit of item i_j, which is given in the unit profit table $\mathcal{PT}$.

Given set of k distinct items $X = \{i_1, i_2, \ldots, i_k\}$, $X \subseteq \mathcal{I}$ and is called k-itemset where k is the length of X, $k = |X|$.

Definition 2. The average utility of a k-itemset X in a transaction T_q, denoted as $au(X, T_q)$, is defined as follows [13].

$$au(X, T_q) = \frac{\sum_{i_j \in X \wedge X \subseteq T_q} q(i_j, T_q) \times p(i_j)}{k} \tag{2}$$

When $k = 1$, X becomes a single item $i_j \in \mathcal{I}$, and thus the average utility of this 1-itemset in transaction T_q can be calculated as follows [13].

$$au(i_j, T_q) = \frac{q(i_j, T_q) \times p(i_j)}{1} = u(i_j, T_q) \tag{3}$$

Definition 3. Average utility of itemset X in database $\mathcal{D}$, denoted as $au(X)$, is defined as follows [13].

$$au(X) = \sum_{X \subseteq T_q \wedge T_q \in \mathcal{D}} au(X, T_q) \tag{4}$$

Definition 4. Utility of transaction T_q, denoted as $tu(T_q)$, is defined as follows [13].

$$tu(T_q) = \sum_{i_j \in T_q} u(i_j, T_q) \tag{5}$$

Definition 5. Total utility of database $\mathcal{D}$, denoted as TU, and is defined as follows [13].

$$TU = \sum_{T_q \in \mathcal{D}} tu(T_q) \tag{6}$$

Definition 6. The transaction-maximum utility of transaction T_q, denoted as $tmu(T_q)$, is defined as follows. [13].

$$tmu(T_q) = \max\{u(i_j, T_q) | i_j \in T_q\} \tag{7}$$

Definition 7. Average-utility upper-bound of an itemset X, denoted as $auub(X)$, is the sum of all the transaction-maximum utilities of transactions containing X [13].

$$auub(X) = \sum\nolimits_{X \subseteq T_q \wedge T_q \in \mathcal{D}} tmu\left(T_q\right) \tag{8}$$

Definition 8. An itemset X is called a high average-utility upper-bound itemset, denoted as $HAUUBI(X)$, if its average-utility upper-bound is no less than $minAU$. $HAUUBI(X)$ is defined as follows [13].

$$HAUUBI(X) = \{X \mid auub(X) \geq TU \times minAU\} \tag{9}$$

Theorem 1. The *auub* measure is downward closed. The transaction-maximum-utility downward closure (*TMUDC*) property holds for any *HAUUBIs* [13].

The proof of this theorem is given in detail in [13]. From Theorem 1, we have two corollaries as follows:

Corollary 1. Given a k-itemset X^k, if X^k is a $HAUUBI$, then all subsets of X^k are also $HAUUBIs$.

Corollary 2. If an itemset X^k is not a HAUUBI, then all supersets of X^k are not $HAUUBIs$.

Theorem 2. The *TMUDC* property ensures that $HAUUBIs \subseteq HAUIs$. Thus, if an itemset is not a $HAUUBI$, then none of its supersets are HAUIs. If an itemset is not a $HAUUBI$, it is also not a HAUI [13].

By using Theorem 2, we can prune a large number of unpromising candidates from the search space and thus, reduce the mining time.

3.2 Problem Statement

Given a user-specified minimum average-utility threshold ($minAU$), $minAU$ is a positive integer. The problem of mining high-average utility itemsets in database $\mathcal{D}$ is the task of discovering the complete set of HAUIs. An itemset X is a HAUI if and only if its utility is no less than $minAU$. The problem can be defined as follows.

$$HAUIs = \{X \mid au(X) \geq TU \times minAU\} \tag{10}$$

3.3 The Revised Database

The original algorithm HAUI-Miner [13] requires two database scans. The first scan discovers the set of high average-utility upper-bound 1-itemsets (1-*HAUUBIs*). The second scan constructs the average-utility list (AU-list) of 1-itemsets. In this second scan, all 1-itemsets that are non-HAUUBIs will be removed from the database. The database obtained after removing all these non-HAUUBIs is called revised database $\mathcal{D}'$. The pseudo-code of the construction of $\mathcal{D}'$ (**InitRevisedDatabase**) is given in Algorithm 1.

3.4 The Average-Utility List (AU-List) Structure

The AU-list of an item or an itemset X is a list of elements, such that each element represents a transaction $T_q \in \mathcal{D}'$ and $X \subseteq T_q$. Each element consists of three fields, as follows:

- The *tid* field indicates the transaction T_q.
- The *iu* field indicates the utility of X in T_q, $u(X, T_q)$.
- The *tmu* field indicates the transaction-maximum-utility of X in T_q, $tmu(X, T_q)$.

To construct the AU-list for k-itemset with $k \geq 2$, it is not necessary to rescan the database. They can be constructed by intersecting the AU-list of smaller itemsets. By using the AU-list, the search space of the whole algorithm can be modelled as a set-enumeration tree. In which, each node represents an itemset. The HAUI-Miner explores the tree using depth-first search and prune the unpromising child nodes early using Theorem 3. This can be done by using the sum of *iu* and *tmu* field in the designed AU-list.

Algorithm 1. The construction of $\mathcal{D}'$ and obtain 1-$HAUUBIs$

```
 Input:  transaction database $\mathcal{D}$, $minAU$ threshold
 Output: $\mathcal{D}'$ and set of all 1-$HAUUBIs$, total utility $TU$.
1: for each transaction $T_q \in \mathcal{D}$ do
2:    scan $T_q$ to calculate $tmu(T_q)$ and $TU$.
3: for each item $i \in \mathcal{D}$ do
4:    calculate $auub(i)$
5: 1-$HAUUBIs \leftarrow \{ i \mid auub(i) \geq TU \times minAU\}$
6: scan $\mathcal{D}$ to remove each item $i$ such that $i \notin$ 1-$HAUUBIs$ and
   obtain $\mathcal{D}'$.
7: for each $T_q \in \mathcal{D}'$ do
8:    sort $T_q$ in ascending order $\prec$ of $auub(i)$.
9: return $\mathcal{D}'$, 1-$HAUUBIs$, $TU$.
```

Algorithm 2. AU-list construction

```
 Input: AU-list of itemset $P, P_x, P_y$: $P.AUL, P_x.AUL, P_y.AUL$
 Output: AU-list of itemset $P_{xy}$: $P_{xy}.AUL$
1: $P_{xy}.AUL \leftarrow null$
2: for each $E_x \in P_x.AUL$ do
3:   if $\exists E_y \in P_y.AUL \wedge E_x.tid = E_y.tid$ then
4:      if $P.AUL = null$ then
5:         $E_{xy} \leftarrow \langle E_x.tid, E_x.u + E_y.u, E_y.tmu\rangle$
6:      else
7:         if exists $E \in P.AUL$ such that $E.tid = E_x.tid$ then
8:            $E_{xy} \leftarrow \langle E_x.tid, E_x.u + E_y.u, E_y.tmu\rangle$
9:      $P_{xy}.AUL \cup E_{xy}$
10: return $P_{xy}$
```

Algorithm 3. DFS-based search space exploration algorithm

```
   Input:  AU-list of itemset P: $P.AUL$, list of AU-list of all
           $P'$s 1-extension: $AULs$, $minAU$ threshold, $TU$ of $\mathcal{D}$
   Output: all HAUI with prefix $P$
1: for each $Y.AUL \in AULs$ do
2:   if $\frac{SUM.Y.iu}{|Y.AUL|} \geq minAU \times TU$ then
3:     $HAUIs \leftarrow HAUIs \cup Y$
4:   if $SUM.Y.tmu \geq minAU \times TU$ then
5:     $extAULs \leftarrow null$
6:     for each $Z.AUL$ after $Y.AUL, Z.AUL \in AULs$ do
7:       $extAULs \leftarrow extAULs \cup \boldsymbol{Construct}(P.AUL, Y.AUL, Z.AUL)$
8:       $\boldsymbol{Search}(Y.AUL, extAULs, minAU, TU)$
```

Theorem 3. Given an itemset X, if the sum of it's *tmu* in all transactions containing X, using the AU-list, is less than $minAU$, all extensions of X are not HAUIs [13].

The pseudo-code of the AU-list construction (**Construct**) is given in Algorithm 2, the pseudo-code of the DFS based search process (**Search**) is given in Algorithm 3.

Algorithm 4. The pHAUI-Miner algorithm

```
   Input: transaction database $\mathcal{D}$, $minAU$ threshold.
   Output: $\mathcal{D}'$ and set of all $1\text{-}HAUUBIs$.
1: $\mathcal{D}', 1\text{-}HAUUBIs, TU \leftarrow \boldsymbol{Init}(\mathcal{D}, minAU)$
2: for each $T_q \in \mathcal{D}'$ do
3:   sort $T_q$ in ascending order $\prec$ of $auub(i)$.
4: for each item $i \in 1\text{-}HAUUBIs$ do parallel
5:   scan $\mathcal{D}'$ to construct $AULs$ containing AU-list of all
     1-extensions of $i$
6:   $\boldsymbol{Search}(i.AUL, AULs, minAU, TU)$
```

3.5 The PHAUI-Miner Algorithm

The original algorithm HAUI-Miner, obtained from the SPMF open-source package [17], perform several database scans when processing each item in the set of all 1-*HAUUBIs*, which is not efficient. To relieve the algorithm from this bottleneck, parallel processing should be considered. Modern processors are now containing multiple cores to handle many tasks simultaneously. To speed-up the process of mining HAUIs, increase the response time and to utilize widely available multi-core processors, we apply multi-thread parallel processing into this phase. The load balance strategy used in our proposed algorithm is Task Parallelism. In detail, we partition the search space into separated sub-search spaces and assign to them a DFS-based search thread using divide and conquer strategy. Each thread in turn will recursively explore down its sub-search space in parallel. Consider the search space of the algorithm as shown in Fig. 1, the search space is partitioned at the level of all items contained in the 1-*HAUUBIs*. Thus, significantly reduce the time needed to discover the complete set of HAUIs.

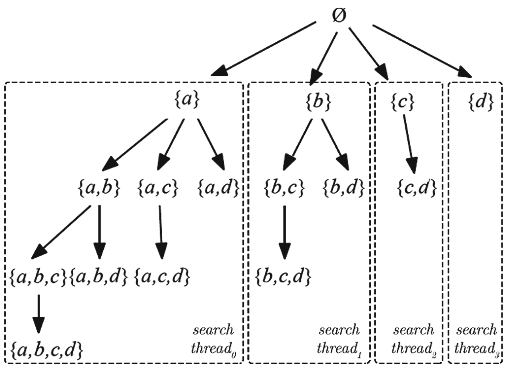

Fig. 1. Partition search space of the pHAUI-Miner algorithm

The pseudo-code of the Algorithms 1 to 3 remains unchanged as in the sequential version. We incorporated them in to a parallel version and name it pHAUI-Miner, whose pseudo-code is shown in Algorithm 4. In which, the parallel processing of each item in the 1-*HAUUBIs* is given from line #4 to #6. Each call of the **Search** function in line #6 operates in its own sub-search space with respect to an item *i*.

4 Experimental Studies

This section presents the experiments of the proposed algorithm on the standard databases to evaluate its performance and effectiveness. All the experiments were conducted on a workstation equipped with an Intel® Xeon® E5-2678 v3 processor (12-core/24-thread) clocked at 2.5 Ghz, 32 GB DDR4 ECC of internal memory and running Windows 10 Pro Workstation. All the algorithms used in the experiments were developed using the Java programming language (JDK8). The HAUI-Miner source code can be obtained from the SPMF package [17].

The databases used in the experiments are standard databases which are used in many data mining researches [17], and can be downloaded at https://bit.ly/2vixvH0. Their characteristics are given in Table 1.

Table 1. Database characteristics

Database	#Transactions	#Items
Mushroom	8,124	119
Retail	88,163	16,470
Kosarak	990,003	41,270
Chainstore	1,112,949	46,086

We compare the runtime of the proposed algorithm, pHAUI-Miner, using 4 threads and 24 threads against the original and sequential algorithm HAUI-Miner on all the test databases. Runtime comparison on the *Mushroom*, *Retail*, *Kosarak* and *Chainstore* database are given from Fig. 2a *to* Fig. 2d, respectively. Furthermore, the average speed-up factors of the pHAUI-Miner algorithm over the original HAUI-Miner algorithm on all the databases are also provided in Fig. 3.

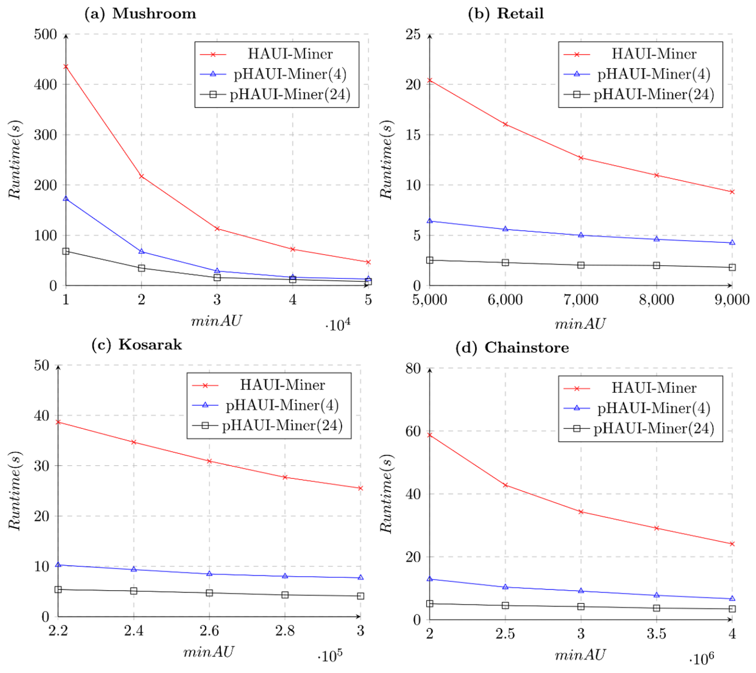

Fig. 2. Runtime comparisons on four databases for various *minAU* thresholds

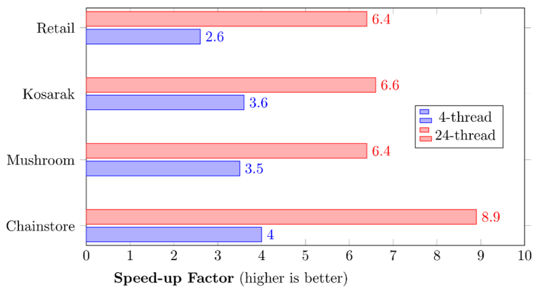

Fig. 3. Average speed-up factor on all test databases

It can be seen in Fig. 2 that the parallel versions pHAUI-Miner dominate all the tests. With the speed-up factor using 4 threads is up to 4 times faster, and with 24 threads, the factor is over 6 to 8 times faster. It can be observed from Fig. 3 that by using 4 threads on the test system, the speed-up factor is the most ideal since the speed-up factor is up to 4 times, while using 24 threads, the speed-up factor is only over 8 times. Considering

4 threads, the highest factor is observed on the *Chainstore* database, the largest one, and the lowest factor is on the *Retail* database. There are several factors that could affect the speed-up when using more processing cores. First, it can be seen in line #5 of Algorithm 4 each thread performs a full scan of the revised database $\mathcal{D}'$, thus using more processing threads would cause excessive scans on $\mathcal{D}'$ and reduce the effectiveness of the algorithm. Next, to discover the complete set of HAUIs while using parallelism, synchronization between threads is required to maintain the processing order of items in the 1-*HAUUBIs*, thus increased the overhead of the algorithm.

5 Conclusions

In this work, we identified and analyzed the issues of the original algorithm HAUI-Miner when mining high-average utility itemsets. Based on this, we proposed an extended version of the original algorithm, named pHAUI-Miner, which address all these issues to improve the effectiveness and performance of the mining process. The proposed algorithm handles effectively all the test databases and has much better performance in terms of runtime than the original one. In the future, we want to improve further the algorithm to further reduce cost of database scans; apply new utility calculations to make it work on dynamic profits databases; applied distributed computing framework such as Apache Spark to allow mining from large-scale databases.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proceedings of the 20th International Conference on Very Large Data Bases, pp. 487–499 (1994)
2. Zaki, M.: Scalable algorithms for association mining. IEEE Trans. Knowl. Data Eng. **12**(3), 372–390 (2000)
3. Han, J., Pei, J., Yin, Y., Mao, R.: Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Data Min. Knowl. Disc. **8**(1), 53–87 (2004). https://doi.org/10.1023/B:DAMI.0000005258.31418.83
4. Hong, T.-P., Lee, C.-H., Wang, S.-L.: Effective utility mining with the measure of average utility. Expert Syst. Appl. **38**(7), 8259–8265 (2011)
5. Lan, G., Hong, T., Tseng, V.S.: Mining high transaction-weighted utility Itemsets. In: 2010 Second International Conference on Computer Engineering and Applications, vol. 1, pp. 314–318 (2010)
6. Lan, G.-C., Hong, T.-P., Tseng, V.: Efficiently mining high average-utility itemsets with an improved upper-bound strategy. Int. J. Inf. Technol. Decis. Making **11**, 1009–1030 (2012)
7. Lan, G.-C., Hong, T.-P., Tseng, V.: A projection-based approach for discovering high average-utility itemsets. J. Inf. Sci. Eng. **28**, 193–209 (2012)
8. Yao, H., Hamilton, H., Butz, C.: A foundational approach to mining itemset utilities from databases. In: Proceedings of the Fourth SIAM International Conference on Data Mining, vol. 4, pp. 22–24 (2004)
9. Liu, Y., Liao, W., Choudhary, A.: 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, pp. 689–695 (2005)

10. Lin, C.-W., Hong, T.-P., Lu, W.-H.: An effective tree structure for mining high utility itemsets. Expert Syst. Appl. **38**(6), 7419–7424 (2011)
11. Zida, S., Fournier-Viger, P., Lin, J.C.-W., Wu, C.-W., Tseng, V.S.: EFIM: a fast and memory efficient algorithm for high-utility itemset mining. Knowl. Inf. Syst. **51**(2), 595–625 (2016). https://doi.org/10.1007/s10115-016-0986-0
12. Nguyen, T.D.D., Nguyen, L.T.T., Vo, B.: A parallel algorithm for mining high utility itemsets. In: Świątek, J., Borzemski, L., Wilimowska, Z. (eds.) ISAT 2018. AISC, vol. 853, pp. 286–295. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-99996-8_26
13. Lin, J.C.-W., Li, T., Fournier-Viger, P., Hong, T.-P., Zhan, J., Voznak, M.: An efficient algorithm to mine high average-utility itemsets. Adv. Eng. Inform. **30**(2), 233–243 (2016)
14. Lu, T., Vo, B., Nguyen, H., Hong, T.-P.: A new method for mining high average utility itemsets. In: Computer Information Systems and Industrial Management, pp. 33–42 (2014)
15. Lin, J.C.-W., Li, T., Fournier-Viger, P., Hong, T.-P., Su, J.-H.: Efficient mining of high average-utility itemsets with multiple minimum thresholds. In: Advances in Data Mining. Applications and Theoretical Aspects, pp. 14–28 (2016)
16. Lin, J.C.-W., Ren, S., Fournier-Viger, P.: MEMU: more efficient algorithm to mine high average-utility patterns with multiple minimum average-utility thresholds. IEEE Access **6**, 7593–7609 (2018)
17. Fournier-Viger, P., Gomariz, A., Gueniche, T., Soltani, A., Wu, C.-W., Tseng, V.S.: SPMF: a Java open-source pattern mining library. J. Mach. Learn. Res. **15**(1), 3389–3393 (2014)