A Novel Approach for Mining High-Average Utility Itemsets on Incremental Database

Ye-In Chang
Department of Computer Science and Engineering
National Sun Yat-sen University
Kaohsiung, Taiwan
changyi@mail.cse.nsysu.edu.tw

Chen-Chang Wu
Department of Biotechnology and Green Industry
Fooyin University
Kaohsiung, Taiwan
pt335@fy.edu.tw

Hsiang-En Kuo
Department of Computer Science and Engineering
National Sun Yat-sen University
Kaohsiung, Taiwan
anderson40205@gmail.com

***Abstract*—High-Utility Itemset (HUI) mining is a data mining technique that identifies meaningful itemsets by integrating factors such as profits, weights, and item quantities. However, traditional HUI mining algorithms often overestimate utility, particularly for longer itemsets, leading to biased results. To address this limitation, High Average-Utility Itemset (HAUI) mining has been proposed, which normalizes utility by considering the length of itemsets, thereby reducing the excessive favoritism toward long itemsets. Despite this advancement, a critical challenge remains that the inherent variability in the value and importance of individual items makes it inappropriate to apply a uniform threshold for identifying HAUI. To tackle this issue, recent research has focused on High Average-Utility Itemset Mining with Multiple Minimum Utility Thresholds, aiming to develop more flexible and accurate methods for utility evaluation. This emerging approach seeks to enhance the relevance and precision of itemset mining by accommodating the diverse significance of items within a dataset. Among the existing approaches, Sethi et al. proposed the GHAIM algorithm, which utilizes a list structure, enforces strict upper bounds, and demonstrates strong performance. However, the GHAIM algorithm requires scanning the database twice and is limited to static databases. To address these limitations, this paper introduces a novel approach designed to scan the database only once, constructing a tree structure to store all transactions efficiently, and enable rapid retrieval of specific itemsets from the tree. Compared to the list structure used in GHAIM, our tree-based approach not only reduces storage space but also eliminates the need for time-consuming item join operations. Furthermore, the proposed approach incorporates several pruning strategies. To handle incremental database environments, our approach employs additional data structures to store HAUI and candidate itemsets, avoiding the need for re-scanning the database when new data is inserted. Performance evaluations demonstrate that the proposed algorithm outperforms the GHAIM algorithm in terms of efficiency.**

Keywords—Data Mining, High Average Utility Itemset Mining, Incremental Mining, Multiple Minimum Utility Thresholds

I. INTRODUCTION

High Utility Pattern Mining (HUPM) is widely applied in areas such as stock market analysis, commodity market evaluation, and medical data processing. Unlike traditional *Frequent Itemset Mining* (FIM), which considers only frequency, *HUPM* incorporates both quantity and profit. However, it does not account for pattern length, which can lead to misleading results.

For example, in a store where most customers buy pencils, erasers, and rulers, a wealthy customer making a large purchase that includes apples and milk could distort the results. Since this transaction has high utility, *HUPM* may incorrectly classify the entire itemset as meaningful. However, the key pattern should consist only of pencils, erasers, and rulers, as they are commonly bought together. *High Average Utility Pattern Mining* (HAUPM) addresses this issue by refining meaningful pattern detection.

Despite its benefits, *HAUPM* has a major limitation: it applies a single minimum high-utility threshold to all items, which is problematic due to the diversity in product attributes. In retail, items vary in price (e.g., diamonds vs. ceramics), purchase frequency (e.g., milk vs. refrigerators), and profit margin (e.g., gold vs. clothes). Using a single threshold may lead to biased evaluations—if set too high, important patterns may be missed; if too low, too many irrelevant patterns may be found. Integrating *HAUPM* with multiple *Minimum Average Utility Threshold Values* (MATV) can improve decision-making in pricing, promotions, and product placement.

Several algorithms [1, 2, 3] have been developed for mining high-average utility patterns, but they rely on user-defined thresholds, which can affect performance and accuracy. Some approaches [4, 5, 6, 7] introduce multiple utility thresholds to handle item diversity. However, existing algorithms often sort thresholds and *Average-Utility Upper Bound* (AUUB) values inconsistently, leading to inefficiencies. The *GHAIM* algorithm [7] addresses this by introducing *Suffix Minimum Average Utility* (SMAU) and tighter upper bounds but is limited to static databases and requires multiple scans.

To overcome these issues, we proposes a novel algorithm that scans the database once and constructs a *TUR*-Tree structure for efficient storage and retrieval of transactions. By leveraging bit strings, the algorithm quickly identifies itemsets without extra item-joining operations. Additionally, we introduce a tighter upper bound than the *eubr* value used in *GHAIM* [7], enhancing pruning efficiency.

For incremental databases, re-scanning data is costly. Our method maintains *HAUI* and Non-*HAUI* candidate lists to track promising patterns dynamically. Experimental results show that our proposed approach is more efficient than *GHAIM*, making it a superior solution for high-average utility pattern mining.

The remainder of this paper is structured as follows: Section 2 provides a review of high-average utility pattern mining algorithms and multiple minimum threshold approaches applied to *HAUPM*. Section 3 introduces the proposed algorithm in detail. Section 4 presents performance evaluations and comparisons with the *GHAIM* algorithm [7]. Finally, Section 5 concludes the paper.

II. A Survey of Algorithms for Mining High Average Utility Itemsets with Multiple Minimum Utility Thresholds

Numerous algorithms have been developed for mining *High Average Utility Itemsets* (HAUIs). However, most existing methods in this field do not take multiple minimum thresholds into account. This section provides an overview of key algorithms designed for discovering high-average utility itemsets, as well as those incorporating multiple minimum utility thresholds for high-utility itemset mining. The algorithms are presented in chronological order based on their publication dates.

A. The TUB-HAUPM Algorithm

The *TUB-HAUPM* algorithm [2], proposed by Wu *et al*., aims to reduce the search space in high-average utility pattern mining by employing upper-bound constraints. Traditionally, three common upper bounds are used: the average utility upper bound, a looser upper bound, and a revised tighter upper bound. However, this algorithm introduces two even tighter upper bounds *Maximum Following Utility Upper-Bound* (mfuub) and *Top-k Transaction-Maximum Utility Upper-Bound* (krtmuub) to efficiently prune unpromising itemsets at an early stage.

In the first database scan, the algorithm computes the *auub* for each item. Items with *auub* values below the threshold are eliminated. The second database scan then sorts transactions in ascending order based on *auub* values. During the exploration stage, the *Transaction-Rival Tight Upper-Bound* (trtub) is determined as the minimum of *mfuub* and *krtmuub*. If *trtub* falls below the threshold, its supersets are not further explored. This process continues until all *HAUI*s are identified..

B. The MEMU Algorithm

The *MEMU* algorithm [6], proposed by Lin *et al*., addresses the limitations of the HUIM-MMAU algorithm [5], which requires multiple database scans and generates an excessive number of candidate itemsets. To enhance efficiency in discovering *HAUI*s, *MEMU* utilizes the *Compact Average-Utility List* (CAU-list) and the *Estimated Average-Utility Matrix* (EAUM) structure.

The algorithm consists of three main stages. In the first stage, the database is scanned once to compute *auub* values. The *Multiple Minimum High Average-Utility* Table (MAUTable) is then checked, and the smallest threshold is identified as the *Least Minimum High Average-Utility Count* (LMAU). Items with *auub* values below *LMAU* are removed from the database. A second scan updates *auub* values and sorts the database in ascending order based on thresholds. The second stage begins the mining process, where *EAUM* is applied as a pruning strategy. If the *EAUM* value of an itemset exceeds *LMAU*, the algorithm proceeds to generate three-itemsets and constructs the *CAU*-list. In the final stage, the *CAU*-list is further refined, and the *LA* pruning strategy is applied to eliminate unpromising candidates early. This process continues until all *HAUI*s are efficiently identified.

C. The GHAIM Algorithm

The *GHAIM* algorithm [7], proposed by Sethi *et al*., addresses the limitations of previous approaches in high-average utility pattern mining. The *MEMU* algorithm [6] improved upon *HUIM-MMAU* [5] by reducing database scans and candidate generation. Meanwhile, the *MHUI* algorithm [4] introduced generalized pruning techniques for the multiple minimum threshold method without sorting user-defined thresholds. However, *MHUI* focused on *High Utility Patterns* (HUPs) rather than *High Average Utility Itemsets* (HAUIs). *GHAIM* effectively overcomes the challenges faced by these earlier strategies [4, 6].

In the first database scan, *GHAIM* computes *auub* values for each item and sorts them in ascending order of *auub values*. A second scan updates *auub* values, and items with *auub* values below the *Suffix Minimum Average-Utility* (SMAU) threshold are removed, applying the *auub* pruning strategy. After this step, two key data structures are created: the *Revised Average Utility List* (RAUL) and the *Estimated Average-Utility Co-occurrence Matrix* (EACM), both based on the sorted revised database.

During the mining phase, *GHAIM* calculates an efficient upper-bound utility using *remu*(*eubr*), a tighter bound that helps reduce the search space. In the construction phase, each itemset stores necessary information in the *RAUL* structure, including the *SMAU* value, and applies *LA* pruning to optimize the mining process. Through these enhancements, *GHAIM* improves efficiency in discovering *HAUI*s.

III. My Proposed Algorithm

In this section, we introduce a tree-based algorithm designed to minimize database scans while eliminating the additional time required for the join process. Moreover, this algorithm efficiently mines *HAUI*s in continuously growing databases with multiple minimum thresholds.

A. Basic Ideas

Fig.1 illustrates database *DB*, which is used in our example. The notation $Q(I, T_n)$ represents the quantity of item I in transaction T_n. For instance, for item b in transaction T_5, we have $Q(b, T_5) = 2$. The profit of an item I is denoted as $Pr(I)$. As shown in Fig.2, the profit table indicates $Pr(a) = 3$. Using this, we can compute the utility of an item I in a transaction T_n, represented as $u_{ti}(I, T_n)$. For an itemset, the total utility is the sum of the individual item utilities.

TID	(item, quantity)
T_1	(a, 5) (b, 2) (e, 2) (f, 1)
T_2	(a, 2) (d, 1) (e, 1) (f, 2)
T_3	(c, 1) (d, 2) (f, 1)
T_4	(a, 1) (d, 3) (f, 1)
T_5	(a, 2) (b, 2) (d, 1) (e, 1) (f, 2)

Fig. 1. The example database *DB*

Item	Profit
a	3
b	1
c	2
d	2
e	1
f	1

Fig. 2. The example of the profit table

Unlike traditional utility, average utility considers the length of the itemset. When evaluating a single item *I*, its average utility per transaction is simply its utility:

$$au_{ti}(I,T_n) = \frac{u_{ti}(I,T_n)}{1} = u_{ti}(I,T_n)$$

For an itemset X, the real length of X is considered, and the average utility is calculated as:

$$au_t(I,T_n) = \frac{u_{ti}(X,T_n)}{|X|}$$

To determine the total average utility across the entire database, we sum the average utility values from all transactions where the item or itemset appears. For example, for item a, we compute:

$$au_D(a) = au_{ti}(a,T_1) + au_{ti}(a,T_2) + au_{ti}(a,T_4) + au_{ti}(a,T_5)$$

Similarly, for itemset *ab*:

$$au_D(ab) = au_{ti}(ab,T_1) + au_{ti}(ab,T_5)$$

To determine whether an item or itemset qualifies as a *HAUI*, we compare its total average utility (au_D) with its corresponding threshold from the *Multiple Average-Utility Threshold Value* (MATV) table, shown in Fig.3.

Item	MATV
a	7
b	6
c	7
d	5
e	9
f	8

Fig. 3. The minimum utility threshold (MATV) of each item

For example, given *MATV*(a) = 7 and au_D(a) = 30, itemset {a} (*a* 1-itemset) is classified as a *HAUI*. For itemset {ab}, the threshold is calculated as the average of the individual *MATV* values:

$$\text{MATV(ab)} = \frac{MATV(a) + MATV(b)}{2} = \frac{7+6}{2} = 6.5$$

Since $au_D(ab) = 12.5$ is greater than $MATV(ab) = 6.5$, itemset {ab} is also considered a *HAUI*.

During the mining process of *HAUI*s, an upper bound is defined to reduce the search space. The most commonly used upper bound is *auub* values. It is calculated by summing the *Transaction Maximum Utility* (tmu) values of all transactions where the itemset X appears. Note that Fig.4 displays all the *auub* values used in our database *DB*. Here, *tmu* represents the highest utility value within a given transaction T_n. This upper bound helps eliminate unpromising itemsets early, improving mining efficiency.

Item	*auub*
a	33
b	21
c	4
d	22
e	27
f	37

Fig. 4. The auub value of each item

After calculating the *auub* for all items, we first create a sorted item list in descending order based on their *auub* values. This sorting ensures that items with higher upper bounds are prioritized during the mining process. Next, our algorithm constructs a *TUR*-Tree following the order defined by the sorted item list. The *TUR*-Tree structure efficiently organizes transaction data, reducing the search space and improving mining performance.

In the *TUR*-Tree structure, the *Remaining Maximum Utility Excluding* itemset X (rmue) represents the highest utility among all items in a transaction, excluding those in itemset X. For example, in transaction T_2, after sorting, we have T_2 = (*d*, *e*, *a*, *f*). The *rmue* of itemset {*d*, *e*} in T_2 is calculated as:

$$rmue(de,T_2) = max(uti(a), uti(f)) = \max(6,2) = 6$$

Another important function is the *Suffix Minimum Average-Utility* (smau), which sets a minimum threshold by comparing *MATV* values of itemsets and their succeeding items in the sorted order. If we compute *smau*(*ba*), we consider the *MATV* values of *b*, *a*, and any items appearing after itemset {*b*, *a*}, which is only item *f*.

$$smau(ba) = min(MATV(b), MATV(a), MATV(f)) = \min(6,7,8) = 6$$

B. Data Structures

Based on three input datasets that include database *DB*, the profit table, and the *MATV* table (Fig.1, 2, and 3, respectively), we propose six key data structures for the mining process. These include: (1) the item-based table, (2) the transaction set table, (3) the *auub* table, (4) the sorted item list, (5) the *SMAU* table, and (6) the *EMURUM* table.

1) The Item-Based Table

The item-based table is implemented using a *HashMap*, a data structure in *Java* [8]. When constructing the *TUR*-Tree, items, *tid* (transaction *ID*), utility, and *rmue* are inserted sequentially based on the sorted item list, eliminating the need for re-sorting. Unlike the *GHAIM* algorithm, which requires a sorting step, our approach streamlines the process.

As shown in Fig.5, the item-based table stores the utility of each item ($item_k$) in each transaction (T_i). For instance, the utility of item *a* in transaction T_1 is 15. This table also helps construct the transaction set table and remains permanently in use. In an incremental mining process, if new data changes the item order, the item-based table is essential for reconstructing the *TUR*-Tree. It efficiently stores utility values for each item within specific transactions (T_n).

TID\item	a	b	c	d	e	f
T_1	15	2	0	0	2	1
T_2	6	0	0	2	1	2
T_3	0	0	2	4	0	1
T_4	3	0	0	6	0	1
T_5	6	2	0	2	1	2

Fig. 5. The item-based table (IBT)

2) The Transaction Set Table

The transaction set table, shown in Fig.6, is derived from the item-based table. It records the transactions in which each item appears based on its utility value. If an item's utility is greater than 0, it means the item is present in that transaction, and its *tid* will be included in the set. Conversely, if the utility value is 0, the item does not exist in that transaction and will not appear in the set.

The primary function of the transaction set table is to quickly locate transactions containing specific itemsets. It also plays a crucial role in the pruning strategy, as an empty transaction set indicates that the corresponding itemset is no longer relevant for further searching. The detailed process of utilizing the transaction set table in mining will be discussed later.

itemset \ transaction set	The transactions where the itemset appears
a	T_1,T_2,T_4,T_5
b	T_1,T_5
c	T_3
d	T_2,T_3,T_4,T_5
e	T_1,T_2,T_5
f	T_1,T_2,T_3,T_4,T_5

Fig. 6. The Transaction Set Table (TST)

C. The Construction of the TUR-Tree

The TUR-Tree is built from the item-based table by inserting items in the order of the sorted item list. During insertion, items are added following the sorted order of *auub* values, maintaining the same transaction within the same path. As shown in Fig.7, the *TUR*-Tree consists of four types of links:

- Child link – Connects a parent node to its child nodes.
- Parent link – Establishes the hierarchical structure.
- Next link – Helps locate the next occurrence of the same item for mining.
- Header table link – Serves as the starting point for the mining process.

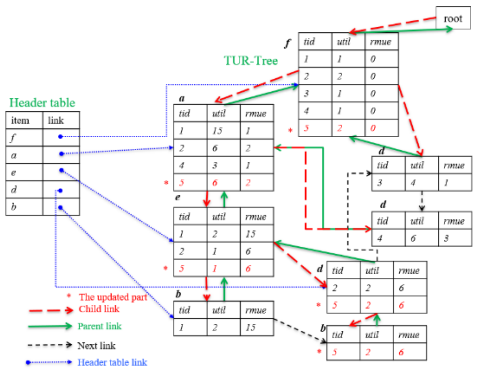

Fig. 7. The TUR-Tree after the insertion of sorted transactions

During insertion, items are added following the sorted order of *auub* values, maintaining the same transaction within the same path. Fig.7 shows the construction of the *TUR*-Tree after the insertion of sorted transactions.

D. Pruning Strategies

Our algorithm incorporates multiple pruning strategies to enhance efficiency, including:

- *AUUB* Pruning Strategy [7] – Eliminates itemsets whose *auub* values fall below the *smau* value.
- Transaction Set Pruning Strategy – Uses the transaction set table to quickly determine if an itemset is empty, avoiding unnecessary searches.
- *TURUB* Pruning Strategy [7, 2] – Applies the *Transaction-Rival Upper Bound* (TURUB) to discard unpromising candidates early.
- *EMURUM* Pruning Strategy [6, 7] – Utilizes the *Estimated Maximum between Utility and Remaining Utility Matrix* (EMURUM) to further reduce the search space.

By integrating these strategies, our algorithm efficiently identifies *HAUI*s while minimizing computation.

E. The Mining Process for the Static Database

In Fig.8, we present the flowchart of the static database mining step. Our approach begins with a *depth-first search* (DFS) following the ascending order of *auub* values.

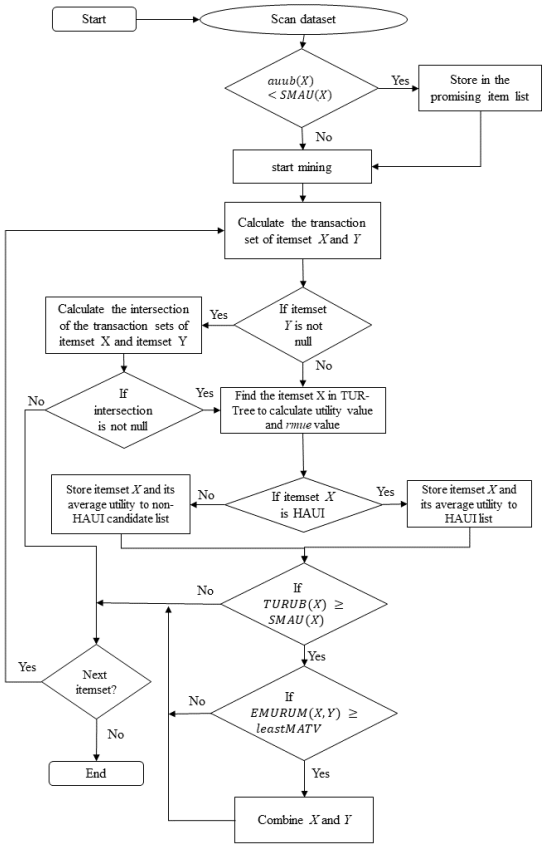

Fig. 8. The flowchart of the static database mining step

Using database *DB* (Fig.1) as an example, we analyze itemset *daf*. The processing order is [*b*, *d*, *e*, *a*, *f*] since item *c* has been pruned using the *auub* pruning strategy. First, we examine the *tid* check set to determine whether it is empty. If it were empty, the transaction set pruning strategy would allow us to skip searching the *TUR*-Tree. However, since *tid* check set = {T_2, T_4, T_5}, we proceed with mining. We begin by locating item *d* in the header table and follow its link to find the first node where *d* appears. The matching *tid*s are {T_2, T_5}, indicating a non-empty intersection. We then accumulate the utility of itemset *daf* for T_2 and T_5. Next, using the parent link, we locate nodes *a* and *f*, continuing the accumulation.

The last visited node is *f*, where we compute *rmue* for T_2 and T_5, both of which are 0. Since *f* is the last item in *daf*, we backtrack to find the next *d* node. After processing T_2 and T_5, we remove them from the *tid* check set and verify if it is empty. Next, we continue searching for item *d* in the next link that intersects with our *tid* check set. We find *tid* = 4 in node d and follow the parent link to locate nodes *a* and *f* for *tid* = 4. After processing, we remove T_4 from the *tid* check set, which now becomes empty. Since there are no remaining *tid*s to process, we stop searching for further *d* nodes.

Now, we compute the *TURUB* value for itemset *daf*. Since item *f* is the last item in *daf* and has the largest *auub* in the sorted list, its *rmue* is 0. Therefore, *rmue*(*daf*, T_2) = 0, *rmue*(*daf*, T_4) = 0, *rmue*(*daf*, T_5) = 0, leading to *TURUB*(*daf*) = 0.

F. The Mining Process of the Incremental Database

In incremental mining, two possible scenarios can occur: unchanged order and changed order. Before discussing these cases, we first define two special item types: revealed items and skippable items. A revealed item was previously pruned due to the *auub* pruning strategy, but as new data increases its *auub* value, it is no longer pruned and is removed from the promising item list. A skippable item was present in the original dataset but does not appear in the updated data. In certain cases, this allows us to skip the item during depth-first search, optimizing the mining process.

When the item order remains unchanged, only certain data structures need to be updated, including the item-based table, *auub* table, *HAUI List*, Non-*HAUI* Candidate *List*, *TURUB List*, and *EMURUM*. In this case, the *TUR*-Tree remains unaffected, which significantly improves efficiency. Compared to situations where the order changes, much of the previously created data can still be used, leading to faster performance.

When the item order changes, more data structures are affected, requiring significant updates. In this case, the *TURUB List* becomes unusable, and the *TUR*-Tree must be reconstructed to reflect the new order. Since the previous mining structure is no longer valid, the system must rebuild key data structures, leading to a higher computational cost compared to the unchanged order scenario. Despite this, updating the item-based table, *auub* table, *HAUI List*, Non-*HAUI Candidate List*, and *EMURUM* table remains essential for accurate incremental mining.

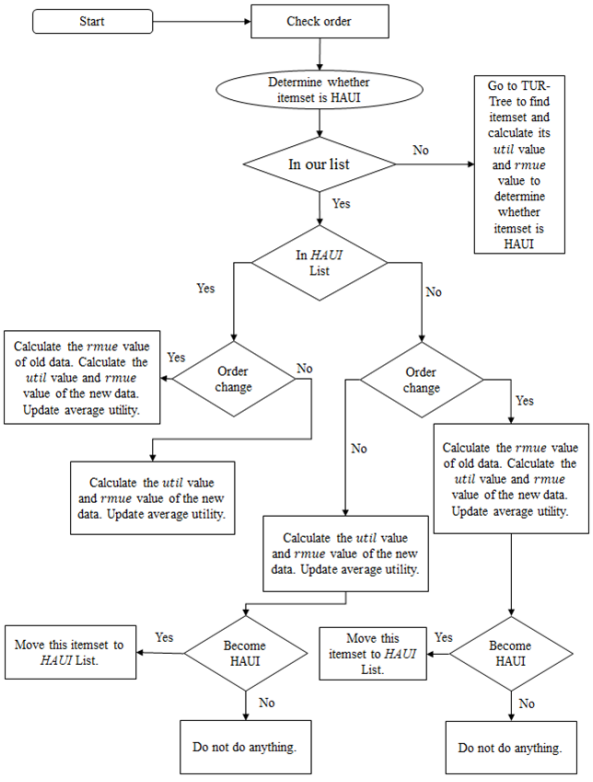

Fig. 9. The Flowchart to determine whether an itemset is in the HAUI List, the Non-HAUI Candidate List, or not in the above two lists

When the order changes, the originally stored *TURUB List* becomes invalid because modifying its values directly would lead to incorrect results. As a result, the *TURUB List* cannot be used, and the *TUR*-Tree must be reconstructed to match the new order. After updating the necessary data structures—including the item-based table, *auub* table, *HAUI List*, Non-*HAUI Candidate List*, and *EMURUM* table—the mining process can begin. The steps for mining remain similar to those in static database mining, with one key difference: determining *HAUI* must follow the process outlined in Fig. 9. This is because the *HAUI* List and Non-*HAUI Candidate List* already contain values from previous mining iterations.

IV. Performance

This study evaluates the performance of the proposed *TURHAIM* algorithm in comparison with the *GHAIM* algorithm [7] using both synthetic and real-world datasets.

A. Synthetic Datasets

In the synthetic dataset, the profit values and quantities of items were randomly assigned within the range of 1 to 15. The dataset was generated based on three key parameters: (1) NIT (*Number of Different Items*), (2) TN (*Total Number of Transactions*) and (3) MT (*Maximum Number of Items per Transaction*). By adjusting these parameters, we controlled the dataset's density.

To evaluate the efficiency of the proposed *TURHAIM* algorithm, we compare its running time with the *GHAIM* algorithm using synthetic datasets. Specifically, we conduct experiments on the datasets: NIT200TN100000MT15. Fig. 10 shows that *TURHAIM* outperforms *GHAIM* in running time on a static synthetic database, demonstrating higher efficiency.

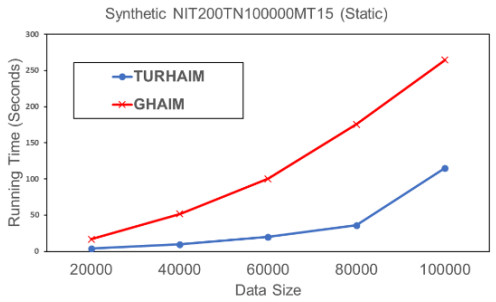

Fig. 10. A comparison of the running time between our algorithm and the GHAIM algorithm based on different data sizes using the synthetic data of NIT200TN100000MT15

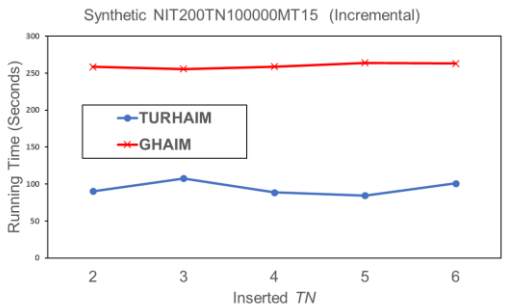

Fig. 11. A comparison of the performance between our algorithm and the GHAIM algorithm in terms of processing time by using synthetic data of NIT200TN100000MT15 under varying values of the inserted TN

Next, we analyze the performance of NIT200TN100000MT15 in an incremental database setting. We examine how running time changes with variations in the inserted T N parameter. Fig.11 shows the processing time of our algorithm is consistently lower than that of the GHAIM algorithm.

B. *Real Datasets*

To further evaluate the robustness of the *TURHAIM* algorithm, experiments were conducted on one real-world datasets: *Connect* [4]. Fig. 12 demonstrates this trend in the connect real database, where our algorithm consistently outperforms the *GHAIM* algorithm in running time. The performance gap widens as the data size increases.

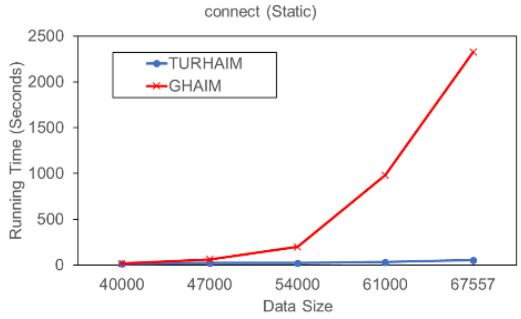

Fig. 12. A comparison of the running time between our algorithm and the GHAIM algorithm based on different data sizes using the real data of connect

Next, we analyze the performance of the connect dataset under the incremental database. As shown in Fig.13, our algorithm consistently outperforms the *GHAIM* algorithm in terms of running time.

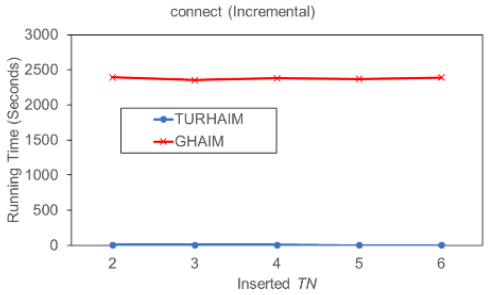

Fig. 13. A comparison of the running time between our algorithm and the GHAIM algorithm using the real data of connect under varying values of the inserted TN

V. CONCLUSION

In this paper, we present a *TURHAIM* algorithm designed to discover high average utility itemsets with multiple minimum thresholds in an incremental database. Our approach requires only a single database scan. When new data is inserted, provided the order remains unchanged, we need to scan only the updated portion of the database, rather than the entire dataset. We compare the performance of our *TURHAIM* algorithm with the *GHAIM* algorithm under both static and incremental database conditions. Experimental results show that our algorithm outperforms *GHAIM* in terms of efficiency.

ACKNOWLEDGMENT

This research was supported in part by the Office of Research and Development, National Sun Yat-sen University under Grant No. 14DS02.

REFERENCES

[1] M.-J. Gao, J.-X. Lin, and J.-W. Wu, "An Efficient Algorithm for High Average Utility Itemset Mining with Buffered Average Utility-List," Proc. of 2020 7th International Conference on Information Science and Control Engineering (ICISCE), pp. 224–228, 2020.

[2] J. M.-T. Wu, J. C.-W. Lin, M. Pirouz, and P. Fournier-Viger, "TUB-HAUPM: Tighter Upper Bound for Mining High Average-Utility Patterns," IEEE Access, Vol. 6, pp. 18655–18669, April 2018.

[3] U. Yun, H. Nam, J. Kim, H. Kim, Y. Baek, J. Lee, E. Yoon, T. Truong, B. Vo, and W. Pedrycz, "Efficient Transaction Deleting Approach of Pre-Large Based High Utility Pattern Mining in Dynamic Databases," Future Generation Computer Systems, Vol. 103, pp. 58–78, February 2020

[4] S. Krishnamoorthy, "Mining Top-k High Utility Itemsets with Effective Threshold Raising Strategies," Expert Systems with Applications, Vol. 117, pp. 148–165, March 2019.

[5] S. Krishnamoorthy, "Efficient Mining of High Utility Itemsets with Multiple Minimum Utility Thresholds," Engineering Applications of Artificial Intelligence, Vol. 69, pp. 112–126, March 2018.

[6] J. C.-W. Lin, S. Ren, and P. Fournier-Viger, "MEMU: More Efficient Algorithm to Mine High Average-Utility Patterns with Multiple Minimum Average-Utility Thresholds," IEEE Access, Vol. 6, pp. 7593–7609, March 2018.

[7] K. K. Sethi and D. Ramesh, "High Average-Utility Itemset Mining with Multiple Minimum Utility Threshold: A Generalized Approach," Engineering Applications of Artificial Intelligence, Vol. 96, pp. 103933–103948, November 2020.

[8] E. Sciore, Sciore, and Gennick, Java Program Design. Springer, 2019.

[9] P. Fournier-Viger, C. W. Lin, A. Gomariz, T. Gueniche, Z. D. A. Soltani, and H. T. Lam, "The Spmf Open-Source Data Mining Library Version 2," Proc. 19th European Conf. on Principles of Data Mining and Knowledge Discovery (PKDD 2016) Part III, pp. 36–40, 2016.