TRƯỜNG ĐẠI HỌC FPT

GRADUATION THESIS REPORT

- Artificial Intelligence -

Mining Top-K Cross-level High Utility Itemset

Supervisor

Le Dinh Huynh

Group Members

Nguyen Tuan Truong	HE150138
Nguyen Duc Chinh	HE150974
Nguyen Khac Tue	HE150066

ACKNOWLEDGEMENT

We are grateful for the assistance and support given during the implementation process by the organizations, agencies, and individuals. We succeeded in finishing the assignment and earning the Bachelor of Artificial Intelligence from FPT University as a consequence.

Our group would first want to express its vibrant and creative environment in which we are grateful to FPT University for providing us with a chance to learn and take part in practical experiences, gain priceless life experience, and establish strong personal values.

Secondly, we would like to thank our supervisor Le Dinh Huynh ,Tran Van Ha and Phan Duy Hung, for all of his guidance, support, and encouragement during the graduation project. Without his expertise and exhortation, we would not have been able to complete this project successfully.

Thirdly, we would like to express our gratitude to the teachers at the Faculty of Artificial intelligence at FPT University for their lectures, specialized knowledge, and life experiences over the last four years, all of which have been very beneficial to us in developing these materials.

Fourthly, without the participation of every group member in order to discuss, make observations, and conduct factual surveys, the project would be difficult to complete. We have also developed a respect for fleeting times by engaging in discussions and working together to meet deadlines.

Last but not least, we would want to express our gratitude to all of our friends, family, and parents for their inspiration and assistance in finishing this project.

FPT University Hanoi, Semester Summer 2023

The authors of this thesis

AUTHOR'S CONTRIBUTIONS

To begin with, we propose an efficient algorithm named TKC-E, which relies on all the proposed techniques to mine cross-level HUIs in taxonomy quantitative databases.

Next, the TKC-E algorithm introduces an efficient technique to reduce the cost of database scans named Database Projection. These techniques, respectively, perform database projections identical in each projected database using a linear time and space implementation that reduce the size of the database as larger itemsets are explored, and thus considerably decrease the cost of database scans.

Based on the concept of item taxonomy and item generalization, we are applied 2 novel pruning strategies named revised sub-tree utility and local utility, which rely on a series of upper bounds and properties to discard unpromising itemsets. They are also mathematically proven to safely reduce the search space, which can considerably reduce the runtime and memory consumption.

Finally, the TKC-E algorithm introduces a new method to compute the utility of generalized items at the same time as specialized items. It also helps to accurately reevaluate the upper bounds to prune cross-level candidates.

ABSTRACT

High utility itemset (HUI) mining extracts frequent itemsets with high utility values from transactional databases. Traditional algorithms have limitations in detecting relationships between items and categories across multiple levels of a taxonomy-based database. Multi-level and cross-level algorithms have been proposed to address this issue while top-k algorithms find the top-k HUIs with the highest utility values. FEACP and TKC algorithms were proposed for HUI mining with high efficiency. However, they suffer from scalability and efficiency issues when dealing with large datasets. To overcome these limitations, we propose a new algorithm called TKC-E (Efficient Top-K Cross-level high utility itemset miner), which combines the strengths of FEACP and TKC while applying efficient strategies to identify cross-level HUIs in taxonomy-based databases, resulting in significantly improved scalability and efficiency. Experimental results show that TKC-E outperforms TKC in terms of processing speed and memory usage, with up to 4 times memory and 60 times runtime improvements on sparse and dense datasets, respectively.

Keywords: TKC-E, High utility itemset, Top-k, Cross-level, taxonomy

List of Abbreviations

S.No	Abbreviations	Description
1	ARM	Association Rule Mining
2	CLHUIs	Cross-Level High Utility Itemsets
3	DFS	Deep first search
4	EUCP	Estimated Utility Co-occurrence Pruning
5	EUCS	Estimated Utility Co-occurrence Structure
6	FIM	Frequent Itemset Mining
7	GHUI	Generalized High-Utility Itemsets
8	HUI	High utility itemset
9	HUIM	High Utility Itemset Mining
10	lu	Local utility
11	PKHUIs	Potential top-k high utility itemsets
12	su	Sub-tree utility
13	TKC-E	Efficient Top-K Cross-level high utility itemset miner
14	TU	Total utility
15	TWU	Transaction Weighted Utilization

List of Table

Table of Contents

1. Introduction

Data mining plays an important role in the field of knowledge discovery. By analyzing the characteristics of a dataset, data mining algorithms can reveal useful information for users. The discovered characteristics, called itemsets, help us understand the data better and support decision-making for many real-world applications [1]. A research method focused on finding frequently occurring sets of items, called Frequent Itemset Mining (FIM), was proposed by Agrawal and Skirant in 1994 [2]. FIM algorithms rely on a support framework to discover itemsets that appear no less frequently than the minimum support threshold. However, FIM treats all items of the transaction database equally, assuming that they are all equally important. This assumption is a major drawback of FIM, as for many real-world applications, items in the transaction database may have different levels of importance. For example, essential daily groceries are purchased more frequently than electronic devices. As a result, the FIM algorithm will provide a list of itemsets that frequently occur in the data, which may not be suitable in practice, as many items with low occurrence frequency may actually generate high profits [3].

The Association Rule Mining (ARM) method was introduced by R. Agrawal in 1993 [1] to search for association rules among itemsets in transactional databases. This method helps to discover relationships and dependencies among itemsets, thereby revealing potential relationships between items and customers. The two important measures in ARM are support and confidence. Support measures the frequency of occurrence of itemsets in transactional data, while confidence measures the degree of certainty of combined rules. ARM uses support and confidence thresholds to filter out uninteresting rules and focus on more important ones. However, ARM also has some limitations. One similar disadvantage to FIM is the assumption that all items in the transactional database are equally important, which may not be true in many real-world situations. Additionally, ARM can generate a large number of combined rules, which may be difficult to explain and may not be relevant to the current issue [4].

Some practical issues in knowledge discovery from transactional data cannot be fully addressed by FIM and ARM techniques. Particularly, efficiently searching for high utility itemsets in transactional databases needs to be addressed [3] [4]. To solve this problem, High Utility Itemsets (HUI) methods have been developed. HUI methods search for itemsets with higher total utility values than a given threshold and contribute significantly to knowledge discovery from transactional data [5].

HUI algorithms use utility values to measure the value of each itemset in the database. These values can be calculated using different algorithms such as Apriori-based, Pattern-

growth-based, or Tree-based [6]. Then, HUI algorithms use appropriate search methods to find itemsets with higher utility values than a predetermined threshold.

HUI algorithms have high practical applications and are widely applied in fields such as retail, banking, insurance, healthcare, and many others [7]. In retail, HUI algorithms are used to search for high utility products and services, helping businesses increase sales and improve customer experience. In banking and insurance, HUI algorithms help to search for high utility customers, helping organizations optimize marketing strategies and provide suitable products and services. In healthcare, HUI algorithms can be used to detect high-risk patients or potential symptoms, thereby making appropriate treatment decisions.

Some popular HUI algorithms include Two-Phase [8], d2HUP [9] [10], UP-Growth [11], HUI-Miner [12], FHM [13] and EFIM [14]. Although most of these algorithms are effective, they ignore the fact that items in transaction databases are often organized into categories and subcategories of a taxonomy.

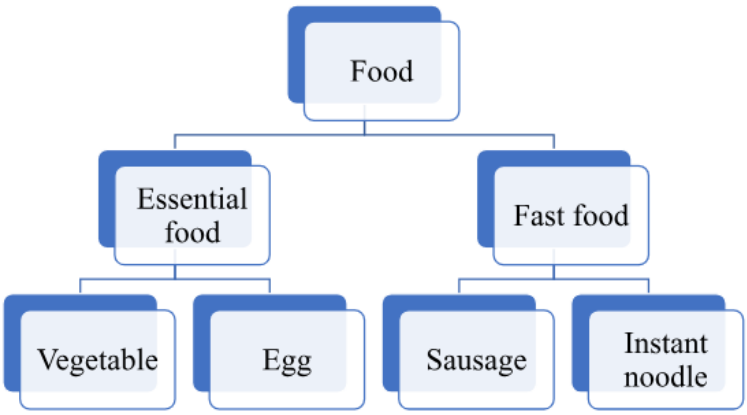

Fig. 1. A taxonomy of food.

For example, Fig. 1 shows the classification of food items sold in a supermarket. The categories of vegetables and eggs are classified as essential foods, while sausages and instant noodles are grouped under fast food. In addition, essential foods and fast food are subcategories of the broader category of general food. Taxonomy can play an important role in mining itemsets in a database, as it allows grouping of items semantically into generalized itemsets, i.e., itemsets that contain generalized items (categories). In Fig. 1, the generalized items are Food, Essential food, and Fast food. As traditional High Utility Itemsets Miner (HUIM) algorithms ignore classification information, they can only discover itemsets containing items that appear at the lowest level of the transaction database's classification tree (Vegetable, Egg, Sausage, Instant

noodle). Therefore, items such as Food, Essential food, and Fast food cannot be found in the output of these algorithms, even if they are HUI.

Recently, numerous studies have been conducted to address the problem of mining multi-level and cross-level high utility itemsets (HUIs). Examples include ML-HUI Miner [15], MLHMiner [16], CLH-Miner [17], FEACP [18]. These studies have focused on proposing various ways to preprocess and organize data and developing new algorithms to simultaneously search for multi-level and cross-level HUIs while improving runtime and memory usage.

The Top-k approach is a variant of HUIM that is particularly useful when users are interested in finding the most important itemsets, rather than all high utility itemsets. In this approach, the objective is to discover the top-k high utility itemsets, where k is a parameter specified by the user [19]. This technique can effectively reduce the number of itemsets that need to be examined and is more suitable for applications that focus on the most important itemsets. The top-k method is a powerful and flexible technique capable of addressing different user options and application requirements in mining high utility itemsets.

In this thesis, we propose a new algorithm called TKC-E (Efficient Top-K Cross-level high utility itemset miner) to address the above problem. Through experiments, TKC-E has demonstrated improved performance and high efficiency, while discovering multi-level and cross-level relationships in the data.

The rest of this thesis is organized as follows. Section 2 reviews the literature on HUIM and Top-K. Section 3 provides an overview and definition of the problem. Section 4 presents the proposed data pruning and processing strategies and the TKC-E algorithm for mining cross-level HUIs. Section 5 then reports the results from an extended evaluation. Finally, Section 6 provides conclusions and plans for future work.

2. Related work

Below is a summary table of the algorithms that we will discuss in this section to provide an overview of the research methods that have been applied in the related field.

Table 1. Summary table of algorithms.

Author	Year of publication	Algorithm abbreviation	Algorithm name	Type
Tseng et al.	2010	UP-Growth [11]	Utility Pattern-Growth	Mining High utility itemsets
Liu & Qu	2012	HUI-Miner [12]	High Utility Itemset Miner	Mining High utility itemsets
Fournier-Viger et al.	2014	FHM [13]	Frequent High Utility Itemset Mining	Mining High utility itemsets
Zida et al.	2016	EFIM [14]	Efficient high-utility Itemset Mining	Mining High utility itemsets
Luca Cagliero et al.	2017	ML-HUI Miner [15]	Multiple-Level High-Utility Itemset Miner	Mining Multiple-Level High utility itemsets
N.T. Tung et al.	2021	MLHMiner [16]	Multiple-Level HMiner	Mining Multiple-Level High utility itemsets
Fournier-Viger et al.	2020	CLH-Miner [17]	Cross-level high utility itemset mining	Mining Cross-Level High utility itemsets
N.T. Tung et al.	2021	FEACP [18]	Fast and Efficient Algorithm for Cross-level high-utility Pattern mining	Mining Cross-Level High utility itemsets
Cheng-Wei Wu et al.	2012/2016	TKU [20] [21]	Top-K High Utility Itemset Miner	Mining Top-k HUIs
Vincent S. Tseng et al.	2016	TKO [21]	Top-K High Utility Itemset Miner in One Phase	Mining Top-k HUIs
Mourad Nouioua et al.	2020	TKC [22]	Top-K Cross-level high utility itemset miner	Mining Cross-level Top-k HUIs

FIM and ARM are two important algorithms in the field of data mining related to the relationships among items in a dataset. FIM is the process of searching for sets of items that frequently appear together in transactions of the dataset. ARM is the process of searching for association rules between items in frequent itemsets, providing recommendations or information relevant to the dataset.

Although FIM and ARM have been widely used in many fields and have many effective applications, they only focus on searching for frequent itemsets and association rules, while ignoring the utility value of the items in those frequent itemsets. Therefore, HUIM was proposed to focus on searching for frequent itemsets with high utility values, based on information related to value, quantity, or the combination of items.

HUIM is a data mining method for exploring a set of items that, when purchased together, provides higher utility value than buying individual items. Utility can be measured using some metrics such as total revenue, profit, or the number of products sold [5]. The high utility itemset mining algorithm can be performed using methods such as UP-Growth [11], HUI-Miner [12], FHM [13] and EFIM [14].

UP-Growth is an efficient algorithm designed for mining high-utility itemsets from transaction databases. It utilizes the UP-Tree data structure to maintain information about high-utility itemsets and achieves high mining performance. By applying four utility reduction strategies, UP-Growth effectively reduces the search space and candidate itemsets. The UP-Growth algorithm requires only two passes through the transaction database to generate potential high-utility itemsets from the UP-Tree. The mining performance is improved by using utility reduction strategies, which enhance efficiency and reduce memory usage during the mining process. Experimental results have demonstrated that UP-Growth is particularly effective when dealing with databases containing long transactions.

The drawback of the UP-Growth algorithm is that it generates a large number of candidates, while the number of high-utility itemsets is much smaller. To address this issue, the HUI-Miner algorithm was proposed in 2012. Unlike UP-Growth, HUI-Miner can mine high-utility itemsets without generating candidate itemsets. This is achieved by using a specialized data structure called the utility-list, which stores information about the value of itemsets and provides crucial information for pruning the search space. HUI-Miner directly mines high utility itemsets from the utility-list constructed from the previously mined database. This not only saves running time but also consumes less memory.

FHM is a method based on the HUI-Miner algorithm. The major difference lies in the improved search space pruning process. FHM utilizes a mechanism called EUCP (Estimated

Utility Co-occurrence Pruning). EUCP operates by constructing a new structure called EUCS (Estimated Utility Co-occurrence Structure). During the search process, EUCP checks the following condition: if there does not exist a triplet (x, y, c) in EUCS such that $c \geq$ minutil, then EUCP immediately prunes a low-value projected extension Pxy and all its indirect extensions. This is accomplished without the need to construct a utility-list, saving time and resources. The purpose of EUCP is to eliminate low-value itemsets without having to check each individual element within them. This ensures that only itemsets with higher potential are further mined, enhancing algorithm efficiency. As a result, FHM can reduce the search space by up to 95% and achieve speeds up to 6 times faster than HUI-Miner.

Runtime and memory are always important issues and need to be improved so that the algorithms give results in less time. Therefore, in 2016, Zida et al. proposed a new algorithm called EFIM, which introduces several new ideas to more efficiently discover high-utility itemsets [14]. Specifically, EFIM relies on two new upper bounds named revised sub-tree utility and local utility to prune the search space, and uses a novel array-based utility counting technique called Fast Utility Counting to calculate these upper bounds in linear time and space. To reduce the cost of database scans, EFIM proposes efficient database projection and transaction merging techniques named High-utility Database Projection and High-utility Transaction Merging. Experimental results show that EFIM is generally two to three orders of magnitude faster than previous algorithms, and its key advantage is low memory consumption.

However, traditional HUIM algorithms do not consider information about hierarchical databases, meaning databases with multiple levels of classification. This makes the algorithms only able to mine HUIs at the same data level and unable to analyze the association between items at different levels in the hierarchical database. This can cause us to miss important HUIs and not fully exploit the characteristics of the data, affecting planning and business processes in practice. To address this issue, some algorithms for mining hierarchical and cross-level HUIs have been proposed, such as ML-HUI Miner [15], MLHMiner [16], CLH-Miner [17] and FEACP [18].

The ML-HUI Miner algorithm, proposed by Cagliero et al. in 2017 [15], focuses on identifying Generalized High-Utility Itemsets (GHUI) - sets of items with high total profit. This algorithm utilizes classification information to enhance the data and explore various levels of abstraction. The process of generating GHUI in ML-HUI Miner is recursive, starting with individual items and then considering pairs of items to form GHUI sets. This recursive process is similar to the FHM algorithm. Moreover, ML-HUI Miner has the capability to extract profitable items and avoid generating uninteresting combinations of items.

The MLHMiner algorithm is a multi-level HUI mining algorithm proposed by N.T.Tung and colleagues in 2020 [16]. This algorithm is designed to address the problem of traditional algorithms in mining large itemsets. MLHMiner uses a multi-level structure to store high utility itemsets, which helps to speed up mining and reduce memory. The MLHMiner algorithm has two main stages: the mining stage and the multi-level structure construction stage. In the mining stage, the algorithm uses a recursive mining method to search for high utility itemsets. In the multi-level structure construction stage, the algorithm uses a criterion to classify high utility itemsets into different levels of the multi-level structure. The MLHMiner algorithm also uses several pruning techniques to reduce the search space, including pruning based on the total utility value of itemsets and pruning based on the total weight of itemsets. Experimental results show that MLHMiner is more efficient than ML-HUI Miner in both of runtime and memory.

One limitation of the ML-HUI Miner and MLHMiner is its inability to find itemsets with items from different classification levels (cross-level HUIs), which may result in missing interesting HUIs. Additionally, the ML-HUI Miner algorithm does not utilize classification to reduce the search space, as it mines each separate classification level.

To overcome the above limitations, CLH-Miner has defined a new problem called Cross-Level High Utility Itemsets (CLHUIs) mining and developed an efficient algorithm to extract all CLHUIs. The CLH-Miner algorithm integrates new pruning strategies to reduce the search space by leveraging the relationships between different abstraction levels. This helps reduce the time and computational resources required to extract CLHUIs from the data. In the original paper [17], the algorithm has discovered interesting patterns from real-world retail data. However, experimental results have shown that CLH-Miner takes more runtime compared to HUI-Miner and ML-HUI Miner. Additionally, it also consumes more memory on certain datasets.

Continuing the improvement of CLH-Miner in 2021, N.T. Tung, Loan T.T. Nguyen, and colleagues proposed the FEACP algorithm [18]. FEACP is an algorithm for mining high-level cross-level utility itemsets in quantitative databases. This algorithm operates by scanning through the database three times, from general computation to classification, while applying search space reduction techniques. To find all Cross-Level HUIs, FEACP performs a deep search based on DFS to recursively traverse the search space and expand the initial frequent itemsets. This algorithm relies on the local utility and sub-tree utility techniques to calculate the utility value for itemsets and their sub-branches. By using upper bounds on utility and pruning

strategies, FEACP can reduce the number of itemsets to be considered in the search space, increase mining efficiency, and find useful information itemsets.

Although the above algorithms can reveal interesting patterns, one issue is that setting the minimum utility threshold is not intuitive and greatly affects the number of patterns found as well as the algorithm's performance. If the user sets the threshold too low, a large number of patterns will be found, and the runtime may be very long, while if the threshold is set too high, very few patterns will be found. Therefore, users often have to run the algorithm multiple times to find a suitable threshold value to obtain just enough patterns. This is why the Top-K algorithm is proposed. The Top-K algorithm allows users to set the number of patterns k to be explored instead of setting a minimum utility threshold. This task is more intuitive for users because they can directly choose the number of patterns without using a minimum utility threshold, which is difficult to set and depends on the hidden characteristics of the database.

Several algorithms have been developed for top-k HUIM, such as TKU [20] [21] and TKO [21]. These algorithms typically operate by cutting down the search space based on utility limits, effectively reducing the computation cost and improving the efficiency of the mining process. The top-k method is a powerful and flexible technique that can address various user options and application requirements in mining high-utility itemsets.

The TKU algorithm was proposed in 2012 by the author team Cheng Wei Wu, Bai-En Shie, Philip S. Yu, Vincent S. Tseng [20]. This method is an extension of UP-Growth [11]. TKU uses the UP-Tree structure of UP-Growth to maintain information about transactions and top-k HUIs. The framework of TKU consists of three parts: (1) construction of UP-Tree, (2) generation of potential top-k high utility itemsets (PKHUIs) from the UP-Tree, and (3) identification of top-k HUIs from the set of PKHUIs.

In 2015, TKU continued to be improved by Philippe Fournier-Viger and his colleagues, combined with a new algorithm called TKO [21]. The new TKU algorithm used a new pruning technique to reduce the number of candidate items that needed to be evaluated, while also using an algorithm to generate fewer candidate itemsets than the previous algorithm. In addition, the way TKU utility was calculated has also changed, making it more efficient than before.

TKO is a one-phase algorithm, meaning it exploits top-k high utility itemsets in one pass [21]. This makes it significantly faster than other top-k algorithms. TKO uses a utility-list data structure to store utility information of itemsets in the database. It also uses a two-level pruning strategy to prune the search space and improve the efficiency of the algorithm.

However, both TKU and TKO have the drawback of using non-categorical data structures - meaning unclassified data. This limits their ability to exploit High Utility Items

across different data levels, reducing the effectiveness in extracting complex patterns and potentially missing opportunities to exploit potential data patterns.

To address this issue, in 2020, Philippe Fournier-Viger and his colleagues proposed a new algorithm called TKC [22]. TKC uses a Tax-utility-lists data structure which consists of a list of itemsets sorted in a similar order to a taxonomy table and containing information about the utility value and utility limit of each itemset. Using tax-utility-lists allows for quick computation of the utility value of itemsets without having to scan the database, helping to increase mining speed and reduce processing time. The TKC algorithm starts searching from single items and uses a priority queue to keep track of the current top-k patterns found. When a new cross-level HUI is found, it is inserted into the priority queue and if the queue contains at least k patterns, the minutil threshold is increased to the utility value of the kth pattern in the queue. All patterns that do not meet the new minutil threshold are removed from the queue. Optimization is used to speed up the increase of the minutil value by raising the minutil threshold with the kth largest utility value in the priority queue. Thus, TKC helps reduce the search space and ensure that all top-k cross-level HUIs are found. However, the runtime and memory usage of TKC are almost indistinguishable. In fact, TKC usually consumes more memory than CLH-Miner.

In this thesis, we proposed a new algorithm named TKC-E, which analyzes the quantitative database data structure. Additionally, TKC-E utilizes the priority queue of TKC to maintain the current top-k found patterns, and an optimization is used to increase the speed of minutil value growth. The TKC-E algorithm also effectively applies the local utility and sub-tree utility techniques of the FEACP algorithm to optimize and reduce the search space. Experimental results show that TKC-E significantly improves performance compared to TKC and FEACP on real datasets and is confirmed to be effective in exploiting high-level and cross-level utility itemsets on different datasets.

3. Preliminaries and Problem Definition

The type of databases considered in this thesis are quantitative transaction databases with a taxonomy that is used in retail stores. It's quite general and can be used to represent data in many other applications. In this section, we first define the problem and main definition for Cross-level HUIM.

Definition 1 (**Transaction database**)[16]: Let I be a finite set of items $I = \{i_1, i_2,..., i_m\}$. A transaction database is a multiset of transactions $D = \{T_1, T_2,..., T_m\}$ such that for each transaction T_c is a set of items $i \subseteq I$ and T_c has a unique identifier c called its Transaction ID(TID). Each item $i \in I$ is associated with a positive number $p(i)$, called its external utility (e.g. unit profit). For each item i in T_c there is a positive number $q(i, T_c)$ associated with it, which represents its internal utility or purchase quantity in that transaction.

Example 1: Consider the database in Table 2, which will be used as the running example. It contains seven transactions (T_1, T_2, …, T_7). Transaction T_2 indicates that items a, c, e and g appear in this transaction with an internal utility of respectively 2, 6, 2 and 5. Table 3 indicates that the external utility of these items are respectively 5, 1, 3 and 1.

Table 2. A transaction database

TID	Transaction
T1	(a, 1),(c, 1),(d, 1)
T2	(a, 2),(c, 6),(e, 2),(g, 5)
T3	(a, 1),(b, 2),(c, 1),(d, 6),(e, 1),(f, 5)
T4	(b, 4),(c, 3),(d, 3),(e, 1)
T5	(b, 2),(c, 2),(e, 1),(g, 2)
T6	(a, 2),(c, 6),(e, 2)
T7	(c, 1),(d, 2),(e, 1)

Table 3. External utility values

Item	Unit Profit
a	5
b	2
c	1
d	2
e	3
f	1
g	1

Definition 2 (**Taxonomy**)[14-18]: A taxonomy τ is a tree(a directed acyclic graph) defined for a transaction database D. Leaf nodes of the taxonomy represent the different items of I , while internal nodes represent categories of items, which are called generalized items or abstract items. Generalized items represent an abstract category that groups all descendant leaf nodes(items) or all descendant categories into one higher-level category. A child-parent edge between two (generalized) items i, j in τ represents an is-a relationship.

Definition 3 (The sets of generalized items and all items)[17]: The set containing all these generalized items is denoted as *GI*, and the set containing both generalized and leaf items is denoted as $AI = GI \cup I$.

Definition 4 (Generalization relationship)[17]: Let there be a relation $LR \subseteq GI \times I$ such that $(g, i) \in LR$ if there is a path from g to i. And, let there be a relation $GR \subseteq AI \times AI$ such that $(d, j) \in$ GR if there is a path from d to j.

Definition 5 (Leaf / Descendant)[16]: Consider a taxonomy τ and a generalized item g in τ. The leaf items of a generalized item g are all leaves that can be reached by following paths starting from g defined as $Leaf(g, \tau) = \{i \mid (g, i) \in LR\}$. The descendant items of a (generalized) item d is the set $Desc(d, \tau) = \{j \mid (d, j) \in GR\}$. The level denotes the number of edges to be traversed to reach an item d starting from the root node of τ.

Example 2: In the taxonomy of Fig. 2, we can see that $Leaf(\{X\}, \tau) = \{a, b, c\}$ and $Desc(\{X\},\tau) = \{Y, a, b, c\}$, the itemset $\{Y, d\}$ is a descendant of $\{X, Z\}$, and level(c) = 2.

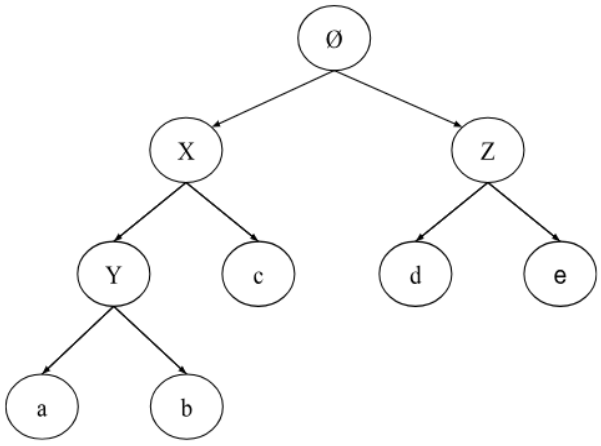

Fig. 2. A taxonomy of items.

Definition 6 (Utility of an item/itemset)[17]: The utility of item i in transaction T_c is defined as $u(i,T_c) = q(i, T_c) \times p(i)$. Similarly, the utility of an itemset P (a group of items i ⊆ I) in transaction T_c is defined as $u(P, T_c) = \sum_{i \in P} u(i, T_c)$. Finally, let g(P) is the set of transactions containing X in D, the utility of an itemset X in a database is defined as $u(P) = \sum_{T_c \in g(P)} u(P, T_c)$.

Example 3: The utility of a in T_1 is $u(a, T_1) = 1 \times 5 = 5$. The utility of $\{a, c\}$ in T_2 is $u(\{a, c\}, T_2) = u(a, T_2) + u(c, T_2) = 2 \times 5 + 6 \times 1 = 16$. The utility of $\{b, c\}$ in the database D is $u(\{b, c\}) = u(\{b, c\}, T_3) + u(\{b, c\}, T4) + u(\{b, c\}, T_5) = 5 + 11 + 6 = 22$.

Definition 7 (Utility of a generalized item/itemset)[17]: The utility of generalized item g in transaction T_c is defined as $u(g, T_c) = \sum_{i \in Leaf(g,\tau)} q(i, T_c) \times p(i)$ .Similarly, let $g(GP)$ is

the set of transactions containing *P* in *D*, the utility of a generalized itemset *GP* in a transaction T_c is calculated by $u(GP, T_c) = \sum_{d \in GP} u(d, T_c)$ .The utility of an itemset *P* in a database is calculated by $u(GP) = \sum_{T_c \in g(GP)} u(GP, T_c)$ where *g(GP) = {$T_c \in D \mid \exists P \subseteq Tc \wedge P$ is a descendant of GP}*.

Example 4: In the taxonomy of Fig. 1, *Z* is a generalized item and $u(Z, T_4) = u(d, T_4) + u(e, T_4) = 3 \times 2 + 1 \times 3 = 9$. The utility of the generalized itemset *{Z, b}* in T_4 is $u(\{Z, b\}, T_4) = u(Z, T_4) + u(b, T_4) = (6 + 3) + 8 = 17$. The utility of the generalized itemset *{Z, b}* in the database is $u(\{Z, b\}) = u(\{Z, b\}, T_3) + u(\{Z, b\}, T_4) + u(\{Z, b\}, T_5) = 17 + 19 + 7 = 43$.

Definition 8 (Cross-level high utility mining)[16]: The problem of cross-level high utility mining is defined as finding all cross-level high-utility itemsets(CLHUIs). A (generalized) itemset *X* is called a CLHUI if $u(X) \geq \mu$ where μ is minutil threshold.

Example 5: If $\mu = 70$, the cross-level high utility itemsets in the database of the running example are: *{X,e}, {Z, X}, {Z, Y}, {Z, Y, c}, {Z, c, a},{e, Y, c}* with respectively a utility of 84, 114, 87, 106, 73 ,80.

Definition 9 (Top-K cross-level high utility mining)[22]: Consider a transaction database *D* and a user-defined parameter k which indicates the desired number of cross-level high utility itemsets. An itemset $X \subseteq AI$ is called a top-k cross-level high utility itemset (top-k CLHUI) if *X* in k-th itemsets in *D* whose utilities are largest. Mining top-k cross-level high utility itemsets consists of identifying top-k CLHUI having the highest utility in database *D*.

Example 6: If k = 3, the top-k cross-level high utility itemsets are: *{Z, X}, {Z, Y, c}, {Z, Y }* with a utility of 114, 106, and 87.

4. Proposed Algorithm

4.1. Search space exploration and pruning techniques

To be able to explore the search space of all itemsets in a systematic way, a processing order $\succ$ is defined on items of AI. According to that total order, two distinct items $a, b \in AI$ are ordered as $a \succ b$ if *level(a) < level(b)*, or if *level(a) = level(b)* $\wedge$ *TWU(a) > TWU(b)*. This ordering between levels ensures that the algorithm considers generalized items before their descendant items, which is important as it enables itemset pruning in the search space. This ordering is defined based on the TWU measure as follows.

Definition 10 (Transaction Weighted Utilization – TWU)[16]: A transaction T_c has its transaction utility defined as $TU(T_c) = \sum_{i \in T_c} u(i, T_c)$. For an itemset $P \subseteq I$, the TWU is

computed as $TWU(P) = \sum_{T_c \in g(P)} TU(T_c)$. Similarly, an itemset $GP \subseteq GI$, the *TWU* is computed as: $TWU(GP) = \sum_{T_c \in g(i \in Leaf(GP,\tau))} TU(T_c)$.

Example 7: the *TU* values of transactions T_1 to T_7 for Table 2 are: 8, 27, 30, 20, 11, 22 and 8, respectively. $TWU(\{a\}) = TU(T_1) + TU(T_2) + TU(T_3) + TU(T_6) = 8 + 27 + 30 + 22 = 87$, $TWU(\{Y\}) = TU(T_1) + TU(T_2) + TU(T_3) + TU(T_4) + TU(T_5) + TU(T_6) = 8 + 27 + 30 + 20 + 11 + 22 = 118$.

Property 1: (Pruning search space using TWU). For any itemset X, if TWU(X) < minutil, then X is a low-utility itemset as well as all its supersets.

Definition 11 (Extension)[16]:An extension of an itemset P is an itemset obtained by adding an item i to P. The set of all extensions of P is defined as $E(P) = \{i \mid i \in AI \wedge w \succ i \text{ and } \forall w \in P, i \notin Desc(w, \tau)\}$

Example 8: If the total order is $X \succ Z \succ c \succ Y \succ e \succ d \succ a \succ b \succ g \succ f$, E(X)= {Z, c, e, d}, E{Z,c}= {Y, a, b}, E{c,e}= {d, a, b}.

Definition 12 (Remaining utility)[16]:The remaining utility of an itemset P in a transaction T_c is defined as $re(P, T_c) = \sum_{i \in T_c \wedge i \in E(P)} u(i, T_c)$.

Example 9: If the total order is $X \succ Z \succ c \succ Y \succ e \succ d \succ a \succ b \succ g \succ f$, re(X, T_2) = 6 +5 = 11, re({c, e}, T_4) = 8+6 = 14.

Definition 13 (Utility-list)[16]: The utility-list of an itemset *X* in a database *D* is a set of tuples such that there is a tuple *(c, iutil, rutil)* for each transaction containing *X*. The iutil and rutil elements of a tuple respectively are the utility of *X* in T_c *(u(X, T_c))* and the remaining utility of *X* in T_c *(re(X, T_c)).*

Example 10: If the total order is $X \succ Z \succ c \succ Y \succ e \succ d \succ a \succ b \succ g \succ f$, then the itemset *{X,d}* has the utility-list: {(1,8,0),(3,22,5),(4,17,0)}.

Definition 14 (Projected database)[17,19]: The set of items obtained by projecting a transaction T_c over an itemset P is denoted as $(T_c)_p$ and defined as $(T_c)_P = \{k \mid k \in T \wedge k \in E(P)\}$. The multiset obtained after projecting itemset P over all transactions in *D*, denoted D_P, is calculated as $D_P = \{(T_c)_P \mid T_c \in D \wedge (T_c)_P \neq \emptyset)\}$.

Example 11: for the database *D* given in Table 2 and P ={d}, the database D_P can be constructed by the following transactions: $(T_1)_P$={a}, $(T_3)_P$={a,b}, $(T_4)_P$={b}.

Definition 15 (Local utility)[17]: Let *P* be an itemset. Consider an item $i \in E(P)$. The local utility of i w.r.t itemset P is defined as $lu(P, i) = \sum_{T_c \in g(P \cup \{i\})} [u(P, T_c) + re(P, T_c)]$. Similarly, if item *z* is a generalized item in taxonomy, the local utility of *z* w.r.t itemset *P* is calculated as $lu(P, z) = \sum_{T_c \in g(P \cup j \in Leaf(z,\tau))} [u(P, T_c) + re(P, T_c)]$.

Example 12: Consider the running example and $P = \{c\}$. We have that *lu(P, a)*= 8 + 22 + 25 + 22 = 77, lu(P, d) = 8 + 25 + 20 + 8 = 61 and lu(P, e) =106.

Theorem 1: (Pruning an item from all sub-trees using the local utility): Let be an itemset *P* and an item $i \in E(P)$. If $lu(P, i) < \mu$, then all extensions of *P* containing *i* are low-utility. So, item *i* can be ignored when exploring all sub-trees of *P*.

Definition 16 (Sub-tree utility)[17]: Let *P* be an itemset. Consider an item $i \in E(P)$. The local utility of i w.r.t itemset *P* is defined as follows:

$$su(P, i) = \sum_{T_c \in g(P \cup \{i\})} [u(P, T_c) + u(i, T_c) + \sum_{j \in T_c \wedge j \in E(P \cup \{i\})} u(j, T_c)]$$

Similarly, if item *z* is a generalized item in taxonomy, the local utility of *z* w.r.t itemset *P* is calculated as follows:

$$su(P, z) = \sum_{T_c \in g(P \cup j \in Leaf(z,\tau))} [u(P, T_c) + \sum_{j \in Leaf(z,\tau)} u(j, T_c) + \sum_{j \in T_c \wedge j \in E(P \cup \{z\})} u(j, T_c)] .$$

Example 13: Consider the running example and $P = \{c\}$. We have that $su(P, a) = 6 + 16 + 10 + 16 = 48$, $su(P, d) = 8 + 22 + 17 + 5 = 52$ and $su(P, e) = 22 + 25 + 20 + 9 + 22 + 8 = 106$.

Theorem 2: (Pruning a sub-tree using the sub-tree utility): Let be an itemset *P* and an item $z \in E(P)$. If $su(P, z) < \mu$, then the single item extension $P \cup \{z\}$ and its extensions are low-utility. In other words, the sub-tree of $P \cup \{z\}$ in the set-enumeration tree can be pruned.

Definition 17 (Primary and secondary items) [17]: Let be an itemset *P*. The primary items of *P* is the set of items defined as $Primary(P) = \{z \mid z \in E(P) \wedge su(P, z) \geq \mu\}$. The secondary items of P is the set of items defined as $Secondary(P) = \{z \mid z \in E(P) \wedge lu(p, z) \geq \mu\}$. Because $lu(p, z) \geq su(P, z)$, $Primary(P) \subseteq Secondary(P)$.

Definition 18 (Utility-Bin array) [17,19]: Consider the set *AI* containing all distinct items in *D* and generalized items in *GI*. The utility-bin array U is an array of length *AI* that has an entry denoted as *U(z)* for each $z \in AI$ called a utility-bin. Each utility-bin can be used to hold a utility value and is initialized to zero.

An algorithm can utilize a utility-bin array to determine the TWU of any item when scanning item z in each transaction T_c is updated as $U[z] = U[z] + TU(T_c)$. For each item z ∈ T ∩ E(P) the sub-tree utility w.r.t. an itemset P as $su[z] = su[z] + u(P, T_c) + u(z, T_c) + \sum_{j \in T_c \wedge j > z} u(j, T_c)]$. It also supports the calculation of the local utility w.r.t. an itemset P as $lu[z] = lu[z] + u(P, T_c) + re(P, T_c)$

In this thesis we extend this array to store and calculate the value of TWU, sub-tree utility and local utility of generalized items in the set GI. When scanning the database to calculate the upper bound, the Utility-Bin array regards generalized items as specialized items. In traditional HUIM algorithms, the upper bounds usually contain only the utility of the promising items. In CLHUI mining, if the upper bounds only contain the utility of the promising specialized items, they will miss out on several generalized items. This is because the utility of these generalized items is aggregated from the unpromising leaf items, which were previously discarded.

4.2. TKC-E Algorithm

The proposed TKC-E algorithm is presented in Algorithm 1. It accepts three input parameters that are a database *D*, a taxonomy τ, and the user-defined number of patterns to be found k.

Algorithm 1: The TKC-E algorithm

input: *D*: a transaction database, τ: a taxonomy, *k*: the number of patterns to be found.
output: the top-k cross-level HUIs.

1. Initializes $\mu = 0$, $P = \{\emptyset\}$ a priority queue *Q* with the top-k cross-level HUIs from AI;
2. Read τ and *D* and use a utility-bin array to calculate to compute *lu (P, z)* of each (generalized) item $z \in AI$;
3. Secondary(*P*) = $\{z \mid z \in AI \wedge lu(P, z) \geq \mu\}$;
4. Compute $\prec$, the total order on items from Level and *TWU* values on Secondary(*P*);
5. Scan D to store each generalized item g ∈ Secondary(*P*) in each transaction, discard every item i ∉ Secondary(*P*) from transactions,sort items in each transaction, delete empty transactions, and then build and store the utility-list of each generalized item;
6. Compute the sub-tree utility *su(P, z)* of each item $z \in$ Secondary(*P*);
7. Primary(*P*) = $\{z \mid z \in AI \wedge su(P, z) \geq \mu\}$;
8. SEARCH (*P, D, Primary(P) , Secondary(P) , k , μ ,Q*);

The overall process is done as follows:

Line #1: Initializes $\mu = 0$, the current itemset *P* to the empty set (Ø) and priority queue *Q*. In the case TWU(Y) = *lu(P, z).*

Line #2: Scan taxonomy τ and database *D* to calculate to compute *lu(P, z)* of each (generalized) item $z \in AI$ and store it using a utility-bin array.

Line #3: The algorithm compares the local utility of each item to construct the set of all secondary items of *P*.

Then line #4: *P* is sorted based on the total order $\prec$ of levels and the TWU .

At line #5: The algorithm performs another database scan of *D* using taxonomy τ to find all generalized items corresponding to secondary items in each transaction, discard every item i ∉ Secondary(*P*) from transactions are removed from the transaction to reduce memory usage,sort items in each transaction, delete empty transactions, and then build and store the utility-list of each generalized item of the database in a variable UtilityList;

Line #6: Performs a third scan of the database and taxonomy to compute the sub-tree utility of every item in Secondary(*P*) using Definition 16.

Based on Definition 17, line #7 constructs the primary set with respect to the itemset *P*; Primary(*P*).

With all the necessary information gathered, the DFS-based Search procedure (Algorithm 2) is executed at line #8 to traverse the search space recursively and extend the initial itemset P to find all top-k CLHUIs.

Algorithm 2: The SEARCH procedure

input: P: itemset, D_P: P-projected database, Primary(P): primary items of P, Secondary(P): secondary items of P, k: the number of patterns to find, μ: the internal threshold, Q: the top-k patterns until now.
output: Q is updated with top-k CLHUIs that are transitive extensions of P.
FOR EACH item z ∈ Primary(P) DO:

1. N = P ∪{z} , Secondary(P)' = {x ∈ Secondary(P) | x ∉ Desc(z, τ)};
2. Scan D_P to determine u(N), construct D_N, remove every item ∈ Desc(z, τ) and remove empty transactions;
3. IF u(N) > μ THEN Insert z into Q;
4. IF Size of Q > k THEN:
   Raises to the k-th largest utility value in Q;
   Remove from Q all patterns with utility less than μ;
5. Scan D_N to compute su(N,w), lu(N,w) for every item w ∈ Secondary(P)';
6. Primary(N) = {x ∈ Secondary(P)' ∧ su(N, z) ≥ μ};
7. Secondary(N) = {x ∈ Secondary(P)'∧ lu(N, z) ≥ μ};
8. SEARCH (N, D_N, Primary(N) , Secondary(N) , k, μ , Q);

END

Algorithm 2 accepts seven arguments as input: an itemset *P* to be extended, the projected database D_P of *P*, the set of primary and secondary items of *P*, the minimum utility threshold , the number of patterns to find k. The details of the depth-first search process to further explore a given itemset are described as follows.

Lines #1 to #8 form a loop to examine every single-item extension of *P* using each item $z \in Primary(P)$ base on Definition 8, that is of the form $N=P \cup \{z\}$, which is given in line #1.

Lines #2: Perform a database scan to determine the utility of *N* and construct *N*'s projected database D_N remove every item $\in Desc(z, \tau)$ and remove empty transactions.

Line #3 and #4 Check whether *N* has a utility greater than μ. If so, then *N* is added in priority queue *Q*. If size of $Q > k$ then raises to the k-th largest utility value in *Q* and removes from *Q* all patterns with utility less than μ.

In this case, D_N is scanned once more at line #5 to determine the sub-tree and local utility of each item w that could be used to extend N. This step is required to construct *Primary(N)* and *Secondary(N)* at lines #6 and #7, respectively.

At line #8, the Search algorithm is then recursively invoked to continue exploring the search space to extend *N*.

4.3. A descriptive example

This subsection offers a detailed walkthrough of how the proposed TKC-E algorithm discovers top k CLHUIs.

To begin with, the proposed TKC-E algorithm(Algorithm 1) accepts three input parameters that are a database D of Table 2 and Table 3, a taxonomy τ in Fig. 2, the user-defined number of patterns to be found k = 3 and is executed by 8 steps follows:

Step 1: Initializes $\mu = 0$, the current itemset P to the empty set (Ø) and priority queue Q.

Step 2: For every item $z \in AI$, the value of $lu(P, z)$ is determined. The $lu(P, z)$ values are given in Table 4.

Table 4. The lu (P,z) value of each z ∈ AI for P = Ø.

Item	**a**	**b**	**c**	**d**	**e**	**X**	**Y**	**Z**
lu	87	61	123	66	115	123	115	123

Step 3: Using the calculated $lu(P, z)$ values of every item, the set *Secondary(P)* is constructed. In this case, *Secondary(P) ={a,b,c,d,e,X,Y,Z}*.

Step 4: The set *Secondary(P)* is sorted in ascending order of level and then descending order of lu (P,z) values. Thus, *Secondary(P) ={X,Z,c,e,Y,d,a,b}*.

Step 5: Every transaction in D is then sorted by the order of *Secondary(P)*. The result is the database given in Table 5. All items not in *Secondary(P)* are removed from all transactions since they cannot be part of a HUI.

Table 5. The preprocessed database.

TID	**Items**
T1	(c, 1),(d, 1), (a, 1)
T2	(c, 6),(e, 2), (a, 2)
T3	(c, 1),(e, 1),(d, 6),(a, 1),(b, 2)
T4	(c, 3), (e, 1),(d, 3), (b, 4)
T5	(c, 2),(e, 1),(b, 2)
T6	(c, 6),(e, 2),(a, 2)
T7	(c, 1),(e, 1),(d, 2)

Step 6: For every secondary item z, $su(P, z)$ is computed for $P = \{Ø\}$. The results are given in Table 6.

Table 6. The calculated su(P, z) value of each item z ∈ AI for P = {Ø}.

Item	**X**	**Z**	**c**	**e**	**Y**	**d**	**a**	**b**
su	114	114	114	87	66	46	34	16

Step 7: Using the values obtained from the previous step, the set *Primary(P)* is constructed as *Primary(P) ={X,Z,c,e,Y,d,a,b}*.

Step 8: The algorithm is then recursively invoked to explore search space by execute DFS-based Search procedure(Algorithm 2) and extend the initial itemset P to find all top-k CLHUIs with the following parameters:

Search(*P = {Ø}, D, Primary(P) ={X,Z,c,e,Y,d,a,b}, Secondary(P) ={X,Z,c,e,Y,d,a,b}, μ = 0, Q = {Ø}, k = 3).*

Next, the Search procedure(Algorithm 2) accepts seven input parameters in step 8 of Algorithm 1 and is executed by 8 steps bellows:

Step 1: The item X is first processed by the Search function, 1. $N = P \cup \{X\} = \{X\}$, then each item of *Desc(N)* is removed. The result is *Secondary(P)' = {Z,e,d}*.

Step 2:Scan database D to calculate u(N) = 66, and the projected database D_N is constructed and remove every item $\in$ Desc(N, τ), which is shown in Table 7.

Table 7. Projecting database N = {X}on D..

TID	Items
T1	(d, 1)
T2	(e, 2)
T3	(e, 1),(d, 6)
T4	(e, 1),(d, 3)
T5	(e, 1)
T6	(e, 2)
T7	(e, 1),(d, 2)

Step 3: Since *u(N) > μ*, so *{X} is* inserted into *Q*.

Step 4: Because size of *Q < k* , then nothing change in this step.

Step 5: Scan database D_N to calculate *su(N,w), lu(N,w)* for every item $w \in Secondary(P)'$. The result is given in Table 8.

Table 8. The calculated values of su(N,w), lu(N,w) for N = {X}.

Item	Z	e	d
lu	114	106	61
su	114	106	52

Step 6 and step 7 : Given that *μ = 0,* the constructed sets *Primary(P)* and *Secondary(P)* with respect to *N* are *Primary(P) ={Z,e,d} and Secondary(P) = {Z,e,d}* respectively.

Step 8: the Search function is recursively invoked to discover CLHUIs using the following parameters: Search(*P = {X}, D_N, Primary(P) ={Z,e,d}, Secondary(P) ={Z,e,d}, $\mu = 0$, Q = {X}, k = 3).*

- With *N ={X,Z}*, it is found that *u(N) =114* > μ .Thus, this itemset is a CLHUI and inserted to *Q*. Moreover, *Q = {{X}, {X,Z}}* and D_N is empty.
- With N *={X,e}*, it is found that u(N) = 84 > μ .Thus, this itemset is a CLHUI and inserted to *Q*. In this case, T_1, T_2, T_5 and T_6 are removed since they are now empty transactions. The result returned is shown in Table 9 and Table 10. Note that, *Q = {{X}, {X,Z}, {X,e}}* and the Search function continue with *N ={X,e,d}*.

Table 9. Projecting database N = {X,e}on D_N.

TID	Items
T3	(d, 6)
T4	(d, 3)
T7	(d, 2)

Table 10. The calculated values of su(N,w), lu(N,w) for N = {X,e}.

Item	d
lu	53
su	53

- With *N ={X,e,d}*, it is found that *u(N) = 53* > μ. Thus, this itemset is a CLHUI and inserted to *Q*. In this case, size of *Q > k* so μ raises to the k-th largest utility value in Q ($\mu = 66$) and remove from Q all patterns with utility less than μ. Moreover, *Q = {{X}, {X,Z}, {X,e}}* and D_N is empty.
- With *N ={X,d}*, it is found that *su(N) =52* < μ. Thus, this itemset is not a CLHUI. At this step, the TKC-E algorithm has now finished processing item X and the next item in the set *Secondary(P)* is then considered.

Finally, when the TKC-E algorithm has now finished processing all item, the result of top-k CLHUIs is given in Table 11.

Table 11. All found top-k CLHUIs.

CLHUIs	Utility
{X, Z}	114
{Z, Y, c}	106
{Z, Y }	87

5. Experiment and Results

5.1. Data Collection

Various categories of actual data sets were utilized in the experiments. We have 2 definitions that need to be clarified: Sparse database and Dense database. These are the two types of databases that will be focused on comparing and analyzing the effectiveness of the algorithm. Sparse databases refer to databases that are designed to handle sparse data, which is data that has many missing or empty values. In a sparse database, only non-empty values are stored, which can help to reduce storage requirements and improve query performance. Sparse databases are commonly used in applications where data is sparsely populated, such as in scientific research, financial data analysis, and social network analysis. They can also be useful for handling large datasets where the majority of the data is empty or missing.

Dense database is a database designed to handle highly dense data, meaning that all values are stored and not missing or empty. The applications of dense databases are often in places that require detailed and complete data storage, such as warehouse management systems, retail systems, and financial applications. A dense database can consume more storage space than a sparse database, which may affect query performance. However, if designed and optimized properly, a dense database can provide high query performance and reliability. To the thesis's databases, Liquor and Fruithut are sparse databases. It contains transactions from US liquor stores and grocery stores. The next two databases are Chess and Accident. In comparison to the previous database, all of them exhibit a higher level of data density. Table 12 shows the detailed characteristics of datasets we used in the experiments.

Table 12. Database characteristics

Database	|D|	|I|	|GI|	MaxLevel	$\|T_{MAX}\|$	$\|T_{AVG}\|$	Density
Liquor	9284	4026	78	7	11	7.87	Sparse
Fruithut	181.970	1.265	43	4	36	3.58	Sparse
Chess	3.196	75	30	3	37	37.00	Dense
Accident	10.000	468	216	6	51	33.80	Dense

The Table 12 includes various parameters such as transaction count of D represented by |D|, number of distinct items represented by |I|, generalized item count represented by |GI|, maximum traction length represented by $|T_{MAX}|$, average transaction length represented by $|T_{AVG}|$ maximum level in each database represented by MaxLevel and the density of each database represented in the last column.

Important to notice that: the $|T_{MAX}|$ and $|T_{AVG}|$ values in the table represent the maximum and average depths of the databases, respectively. These values can have an impact on the density of the database, but they are not the primary factors that determine whether a database is dense or sparse. A database with a high $|T_{MAX}|$ value indicates that it has a high maximum depth, which means that it has a large number of levels or hierarchies within the data. This can increase the complexity of the database and make it more difficult to analyze, but it does not necessarily make the database dense or sparse. Similarly, a high $|T_{AVG}|$ value indicates that the database has a high average depth, which means that it is structured in a complex way. Again, this can make the database more difficult to analyze, but it does not necessarily make it dense or sparse. In contrast, the density of a database is primarily determined by the amount of data stored within a given physical space. A database is considered dense if it contains a high amount of data within a small physical space, regardless of its depth or complexity

A glance at the above-mentioned table reveals the all-detailed characteristics of 4 databases, each of database will have 7 attributes. Overall, what stands out from the table is that Fruithut is the largest of all database, containing almost 182 K transactions and 1.2 K number of distinct items. In terms of size, the databases vary significantly. The Chess database is the smallest, with just 3,196 transactions and 75 items, while the Fruithut database is considerably larger, with almost 182,000 transactions and 1,265 items. The Liquor and Accident databases are relatively small, with 9,284 transactions and 4026 items, and 10,000 transactions and 468 items, respectively. This suggests that each database may have different use cases, depending on the size and complexity of the data it handles. Regarding structure, the databases differ in the number of generalized items (GI) and maximum number of abstraction levels (MaxLevel). The Fruithut database has the highest number of generalized items (43), indicating a high level of abstraction and grouping of items. On the other hand, the Liquor database has the lowest number of generalized items (78), indicating a less complex structure. However, Liquor has the highest maximum number of abstraction levels (7), suggesting that its structure is more complex than the other databases. The Chess database has the lowest maximum number of abstraction levels (3), indicating that it has a simpler structure compared to the other databases. Finally, the density of the databases can be compared. Liquor and Fruithut are classified as sparse, which means they have a lower concentration of data. Sparse databases typically have many transactions with fewer items per transaction. In contrast, Chess and Accident are considered dense databases, meaning they have a higher concentration of data. Dense databases typically have fewer transactions with more items per transaction. This suggests that the processing and analysis techniques used for sparse and dense databases may differ, depending on their unique

characteristics. In conclusion, the given table provides a comprehensive comparison of four different databases based on their size, structure, and density. Each database has unique properties that can provide valuable insights into its intended use case and suitability for the TKC and TKC-E algorithms.

5.2. Experiments

In the section, we evaluate the performance of the proposed algorithm. We do the experiments on a computer with an Intel Core-i7^{TM} processor clocked at 4.5GHz and 16 GB of RAM, running on the Windows 11 operating system. We compared the performance of the algorithm in terms of execution time and peak memory usage for mining our proposed algorithm and TKC algorithm. The compared algorithms were implemented using the Java programming language with version JDK 11, the runtime and memory usage were measured using the Java API.

5.3. Result and analysis

First, we evaluated how the runtime and memory of each algorithm is influenced by the K threshold on the test datasets for various K threshold values. The results show that TKC-E has better performance than TKC for almost K threshold values. The Figure 3 is represented for two first datasets are Liquor and Fruithut, these are sparse databases. That means the maximum transaction length and average transaction length is low, Liquor only has several 11 for $|T_{MAX}|$ and 2.7 for $|T_{AVG}|$, Fruithut also has 36, 3.58 respectively. The K value for sparse datasets will be measured on a 50-point scale, where the minimum value is 50 and the largest is 1000. To begin testing the algorithms, they were first run on the test databases using a low value for the K threshold. The threshold was then gradually increased until one of the following conditions occurred: the test algorithms took too much time to execute, encountered an error due to insufficient memory, or one of the algorithms clearly outperformed the others.

The execution times of the tested algorithms are presented in Fig. 3b and Fig. 3d for Liquor, Fruithut database. The results indicate that TKC-E outperformed TKC significantly for almost values of the K threshold. When it comes to first K threshold values, such as 50 and 100, the efficiency of the two algorithms is almost the same. With the Liquor database, both algorithms use approximately 20 seconds to process the datasets. When the K value = 50, TKC terminated in 17seconds and TKC-E completed its work in 19 seconds. And the outcome is equivalent when reaching the threshold K = 100. Along with Fruithut database, both algorithms showed similar levels of performance and there was no clear advantage of one over the other.

The main difference between the two algorithms is evident in the next K values, the reasoning behind this is evidently rooted in the definitions of sub-tree utility and local utility. There are two upper bound measures used to calculate the utility of items in a database. The first measure is the local utility bound, which is equivalent to the upper bound of utility for an item in the HUP (High Utility Itemset Mining using Pattern-growth) algorithm. The second measure is the sub-tree utility upper bound, which is calculated using the remaining utility upper bound and the upper bound of fixed pattern extension (uBfpe) in the HUI-Miner algorithm. As for example, on the Liquor database (Fig. 3b), TKC-E finished in 20s and TKC required 25s. With K =500, TKC-E took considerably less time to complete processing on this database compared to TKC, specifically 2 times less time. Especially at the latest K value, the performance of TKC-E algorithm was significantly better than that of TKC on the Fruithut database, showcasing a substantial increase in performance, surpassing the previous level by a significant margin. This suggests that TKC-E is much more efficient than TKC on runtime when processing the Fruithut database.

Similar results were observed on the Fruithut database, which is a large and sparse database with over 200,000 transactions. The performance of TKC-E was significantly better than that of TKC. In terms of speed, TKC-E demonstrated a similar level of improvement as on Liquor, with a performance gain of 1:2 to 1:8 times better. Despite the Fruithut database being large and sparse, with short transactions and three abstraction levels and nearly 1.3K generalized items, the algorithm was still able to handle it efficiently.

To assess the impact of the K threshold on memory usage, the algorithms' peak memory consumption was recorded and presented in Figure 3a and Figure 3c. In general, for both of the first two databases, TKC-E consistently demonstrated lower memory usage than TKC due to its ability to narrow down the search space more effectively using the sub-tree utility. This is similar to the runtime evaluations on the Liquor database shown in Figure 3a, where TKC-E had much lower memory usage than TKC, thanks to the effective use of upper bounds for pruning the search space. At K=200, TKC-E reduced memory consumption by 25% compared to TKC, and at K=500, the reduction was 52%. At the final K value, TKC-E reduced memory consumption by 70% compared to TKC.

On the large and sparse Fruithut database, TKC-E still demonstrated better memory consumption compared to TKC. During the initial K values, TKC-E showed only a slight improvement over TKC, with a performance gain of around 25-45%. However, the most significant difference was observed at K=1000, where TKC-E used less than half of the memory utilized by TKC. This indicates that TKC-E is more memory-efficient than TKC, particularly

when handling larger values of K. On average TKC-E consumed 30% less memory than TKC across all K thresholds. It is concluded that TKC-E is an efficient algorithm because it had a better performance compared with TKC on both runtime and memory usage.

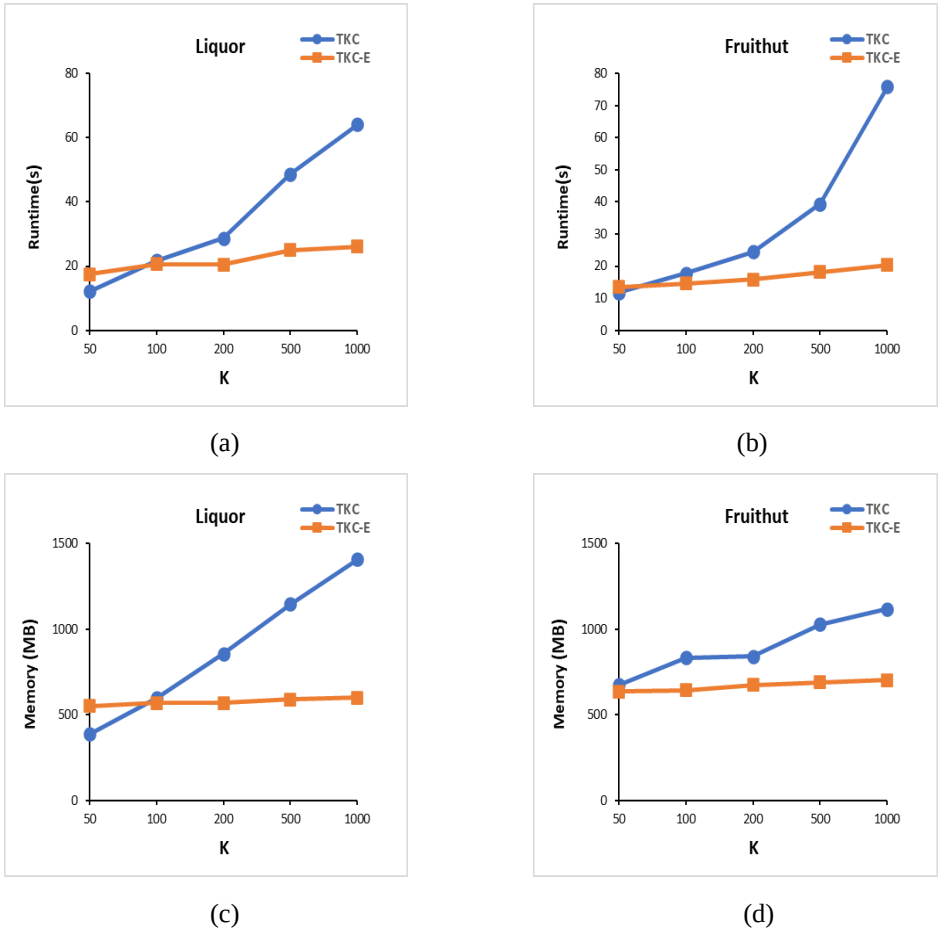

Fig. 3. Runtime, memory usage comparison in sparse datasets.

Next, we show the performance on the Chess and Accident database. They all have high density compared to the previous databases. The results are shown in Figure 4. The "Chess" and "Accident" databases are considered dense because they have a high amount of information or data stored within a relatively small physical space. In the case of the "Chess" database, it has a small size of 3.196 units, but it contains 75 individual items and a maximum level of 37. This indicates that the database is densely populated with data, with each item containing a lot of information. Similarly, the "Accident" database has a size of 10,000 units, but it contains 468 individual items and a maximum level of 51. This indicates that the database is densely populated with data, with each item containing a significant amount of information.

The TKC-E algorithm demonstrated its effectiveness on the Chess database, which is a dense database, by utilizing its tight sub-tree utility pruning technique and achieving an extremely low runtime compared to TKC. At K=10 and beyond, TKC's execution time exceeded 200 seconds, so no further testing was conducted as TKC-E only took between 12.37 to 20.46 seconds at the same K thresholds**.** As a result, TKC-E's runtime was outperformance than TKC. The same trend was observed in the Accident database, where TKC-E was faster than TKC by 2 to 3 times at different K thresholds. Among all the databases, the Chess database demonstrated the highest increase in speed for FEACP. TKC-E finished the work in 6.34 to 20.46 seconds at all tested K thresholds, while TKC took 383 to 1,218 seconds to finish the same task within the same K range. Based on the results, TKC-E was found to be 12 to 60 times faster than TKC, highlighting the efficiency of TKC-E's pruning strategies, which can eliminate many candidates found in dense databases and significantly reduce mining time. Overall, based on this evaluation, the TKC-E algorithm demonstrated the highest mining performance on dense databases.

In terms of memory usage, there was a slight difference between TKC-E and TKC. In both the Chess and Accident datasets, TKC-E was found to use slightly more memory compared to TKC. In the case of the Chess dataset (Figure 4a), which was the smallest dataset in the evaluation, TKC-E's memory usage remained as high as 15% of TKC's usage at high K thresholds, and up to 25% lower at K=50. Therefore, TKC-E used more memory than TKC at every K value, but the difference was not significant. The amount of memory used by both algorithms increased proportionally at a ratio of 1.5. The memory usage of TKC-E was deemed acceptable to compensate for the increase in runtime. In the case of the Accident dataset (Figure 4c), TKC-E also had higher memory usage compared to TKC on average. Memory sacrifice was observed in the Accident dataset, with TKC-E using almost 3 times as much memory as TKC at K=20. The difference in memory usage between the two algorithms appeared at the first K level, with TKC-E's usage being more than 2 times that of TKC, and the difference continued to increase at subsequent K levels.

Although at the first K threshold, the TKC-E algorithm is still limited in that it uses a lot of memory, the trade-off with it is the processing time of the algorithm. The reason is that it applies efficient pruning strategies managed to keep the average memory usage low by eliminating unpromising candidates from the search space when mining cross-level itemset.

Overall, the test results have shown that the TKC-E algorithm we developed is time efficient and also optimal in terms of memory usage. TKC-E had achieved a substantial speed boost and has more effective memory consumption. The reason is that it applies efficient

pruning strategies managed to keep the average memory usage low by eliminating unpromising candidates from the search space when mining cross-level itemset.

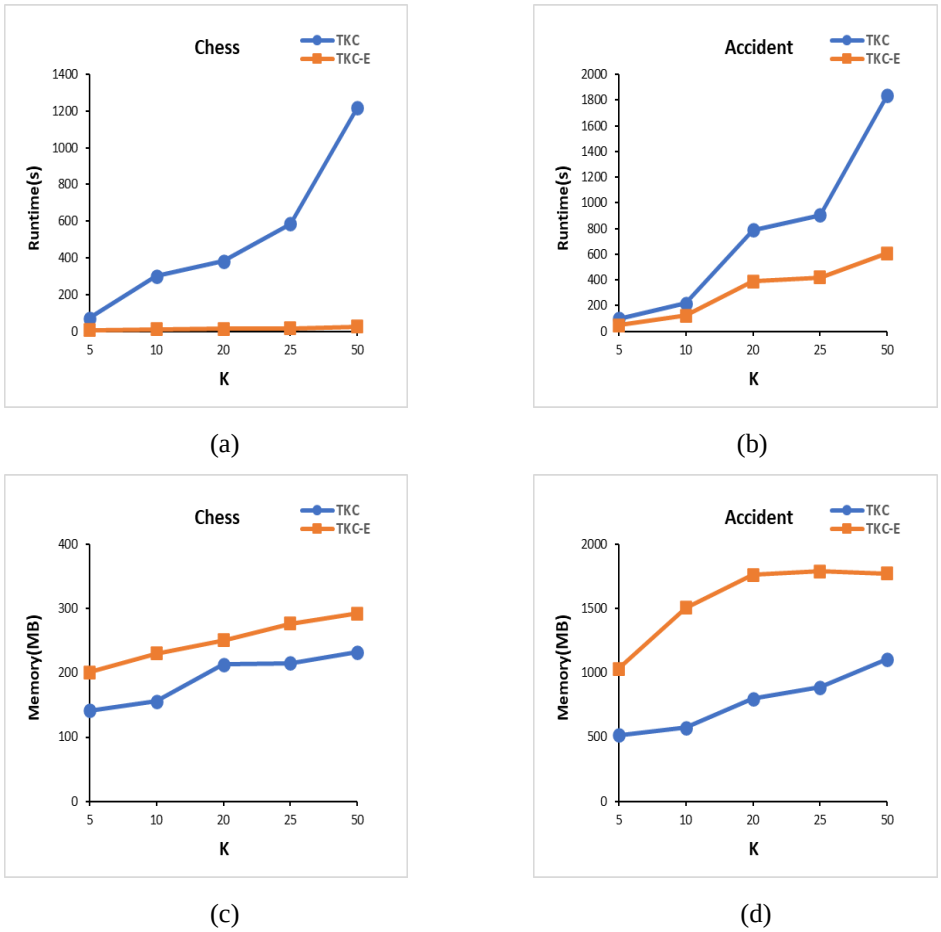

Fig. 4. Runtime, memory usage comparison in dense datasets.

6. Conclusion

In this work, a novel algorithm called TKC-E was presented to efficiently discover cross-level high-utility itemsets in transaction databases. The algorithm takes as input a transaction database with an item taxonomy that describes categories and subcategories of items. TKC-E uses the method of keeping items in a priority queue of the TKC algorithm and the min utility threshold strategy to quickly eliminate patterns and narrow the search space. Additionally, we combined a new pruning strategy from FEACP that uses local utility and sub-tree utility techniques to create upper bounds for each category in the classification system, which significantly improves the time required to search for HUIs. Experimental evaluations demonstrated that TKC-E outperforms its predecessor, the TKC algorithm, in terms of time and memory consumption. The runtime and memory consumption were improved up to 4 times on sparse datasets. However, on dense datasets, TKC-E exhibited higher from 2.0 to 3.0 times memory usage than TKC. Nevertheless, TKC-E was up to 60 times more efficient than TKC in terms of runtime on dense datasets. For future work, we will focus on improving the memory usage of TKC-E for both sparse and dense datasets. Parallel computing frameworks will be studied to reduce mining time, as well as enable computation with larger databases.

REFERENCES

1. Agrawal, R., & Srikant, R. (1994). Fast Algorithms for Mining Association Rules in Large Databases. In VLDB '94 Proceedings of the 20th International Conference on Very Large Data Bases, San Francisco (pp. 487-499).
2. Agrawal, R., Imielinski, T., & Swami, A. (1993). Mining Association Rules Between Sets of Items in Large Databases. ACM SIGMOD Record, 22(2), 207-216.
3. Yao, H., & Hamilton, H. J. (2006). Mining Itemset Utilities from Transaction Databases. Data & Knowledge Engineering, 59(3), 603-626.
4. Yao, H., Hamilton, H. J., & Butz, C. J. (2004). A Foundational Approach to Mining Itemset Utilities from Databases. Proceedings of the 2004 SIAM International Conference on Data Mining, (pp. 482-486).
5. Fournier-Viger, P., Chun-Wei Lin, J., Truong-Chi, T., & Nkambou, R. (2019). A Survey of High Utility Itemset Mining. Lecture Notes in Computer Science, 1-45.
6. Liu, Y., Wang, L., Feng, L., & Jin, B. (2020). Mining High Utility Itemsets Based on Pattern Growth without Candidate Generation. Mathematics, 9(1), 35.
7. Krishnamoorthy, S. (2018). Efficient Mining of High Utility Itemsets with Multiple Minimum Utility Thresholds. Engineering Applications of Artificial Intelligence, 69, 112-126.
8. Liu, Y., Liao, W., & Choudhary, A. (2005). A Two-Phase Algorithm for Fast Discovery of High Utility Itemsets. In Lecture Notes in Computer Science (pp. 689-695).
9. Liu, J., Wang, K., & Fung, B. C. M. (2012). Direct Discovery of High Utility Itemsets without Candidate Generation. In 2012 IEEE 12th International Conference on Data Mining (pp. 703-712).
10. Liu, J., Wang, K., & Fung, B. C. M. (2016). Mining High Utility Patterns in One Phase without Generating Candidates. IEEE Transactions on Knowledge and Data Engineering, 28(5), 1245-1257.
11. Tseng, V. S., Wu, C.-W., Shie, B.-E., & 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 - KDD '10.
12. Liu, M., & Qu, J. (2012). Mining High Utility Itemsets without Candidate Generation. Proceedings of the 21st ACM International Conference on Information and Knowledge Management - CIKM '12.

13. Fournier-Viger, P., Wu, C.-W., Zida, S., & Tseng, V. S. (2014). FHM: Faster High-Utility Itemset Mining Using Estimated Utility Co-occurrence Pruning. In Foundations of Intelligent Systems (pp. 83-92).
14. Zida, S., Fournier-Viger, P., Lin, J. C.-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(2), 595-625.
15. Cagliero, L., Chiusano, S., Garza, P., & Ricupero, G. (2017). Discovering High-Utility Itemsets at Multiple Abstraction Levels. In New Trends in Databases and Information Systems (pp. 224-234).
16. Tung, N. T., Nguyen, L. T. T., Nguyen, T. D. D., & Vo, B. (2021). An Efficient Method for Mining Multi-level High Utility Itemsets. Applied Intelligence.
17. Fournier-Viger, P., Wang, Y., Lin, J. C. W., Luna, J. M., & Ventura, S. (2020). Mining Cross-Level High Utility Itemsets. In Trends in Artificial Intelligence Theory and Applications. Artificial Intelligence Practices, 33rd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2020, Kitakyushu, Japan, September 22-25, 2020, Proceedings (pp. 858-871).
18. Tung, N. T., Nguyen, L. T. T., Nguyen, T. D. D., Fournier-Viger, P., Nguyen, N. T., & Vo, B. (2021). Efficient Mining of Cross-Level High-Utility Itemsets in Taxonomy Quantitative Databases. Information Sciences, 587, 2.
19. Chan, R., Yang, Q., & Shen, Y-D. (2003). Mining High Utility Itemsets. Third IEEE International Conference on Data Mining.
20. Wu, C.-W., Shie, B.-E., Tseng, V. S., & Yu, P. S. (2012). Mining Top-K High Utility Itemsets. Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '12.
21. 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.
22. Nouioua, M., Wang, Y., Fournier-Viger, P., Lin, J. C.-W., & Wu, J. M.-T. (2020). TKC: Mining Top-K Cross-Level High Utility Itemsets. 2020 International Conference on Data Mining Workshops (ICDMW).