Knowledge and Information Systems (2024) 66:2885–2924
https://doi.org/10.1007/s10115-023-02045-8

REGULAR PAPER

Mining Top-K constrained cross-level high-utility itemsets over data streams

Meng Han[1] · Shujuan Liu[1] · Zhihui Gao[1] · Dongliang Mu[1] · Ang Li[1]

Received: 19 July 2023 / Revised: 28 September 2023 / Accepted: 7 December 2023 /
Published online: 21 January 2024

Abstract
Cross-Level High-Utility Itemsets Mining (CLHUIM) aims to discover interesting relationships between hierarchy levels by introducing the taxonomy of items. To tackle this issue of the current CLHUIM algorithms encountering a challenge in dealing with large search spaces, researchers have proposed the concept of mining Top-K cross-level high-utility itemsets(CLHUIs). However, the results obtained by these methods often contain redundant itemsets with significant differences in hierarchy levels, and a large proportion of itemsets with higher abstraction levels, making it neglect some detailed information and unable to provide information of itemsets within the specified hierarchy range. Additionally, they are unable to handle dynamic transactional data. To address the aforementioned problems, this paper proposes Top-K Constrained Cross-Level High-Utility Itemsets Mining (TKC-CLHM) algorithm to efficiently mine Top-K itemsets across different hierarchy levels over data streams. Firstly, a new hierarchical level concept is introduced to control the abstraction level of the introduced items, and Top-K itemsets are mined within a specific hierarchy range based on this concept. Secondly, a sliding window-based data structure called Sliding Window-based Utility Projection List (SUPL) is designed, which combined with transaction projection techniques to mine CLHUIs efficiently. Lastly, a Batch and Utility Hash Table (BUHT) structure capable of storing batch and (generalized) item utility information is proposed, along with a new threshold raising strategy. Extensive experiments on six datasets with taxonomy information demonstrated that the proposed algorithm exhibited significant improvements in runtime and scalability performance compared to the state-of-the-art algorithms.

Keywords High-utility itemsets · Cross-level itemsets · Top-K high-utility itemsets · Taxonomy · Slide window

✉ Meng Han
2003051@nmu.edu.cn

[1] School of Computer Science and Engineering, North Minzu University, Yinchuan 750021, China

1 Introduction

The task of itemsets mining is to discover valuable information within a large dataset, aiding users in understanding the data and making informed decisions. Frequent Itemsets Mining (FIM) is a popular subfield in this domain, which can identify correlations between transactions and itemsets and serve as the foundation for Association Rule Mining (ARM). The objective of FIM is to mine itemsets that frequently occur in transaction databases. However, this method overlooks the relative importance and quantity of each item, making it challenging to accurately assess the user's level of interest in particular items. To address this issue, researchers have introduced the task of High-Utility Itemsets Mining (HUIM), which defines internal and external utilities of items to discover itemsets that possess relative importance in transaction datasets, known as High-Utility Itemsets (HUIs). In recent years, HUIM has found applications in various domains such as cross-selling in the retail industry [1], query recommendations in search engines, and stock investment strategies [2]. The specific meanings of internal and external utilities may vary depending on the context. For instance, in market data, they can represent purchase quantities and corresponding sales profits [1], while can represent term frequencies and inverse document frequencies in the collection of documents [3]. Although HUIM addresses the issues present in FIM tasks, it disregards the hierarchical information between the data, limiting its ability to discover associations involving items only at the lowest abstraction level. Figure 1 illustrates the taxonomy of digital products within a certain store. Network devices and peripheral devices are subcategories of digital products. Routers and switches are classified as network devices, while mice and keyboards are classified as peripheral devices. Furthermore, routers and keyboards can be further subdivided based on criteria such as functionality, appearance, and brand. Due to the sparsity of the data, it is challenging to generate meaningful strong association rules between data items at the lowest abstraction level. Moreover, the association rules generated by such HUIs are overly specific, resulting in recommendations that are too detailed and do not meet the application scenarios of daily life. For example, recommending an "HP mouse" to a user who intends to purchase an "HP mechanical keyboard". Users typically expect recommendations such as suggesting a mouse when purchasing a keyboard, providing them with a more diverse set of

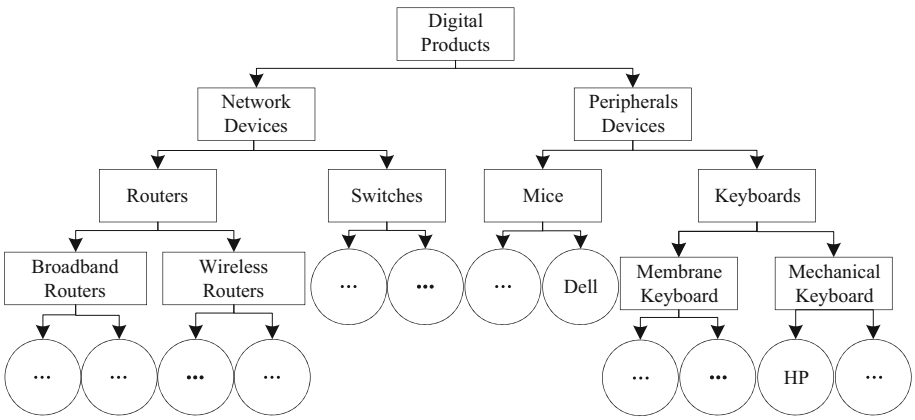


Fig. 1 Taxonomy tree of digital products

choices. However, traditional HUIM tasks fail to discover these generalized items, such as {keyboard} and {mice}, which do not exist as individual items in the transaction dataset.

In the field of FIM and ARM, researchers have introduced the concept of taxonomy-based methods to mine cross-level itemsets [4–8], which encompass items from different abstraction levels. In the HUIM field, the ML-HUI Miner algorithm [9] is proposed to mine multi-level HUIs based on the taxonomy of items. However, it can only mine within the same category and does not utilize categories of items to prune the search space. To address the issues, Fournier-Viger et al. [10] proposed an efficient CLHUIM task, which can discover itemsets with high utility at different levels of granularity. The FEACP algorithm [11] extends the concepts of local utility and subtree utility to generalized items, enabling the effective discovery of Cross-Level High-Utility Itemsets (CLHUIs) in a database. In response to the difficulty in selecting the minimum utility threshold (*minutil*), Nouioua et al. [12] proposed the TKC algorithm, which is currently considered the state-of-the-art for mining compressed patterns of CLHUIs. The algorithm mines the Top-K CLHUIs with the highest utility values without requiring the user to provide a customized threshold.

In summary, researchers have conducted extensive research in this area, but algorithms for mining such itemsets still have drawbacks and limitations. The main challenges faced by current CLHUIM algorithms are discussed as follows:

1) The mining results of CLHUIs often contain a large number of itemsets, some of which have significant hierarchical differences and higher abstraction levels. However, combinations of items between excessively high and low abstraction levels are often unable to provide decision-makers with valuable information. Moreover, the most valuable insights are typically preserved in the lower levels of the concept hierarchy. This presents challenges in interpreting and comprehending the mining results, thereby limiting their practical application in decision-making processes.
2) The process of mining CLHUIs requires consideration of relationships between multiple levels of taxonomy. As the depth of the concept hierarchy tree and the number of items in the tree increase, the search space scales exponentially, thus increasing the computational cost of the algorithm.
3) Existing algorithms for mining CLHUIs are currently static, meaning they do not take into account the temporal changes in the data during the mining process. Therefore, static algorithms may not fully leverage the temporal characteristics and dynamic information of the data, limiting their effectiveness and application scope in practical scenarios.

For Challenge 2, the TKC algorithm limits the number of mining results to a certain range by providing compact itemsets, which reduces the number of results and computational complexity. In addition, returning mining results based on the number of user-specified itemsets of interest can be more intuitive to help users discover important information in the data. However, the TKC algorithm mainly mines the Top-K CLHUIs, which are likely to consist of item combinations at higher abstraction levels. Thus, it does not address challenges 1 and 3. Furthermore, the algorithm relies on the Tax-Utility-List (TUL) structure, which incurs high time costs for maintaining and extending itemsets. In terms of the pruning strategy, the remaining utility upper bound used by the algorithm is still not tight enough, so some hopeless candidates are generated during the mining process. In terms of result sets, the TKC algorithm loses some of the CLHUIs when extending itemsets.

Therefore, this paper proposes the first algorithm for mining Top-K constrained cross-level high-utility itemsets over data streams, called TKCCLHM algorithm. The main contributions of this paper are as follows:

1) A new hierarchical concept is proposed to aid in identifying items closer to the bottom of the taxonomy tree. This concept introduces constraints on the abstraction level and search space during itemsets extension, which helps reduce computational complexity and accelerates the mining process of itemsets.
2) A sliding window-based list structure called Sliding Window-based Utility Projection List (SUPL) is introduced to store and maintain the information of itemset and batch within the window. Transaction projection techniques are utilized to generate candidates within each window, and the information of itemsets is efficiently updated when the window slides to reduce the times of database scans.
3) A novel data structure called Batch and Utility Hash Table (BUHT) is proposed to store utility information between (generalized) items, and the *minutil* raising strategy is designed based on this structure.
4) Extensive experiments are carried out to evaluate the effectiveness and efficiency of the TKCCLHM algorithm on sparse and dense real-world datasets based on taxonomy in comparison with previous state-of-the-art algorithms.

The remaining sections of this paper are organized as follows. The next section reviews related work on itemsets discovery in the context of taxonomy, Top-K high-utility itemsets mining, and high-utility itemsets mining over data streams. Section 3 presents the fundamental concepts of CLHUIM. In Sect. 4, new concepts, the SUPL structure, the BUHT structure, the mining algorithm TKCCLHM, and examples are elaborated. Section 5 discusses and analyzes the experimental results. Section 6 concludes the paper and outlines future research directions.

2 Related work

HUIM is an extension of itemset mining tasks with the primary goal of identifying itemsets that possess high profitability and importance, rather than just frequent itemsets. It places a greater emphasis on the utility of itemsets. The HUI-Miner algorithm [13] achieves efficient mining by employing a vertical data structure known as the utility-list to store information about the utility of itemsets. Through recursive scanning and connection operations on this structure, it rapidly computes the utility values of itemsets, thus facilitating an efficient mining process. However, due to the substantial computational cost associated with the numerous connection operations on utility-lists, the FHM algorithm [14] builds upon the HUI-Miner algorithm and introduces a data structure called EUCS to store transaction-weighted utility for pairs of items. This innovation allows the algorithm to find the transaction-weighted utility of any pair of 2-itemsets in constant time. Building on this foundation, the FHM algorithm proposes a Co-occurrence-based pruning strategy, significantly reducing the number of connections made to utility-lists during the mining process. The HUP-Miner algorithm [15] develops a partitioned utility-list structure and introduces two pruning strategies, which eliminate itemsets that are deemed to be unpromising in advance, further improving upon the HUI-Miner algorithm. The EFIM algorithm [16] introduces database projection and transaction merging techniques to reduce dataset scanning costs. It also designs more compact subtree utility and local utility upper bounds to prune the search space and incorporates an array-based utility counting technique for estimating high-utility upper bounds more efficiently. Subsequently, Peng et al. [17] proposed the mHUIMiner algorithm, which combines the IHUP-Tree structure to avoid generating itemsets that do not exist in the dataset, thereby significantly reducing the number of connections during expansion. HMiner algorithm [18]

introduces a novel compact utility-list structure to store repeated transaction or itemset extensions along a given search exploration path and introduces concepts of closed and non-closed utilities to compress utility values. Additionally, it proposes a virtual hyperlink structure as a novel approach to identify duplicate transactions. In addition to these fundamental algorithms in the HUIM field, this section also explores derivative tasks of HUIM, including itemsets discovery in the context of taxonomy, Top-K high-utility itemsets mining, and high-utility itemsets mining over data streams.

2.1 Itemsets discovery in the context of taxonomy

Mining the hierarchical relationships between data based on taxonomy has significant applications in various fields and domains. In the retail industry, analyzing combinations of items that span multiple hierarchical levels can reveal correlations among products, facilitating the optimization of cross-selling and promotional strategies by examining product combinations and user behavior patterns. In the financial sector, CLHUIM tasks can help predict risks and provide personalized services by analyzing multi-level data attributes of customers. In the medical field, mining the relevance of hierarchical information regarding diseases of patients can support doctors in diagnosis and decision-making processes. However, traditional FIM and HUIM algorithms do not consider categories and subcategories relationships of items, resulting in the loss of valuable insights. Several studies in FIM and HUIM have tackled the problem of hierarchical data. Srikant and Agrawal [4] were the first to introduce taxonomy into the problems of FIM and ARM to obtain cross-level itemsets. They proposed the algorithm named Cumulate to extract frequent itemsets and generate association rules, which include items from different abstraction levels. Later, Hipp et al. [5] proposed the Prutax algorithm. This algorithm is based on a vertical database representation and utilizes a most-right-depth-first search strategy to explore the most promising branches first. Additionally, two classification-based pruning strategies are designed to effectively prune the search space of frequent itemsets, leveraging the hierarchical information. The SET algorithm [6] improves on the Prutax algorithm by proposing two optimization constraints to accelerate the itemsets discovery process by restricting the search space and eliminating the generation of irrelevant itemsets. In addition to the above algorithms, other variants of mining frequent itemsets based on taxonomy have been studied, such as mining multi-level association rules [7] and mining constrained generalized rules [8].

In the field of HUIM, the ML-HUI Miner algorithm [9] introduces the concept of taxonomy to utility mining tasks for discovering more interesting information. However, the CLHUIs in the generated results are limited to the same abstraction level, ignoring the relationships between different levels and lacking the ability to effectively prune the search space based on the taxonomy of items. To address this issue, Fournier-Viger et al. [10] defined the more general CLHUIM task, aiming to discover relationships between items at different levels of granularity. They proposed the CLH-Miner algorithm, which builds upon the join-based extension approach and introduces the concept of tax-based extension. This concept expands the generalized items from the taxonomy into itemsets, while incorporating novel pruning strategies that leverage the relationships between abstraction levels to reduce the search space. To expedite the mining process, the FEACP algorithm [11] employs database projection techniques and introduces tighter upper bounds to discard unpromising itemsets at an early stage. Jiang et al. [19] devised an efficient cross-level high utility itemsets mining algorithm called DISCH, which relies on a data index structure. This algorithm rapidly identifies itemset positions using this structure and employs a reuse strategy to clear unused utility-lists

from memory, thereby improving memory utilization. Tung et al. [20] introduced a parallel approach named pCLH-Miner, which divides the search space and distributes it to different CPU cores to minimize search time and significantly enhance algorithm performance. Currently, there are three compact representations for cross-level high-utility itemsets, including closed cross-level high-utility itemsets, maximal cross-level high-utility itemsets, and Top-K cross-level high-utility itemsets. The CLH-Miner$_{closed}$ algorithm [21] defines the concept of closed cross-level itemsets to consider different levels. It incorporates the notion of closed HUIM into CLHUIM, simplifying the mining results of the CLHUIM algorithm and providing a concise, lossless representation of closed itemsets. To further reduce redundancy, the CLH-Miner$_{maximal}$ algorithm [21] is proposed to mine maximal cross-level high-utility itemsets. The recently proposed TKC algorithm [12] is a variant of the CLH-Miner algorithm, specifically designed for mining Top-K CLHUIs.

Table 1 summarizes existing cross-level high-utility itemsets mining algorithms and shows the similarities and differences between the TKCCLHM algorithm and these existing approaches. In particular, the CLH-Miner algorithm incorporates two pruning strategies based on GWU and remaining utility upper bound. It employs the TUL structure to store item utility information within transactions and item hierarchy information within the taxonomy tree. Subsequently, researchers have made improvements to the CLH-Miner algorithm, leading to the development of the pCLH-Miner algorithm, which introduces load-balancing strategies and extends the original CLH-Miner algorithm into a parallelized version, using multiple processors to concurrently handle each subspace. Additionally, when extending the CLH-Miner algorithm to tasks like compact CLHUIM tasks, these algorithms also rely on the two pruning strategies and the TUL structure. The CLH-Miner$_{closed}$ algorithm employs a CheckClosed strategy to reduce redundant information in the result set and discover closed CLHUIs. The CLH-Miner$_{maximal}$ algorithm employs CheckMaximal strategy to mine maximal CLHUIs. The TKC algorithm uses a threshold raising strategy based on the utility of items across the entire dataset to accelerate the mining of K CLHUIs. However, the TUL structure consumes significant memory space, and the list connection process is time-consuming. The DISCH algorithm introduces the DIS that utilizes memory reuse strategies to recycle and reallocate memory space occupied by utility-lists that do not meet the specified conditions. In addition to GWU and REU, the algorithm integrates pruning strategies based on the EUSC structure and the EA pruning strategy to accelerate mining. The FEACP algorithm is an extension of the EFIM algorithm and introduces local utility and subtree utility upper bounds. It employs a utility-bin array for rapid computation of upper bounds for itemsets and enhances algorithm performance step by step by generating projected datasets to reduce the size of the original dataset.

The above algorithms are all capable of handling static datasets only and cannot process unbounded data streams to discover valuable information from the latest data. Furthermore, the hierarchy definitions in existing algorithms are all top-down, without any constraints on the mining scope. When the taxonomy tree is unbalanced, they cannot mine itemset relationships at the lowest-level leaf nodes (i.e., HUIs). Additionally, mining requires considering the entire taxonomy of items, which can significantly increase the complexity of computations and results. The algorithm proposed in this paper, TKCCLHM, introduces the SUPL structure to maintain itemset utility and batch information in the data stream. It defines a bottom-up hierarchy and imposes constraints on the hierarchy. Within the specified hierarchy range, it mines the Top-K CLHUIs with the highest utility values in each window. Moreover, it introduces local utility and subtree utility upper bounds and designs the BUHT structure to store information about 2-itemsets in the window. Based on this structure, it proposes a minimum utility threshold raising strategy suitable for generalized item existence

Table 1 A comparison between Table 1 and existing cross-level high-utility itemset mining algorithms

Algorithm	Data	Itemset	Data structure	Strategy	Hierarchical	Hierarchical constrained
CLH-Miner	Static	CLHUIs	TUL	GWU, REU	Top-down	No
FEACP	Static	CLHUIs	Utility-bin array, Projection dataset	LU, SU	Top-down	No
DISCH	Static	CLHUIs	DIS	GWU, REU, EUSC, EA	Top-down	No
pCLH-Miner	Static	CLHUIs	TUL	GWU, REU, Load balancing strategy	Top-down	No
CLH-Miner$_{closed}$	Static	Closed CLHUIs	TUL	GWU, REU, CheckClosed strategy	Top-down	No
CLH-Miner$_{maximal}$	Static	Maximal CLHUIs	TUL	GWU, REU, CheckMaximal strategy	Top-down	No
TKC	Static	Top-K CLHUIs	TUL	GWU, REU, Raising the *minutil* with (generalized) items in dataset	Top-down	No
TKCCLHM	Stream	Top-K CLHUIs	SUPL, Utility-bin array, BUHT	LU, SU, Raising the *minutil* with (generalized) items in currenWin, Raising the *minutil* with 2-(generalized) itemsets in currenWin,	Down-top	No

and applies a 1-itemsets threshold raising strategy within the sliding window to accelerate the early pruning of itemset search space.

2.2 Top-K high utility itemsets mining

The Top-K HUIM task aims to find the itemsets with the K highest utility values. This task overcomes the limitations of traditional HUIM algorithms, where setting the *minutil* can be either too high or too low. It eliminates the need for lengthy threshold testing processes by directly providing the Top-K most valuable or relevant itemsets from the dataset without the requirement of setting the *minutil*. In the early stages of research, Wu et al. [22] proposed a new framework for mining Top-K HUIs and designed the TKU algorithm, which relied

on the UP-Tree structure. Subsequently, the REPT algorithm [23] builds upon this structure and designed several threshold raising strategies, including PUD, RIU, RSD, and SEP. To reduce the time and space consumption resulting from multiple scans of the database in Two-phase algorithms, Tseng et al. [24] introduced the One-phase TKO algorithm. It utilizes the utility-list structure to store the information of itemset and designed novel pruning strategies, including RUC, RUZ, and EPB, to enhance algorithm performance. Furthermore, Duong et al. [25] proposed the kHMC algorithm, which employs the EUCPT and TEP pruning strategies. To quickly increase the *minutil*, the algorithm uses the RIU, CUD, and COV strategies, significantly reducing the runtime and space consumption of the algorithm. The TKEH algorithm [26] further reduces the cost of scanning the dataset by employing database projection and transaction merging techniques. It also utilizes the utility-bin array to calculate the utilities and upper bounds of itemsets in linear time. Srikumar et al. [27] introduced a memory-efficient LIU structure and proposed the LIU-E and LIU-LB threshold raising strategies. Experimental results demonstrated the high performance of the algorithm. Researchers have also investigated several variations of Top-K high-utility mining, such as mining Top-K high-utility itemsets with negative items [28, 29], mining Top-K average high-utility itemsets [30], and mining Top-K high-utility itemsets over data streams [3].

2.3 High utility itemsets mining over data streams

The task of mining HUIs over data streams addresses the challenge of discovering HUIs in real-time and dynamic environments. Currently, mining HUIs over data streams are performed using window models, including the damped window model, sliding window model, and landmark window model. Among these models, the sliding window model is more widely applied. To handle data streams, Ahmed et al. [31] proposed the novel tree structure called HUS-Tree and the HUPMS algorithm, which stores the information of each item within the windows and mines patterns through itemsets growth. Ryang et al. [32] designed the SHU-Tree structure and introduced the RGE and RLE strategies to efficiently reduce candidates. Baek et al. [33] combined the sliding window model with the time fading model and proposed the RHUPS algorithm for mining HUIs in a time-sensitive data stream database. Dawar et al. [3] introduced a One-phase Top-K HUIM algorithm based on the utility-list. Subsequently, Jaysawal et al. [34] proposed an efficient SOHUPDS algorithm, which develops the IUDataListSW structure to record item information within the window and extend the upper bound, enabling fast projection of items based on positional indexing. Cheng et al. [35] developed a data structure called CIUDataListSW, which bases on the sliding window model to rapidly mine Top-K HUIs over data streams. In addition, several studies have investigated derivative algorithms for mining HUIs over data streams. Yun et al. [36] pioneered the application of average utility factor and data stream decay concepts to handle transactions over sensitive data streams. They proposed the MPM algorithm based on the damped window, which maintains the damped average utility of transactions by designing the Damped Average utility Tree (DAT). The DAT structure is used to manage transaction utility information and identify invalid transactions. The next step of candidate verification is determined using the Transaction Utility List, which aids in the evaluation process. Han et al. [1] first introduced the HND algorithm, which mines HUIs with negative items over data streams. Inspired by the ULB-Miner algorithm, they developed a new List Index Structure (LIS). Unlike traditional utility-lists, this structure reduces memory consumption costs through memory reuse strategies, significantly improving mining performance.

3 Preliminaries

The database type considered in this paper is a transaction database with a taxonomy-based. In this section, we focus on the definitions and key notations related to mining CLHUIs over data streams, including the concepts of a data stream, taxonomy, and relationships taxonomy-based. Let $I = \{i_1,i_2,\ldots,i_m\}$ be a set of items. For each transaction $T_c \subseteq I$, there is a unique transaction identifier *TID*, and each item i within the transaction has corresponding internal utility $in(i,T_c)$ and external utility $ex(i)$. The data stream *DS* is a sequence of transactions that arrive in order. Each window $W_s = \{B_i, B_{i+1}, \ldots, B_n\}$ contains a fixed number of batches, and each batch $B_k = \{T_j,T_{j+1},\ldots,T_v\}$ contains a fixed number of transactions. The window size *WinSize* is n-i + 1, and the batch size *BatchSize* is v-j + 1. Figure 2 presents an example of a data stream, where *WinSize* = 2 and *BatchSize* = 3. Each transaction consists of items, and the corresponding $ex(i)$ for each item i are shown in Table 2.

Definition 1 *(Taxonomy)* [10] The taxonomy τ is denoted as a directed acyclic graph (tree) defined for the data stream. The leaf nodes of the taxonomy represent different items i in the set I; while, the internal nodes represent categories of items, referred to as generalized items gi. The generalized items represent abstract categories that aggregate all descendant leaf nodes or descendant categories into a higher-level category. The child-parent edges between two (generalized) items i and j in τ represent the "*is-a*" relationship, which is $Child(j,\tau) = i$, indicating that item i is a child of item j. The taxonomy τ corresponding to the items in Fig. 2 is shown in Fig. 3.

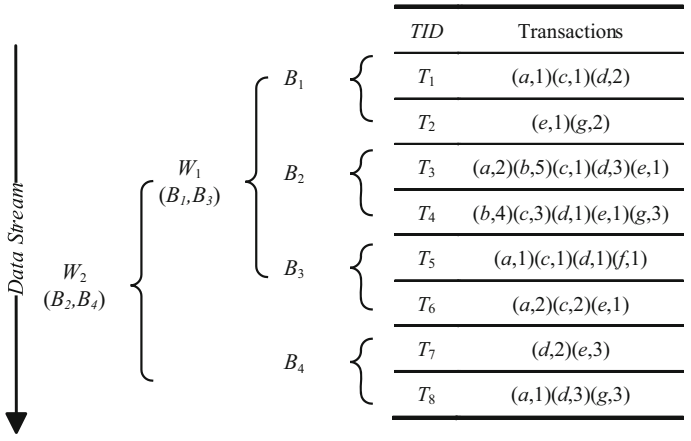
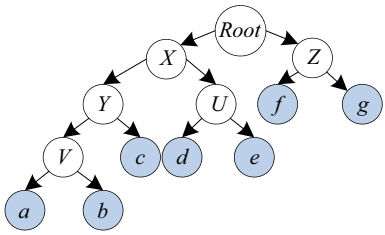


Fig. 2 Example of the data stream

Table 2 Table of external utility

Item	*a*	*b*	*c*	*d*	*e*	*f*	*g*
Utility	5	2	1	2	3	6	10

Fig. 3 Taxonomy of items



Definition 2 *(The sets of generalized items and non-generalized items)* [10] The set of all generalized items is denoted as GI, the set of non-generalized items is denoted as I, and the set of all items is denoted as $AI = GI \cup I$.

Definition 3 *(Level)* [10] The level of item p represents the number of edges traversed from the root node to reach the item.

Example 1 $Level(Root) = -1$, $Level(a) = 4$.

Definition 4 *(Relationship)*[10] If there is a path between a generalized item gi and a leaf item i, this relationship LR is defined as $(gi, i) \in LR$, where $LR \subseteq GI \times I$. If there is a path between (generalized) items p and q, this relationship GR is defined as $(p, q) \in GR$, where $GR \subseteq AI \times AI$. The set of leaf items reached through gi is defined as $Leaf(gi,\tau) = \{i | (gi,i) \in LR\}$. If p is a generalized item, its set of descendants is defined as $Desc(p,\tau) = \{q | (p,q) \in GR \wedge Level(p) > Level(q)\}$.

Example 2 For the τ showed in Fig. 3, $(Y,c) \in LR$, $(Y,V) \in GR$, $Leaf(Y,\tau) = \{a,b,c\}$, $Desc(Y,\tau) = \{V,a,b,c\}$.

Definition 5 *(Utility of an item/itemset)* [3] The utility of item i in window W_s is defined as $u(i, W_s) = \sum_{T_c \in g(i)} u(i, T_c) = \sum_{T_c \in g(i)} in(i, T_c) \times ex(i)$. The utility of itemset X in window W_s is defined as $u(X, W_s) = \sum_{T_c \in g(X)} u(X, T_c) = \sum_{T_c \in g(X) \wedge i \in X} in(i, T_c) \times ex(i)$, Where $g(X)$ represents the set of transactions in window W_s that contain itemset X. The utility of an item/itemset in batch B_k is obtained by replacing W_s with B_k in $g(X)$.

Example 3

$$u(a, W_1) = u(a, T_1) + u(a, T_3) + u(a, T_5) + u(a, T_6) = 1 \times 5 + 2 \times 5 + 1 \times 5 + 2 \times 5 = 30,$$
$$u(\{a, e\}, W_1) = u(\{a, e\}, T_3 + u(\{a, e\}, T_6) = u(a, T_3) + u(e, T_3) + u(a, T_6) + u(e, T_6)$$
$$= 2 \times 5 + 1 \times 3 + 2 \times 5 + 1 \times 3 = 26.$$

Definition 6 *(Utility of a generalized item/itemset)* The utility of the generalized item gi within window W_s is defined as $u(gi, W_s) = \sum_{T_c \in g(i) \wedge i \in Leaf(gi,\tau)} in(i, T_c) \times ex(i)$. The utility of the generalized itemset GX within transaction T_c is defined as $u(GX, T_c) = \sum_{p \in GX} u(p, T_c)$, and the utility of GX in window W_s is represented as $u(GX, W_s) = \sum_{T_c \in g(GX)} u(GX, T_c)$, where $g(GX) = \{T_c \in W_s | \exists X \subseteq T_c \wedge X \subseteq desc(GX, \tau)\}$. The utility of an item/itemset in batch B_k is obtained by replacing W_s with B_k in $g(X)$.

Example 4

$$u(Z, W_1) = u(g, B_1) + u(g, B_3) + u(f, B_3)$$

Table 3 GWU value of the 1- itemsets in W_1

Item	Z	X	f	g	Y	U	d	e	c	V	a	b
GWU	83	138	14	69	115	138	100	114	115	115	69	76

$= u(g, T_2) + u(g, T_4) + u(f, T_5) = 2 \times 10 + 3 \times 10 + 1 \times 6 = 56,$
$u(\{a, Z\}, W_1) = u(\{a, f\}, B_3) = u(\{a, f\}, T_5) = u(a, T_5) + u(f, T_5) = 1 \times 5 + 1 \times 6 = 11.$

In cross-level highutility itemsets mining algorithms, the sorting of items is determined by considering the hierarchical relationships among items, along with a generalized form of the GWU metric, which extends the TWU metric traditionally used in HUIM [10]. The following introduces a generalized form of the GWU metric within the context of a data stream environment.

Definition 7 *(Total order of items)* Let there be a set $AI^* \subseteq AI$, AI^* represents the set of items in the sliding window W_s. That total order $\prec$ is defined such that $a \prec b$ for two items $a,b \in AI^*$, if $Level(a) = Level(b) \wedge GWU(a,W_s) < GWU(b,W_s)$ or $Level(a) < Level(b)$.

Definition 8 *(The GWU measure)* [10] The transaction utility (TU) of T_c in window W_s is defined $TU(T_c) = \sum_{i \in T_c} in(i, T_c) \times ex(i)$. For an itemset $X \subseteq I$, the Generalized Weighted Utility (GWU) of X in window W_s is defined $GWU(X, W_s) = \sum_{T_c \in g(X)} TU(T_c)$. For an itemset $X \subseteq GI$, the GWU of X in window W_s is defined $GWU(X, W_s) = \sum_{T_c \in g(i) \wedge i \in Leaf(X,\tau)} TU(T_c)$, Where $g(X)$ represents the set of transactions in window W_s that contain itemset X.

Example 5 $GWU(g,W_1) = TU(T_2) + TU(T_4) = u(\{e,g\},T_2) + u(\{b,c,d,e,g\},T_4) = 23 + 46 = 69$, $GWU(Z,W_1) = TU(T_2) + TU(T_4) + TU(T_5) = 69 + u(\{a,c,d,f\},T_5) = 69 + 14 = 83$. The order of items is $Z \prec g$ because $Level(Z) < Level(g)$. Based on the GWU values of items in window W_1, the total order is $Z \prec X \prec f \prec g \prec Y \prec U \prec d \prec e \prec c \prec V \prec a \prec b$, as shown in Table 3.

The TKCCLHM algorithm conducts a depth-first search starting from the root (empty set) to explore the search space of itemsets. This search is based on the order of itemsets defined within the window. During this depth-first search, for any itemset α, items are processed in the order specified within the window. Recursively, at each step, an item z is appended to α to generate larger itemsets. Next, this paper will introduce definitions related to the depth-first search exploration of itemsets.

Definition 9 *(Extension in a sliding window)* Let AI^* be a set of items in window W_s, $\prec$ is a total order of AI^*, and α is an itemset. The set of all items in sliding window W_s that are used to expand item α based on depth-first search is defined as $E_{W_s}(\alpha) = \{y | y \in AI^* \wedge x \prec y, \forall x \in \alpha, y \notin Desc(x, \tau)\}$.

Example 6 As shown in Fig. 3, the descendants of item U are d and e. The total order of items in the window W_1 is illustrated in Example 5. Therefore, based on the total order of items, extract the items located after item U and remove the descendants of U. $E_{W_1}(U) = \{c, V, a, b\}$.

In order to efficiently reduce the search space, remaining utility pruning has been applied in many algorithms [10, 12, 35]. The algorithm presented in this paper employs utility upper

bounds that are more compact than remaining utility upper bounds, including local utility and subtree utility, which were introduced in state-of-the-art algorithms [11, 35]. This paper extends these concepts to the data stream environment, introducing the concept of windows.

Definition 10 *(The upper bound of the remaining utility in a sliding window)* [35] Let α be an itemset, the remaining utility of itemset α in transaction T_i is defined as $re(\alpha, T_i) = \sum_{i \in E_{W_s}(\alpha) \wedge \alpha \in T_i} u(i, T_i)$. The remaining utility of itemset α in window W_s is defined as $re(\alpha, W_s) = \sum_{i \in E_{W_s}(\alpha) \wedge \alpha \in T_i \wedge T_i \in W_s} u(i, T_i)$. The upper bound of the remaining utility of itemset α in window W_s is defined $reu(\alpha,W_s) = u(\alpha,W_s) + re(\alpha,W_s)$.

Definition 11 *(Local utility and Subtree utility in a sliding window)* Let α be an itemset and item $z \in E_{W_s}(\alpha)$. If the item $z \in I$, the local utility and subtree utility of z [35] regarding as α in window W_s are defined $lu_{W_s}(\alpha, z) = \sum_{T_i \in \varphi(\alpha \cup z)} u(\alpha, T_i) + re(\alpha, T_i)$ and $su_{W_s}(\alpha, z) = \sum_{T_i \in \varphi(\alpha \cup z)} (u(\alpha, T_i) + u(z, T_i) + \sum_{i \in E_{W_s}(\alpha \cup z) \wedge i \in T_i} u(i, T_i))$ If the item $z \in GI$, the local utility and subtree utility of z regarding as α in window W_s are defined $lu_{W_s}(\alpha, z) = \sum_{T_i \in \varphi(\alpha \cup i), \exists i \in Leaf(z,\tau)} u(\alpha, T_i) + re(\alpha, T_i), su_{W_s}(\alpha, z) = \sum_{T_i \in \varphi(\alpha \cup i), \exists i \in Leaf(z,\tau)} (u(\alpha, T_i) + u(z, T_i) + \sum_{i \in E_{W_s}(\alpha \cup z) \wedge i \in T_i} u(i, T_i))$, where $\varphi(x)$ is represented a set of transactions including itemset x in window W_s.

Example 7 $lu_{W_1}(\varnothing, f) = \sum_{T_i \in \varphi(f)} u(\varnothing, T_i) + re(\varnothing, T_i) = u(\varnothing, T_5) + re(\varnothing, T_5) = GWU(T_5) = 14$, $lu_{W_1}(\varnothing, V) = TU(T_1) + TU(T_3) + TU(T_4) + TU(T_5) + TU(T_6) = 10 + 30 + 46 + 14 + 15 = 115$, Based on the total order of items within the W_1 as shown in Example 5, $E_{W_1}(f) = \{g, d, e, c, a, b\}$, $E_{W_1}(V) = \varnothing$, $su_{W_1}(\varnothing, f) = u(f, T_5) + u(d, T_5) + u(c, T_5) + u(a, T_5) = 14$, $su_{W_1}(\varnothing, V) = u(V, W_1) = u(a, W_1) + u(b, W_1) = 30 + 18 = 48$.

Based on local utility and subtree utility, the following definitions provide the concepts of primary itemsets and secondary itemsets. These items need to be considered when exploring the search space.

Definition 12 *(Primary items and Secondary items)* [35] Let α be an itemset in the window W_s, the primary items and secondary items of α are defined $Primary_{W_s}(\alpha) = \{z | z \in E_{W_s}(\alpha) \wedge su_{W_s}(\alpha, z) \geq minutil\}$ and $Secondary_{W_s}(\alpha) = \{z | z \in E_{W_s}(\alpha) \wedge lu_{W_s}(\alpha, z) \geq minutil\}$. They are follows $Secondary_{W_S}(\alpha) \subseteq Primary_{W_S}(\alpha)$.

The TKCCLHM algorithm calculates the utility of itemsets and their upper bounds through window scanning. To reduce the cost of scanning, when considering an itemset α during the depth-first search, while scanning the database to calculate the utility of itemsets within the subtree of α or their upper bounds, all items not belonging to $E_{W_s}(\alpha)$ can be ignored.

Definition 13 *(Projected window)* [35] The set of items organized by projecting a transaction T_c over an itemset α is defined as $\alpha - T_c = \{i | i \in T_c \wedge i \in E(\alpha)\}$. The several sets obtained after projecting itemset α over all transactions in a window W_s are defined as $\alpha - W_s = \{\alpha - T_c | T_c \in W_s \wedge \alpha - T_c \neq \varnothing\}$.

Example 8 According to the total order of items in the window W_1, $\{c\} - T_3 = \{a,b\}$,$\{U\} - T_3 = \{c,a,b\}$.

Definition 14 *(Top-K cross-level high-utility itemsets mining)* Let X be a (generalized) itemset, *minutil* and K are user-defined parameters. The itemset X is a CLHUI if $u(X,W_s) \geq minutil$. The main task of the Top-K Cross-Level High-Utility Itemsets problem is to discover a set of K CLHUIs with the highest utility values in each window.

Example 9 When $K = 5$, the Top-K CLHUIs in window W_1 are {X,Z:83}, {X:82}, {Y,U:79}, {c,V,U:79} and {V,U:71}.

4 TKCCLHM algorithm

In this section, we primarily introduced a new hierarchy concept, the proposed data structure, and the key techniques utilized by the algorithm. In a concept hierarchy tree, valuable information is typically found at lower abstraction levels because items at lower levels are more specific and granular, thus providing more detailed and specific information. Combinations of items at lower levels can better reflect associations and user behavior patterns in real-world scenarios. However, data at lower abstraction levels is generally sparse, making it difficult to generate strong association rules for effective decision-making. Therefore, the CLHUIM task incorporates generalized items into consideration, which to some extent condenses partial information and enables the discovery of meaningful relationships between abstracted levels. However, current CLHUIM algorithms tend to include all generalized items from the concept hierarchy tree in the search process, which increases search costs to some extent. Additionally, they may mine meaningless itemsets with significant level differences. To enhance the flexibility of mining, this algorithm defines a new hierarchical concept to identify items closer to the bottom of the taxonomy tree. Based on this concept, the algorithm constrains the hierarchical level of itemsets extension during the mining process.

Definition 15 *(BottomLevel)* If an item i exists, then define $BottomLevel\ (i) = 1$. If a generalized item gi, exists, then define $BottomLevel(gi) = \{x \in Child(gi) | Min\{BottomLevel(x)\} + 1\}$.

Example 10 $BottomLevel(Y) = Min\{1,2\} + 1 = 2$.

Definition 16 *(MaxBottomLevel)* For the item $i \in AI$, $MaxBottomLevel(\tau) = Max\{BottomLevel(i)\}$.

Based on the definitions of relevant hierarchy provided above, the BottomLevel is constrained to prevent the mining of itemsets that do not fall within the parameter-defined range. The following definitions outline the constraints on the hierarchy range for mining items and their extensions.

Definition 17 *(BottomLevel constraint)* If it is specified that all generated CLHUIs fall within the hierarchy range [$bottom\_k$, top_k], then for any itemset X, if $i \in X$ then $BottomLevel(i) \in [bottom\_k, top\_k]$, where $bottom\_k \geq 1$ and $top\_k \leq MaxBottomLevel(\tau)$.

Example 11 For a taxonomy τ in Fig. 3, the $MaxBottomLevel(\tau) = 3$. When $K = 5$, the CLHUIs in window W_1 under different hierarchical constraints are shown in Table 4. Because $BottomLevel(X) = 3$, if the hierarchical BottomLevel range does not include 3, the itemset X and its supersets do not need to be extended. This reduces the introduction of generalized items at higher abstraction levels to some extent. For example, if the hierarchical constrained range is [1, 2], the item X is ignored during mining. The result set does not include X and its superset. Additionally, when $bottom\_k = top\_k$, individual level information can be mined. In particular, the mining results are consistent with HUIM algorithms, when $bottom\_k = top\_k = 1$. The proposed algorithm in this paper allows customization of the hierarchical range according to specific requirements to explore the Top-K interesting association about items within a specific BottomLevel range.

Table 4 CLHUIs in W_1 under different BottomLevel constraints

BottomLevel	CLHUIs with BottomLevel constraints
[1, 3]	{*X*,*Z*:83},{*X*:82},{*Y*,*U*:79}{*c*,*V*,*U*:79},{*V*,*U*:71}
[1, 2]	{*Y*,*U*:79},{*c*,*V*,*U*:79},{*V*,*U*:71},{*U*,*Z*:66}{*Y*,*U*,*Z*:60}
[2, 3]	{*X*,*Z*:83},{*X*:82},{*Y*,*U*:79},{*V*,*U*:71},{*U*,*Z*:66}
[3, 3]	{*X*:82}
[2, 2]	{*Y*,*U*:79},{*V*,*U*:71},{*U*,*Z*:66}{*Y*,*U*,*Z*:60},{*Z*:56}
[1, 1]	{*e*,*g*:56},{*g*:50},{*b*,*c*,*d*,*e*,*g*:46},{*b*,*c*,*e*,*g*:44},{*b*,*d*,*e*,*g*:43}

4.1 SUPL structure

The paper proposes a new list structure called SUPL, as shown in Fig. 4. This list structure is designed to store details of item within the sliding window W_s, including batch information, utility information within transactions, and category information. The projection technique is used to project the set of transactions for the item within the window. As the explored itemsets within the window increase, the projected transaction set for that itemsets becomes smaller, effectively reducing the scanning cost of transactions within the window.

In the structure, "*Item*" represents the name of the item, which corresponds to a node in the taxonomy tree. Each item has information about its name, children nodes, parent node, level, and BottomLevel. "*WinIU*", "*WinLU*", and "*WinSU*" store the utility value, local utility value, and subtree utility value of the item within the entire window. "*BID*", "*BIU*", "*BLU*", and "*TransProjection*" represent the batch identifier, utility within the batch, the local utility within the batch, and the projected transaction set, respectively. "*Item-T*", "*TU*", and "*PrefixUtility*" indicate the projected transactions for the item within the current window, the transaction utility of the projected transactions, and the prefix utility of the item, which is the utility value of the item in this projected transaction. The projected transactions are similar to the pattern growth method and do not generate non-existent itemsets. Based on the projected transactions, the local utility upper bound and subtree utility upper bound of the next extension itemset can be efficiently calculated using the utility-bin array. When the

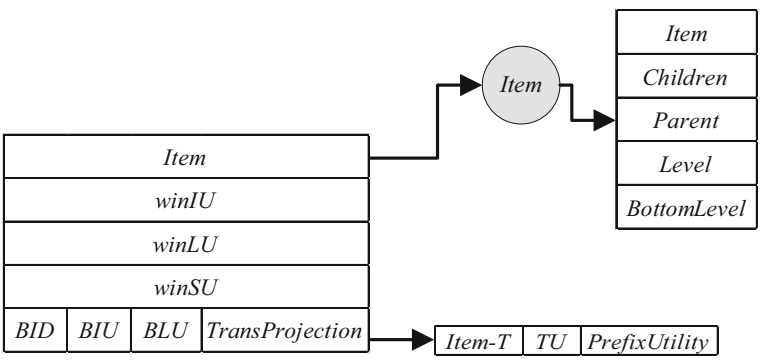


Fig. 4 SUPL structure

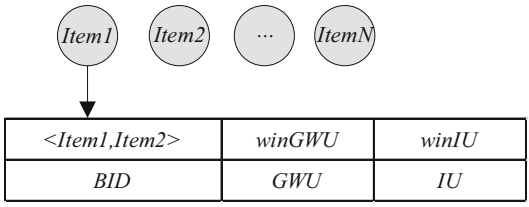


Fig. 5 BUHT structure

window slides, the SUPL structure can maintain the information of itemset within the current window by adding or removing information of batch.

4.2 Threshold raising techniques

Optimization 1 (Raising the threshold using 1- (generalized) itemsets within window W_s) The *minutil* is raised to the utility value of the K-th highest 1-(generalized) itemset within the current window.

A non-increasing itemset utility queue of length K is maintained to effectively raise the *minutil* within the current window. In previous work, during the initial phase, the *minutil* is raised to the K-th highest utility value among all generalized and non-generalized items in the entire dataset. However, this paper focuses on the mining task of CLHUIs based on sliding windows. Therefore, the threshold is raised to the K-th highest utility value among all 1-itemsets within the current window, when the window slides.

Optimization 2 (Raising the threshold using 2- (generalized) itemsets within window W_s) The *minutil* is raised to the utility value of the K-th highest 2-(generalized) itemset within the current window. It is worth noting that there are no ancestor–descendant relationships among the items in the 2-itemsets.

The paper proposes the BUHT structure to store details about 2-itemsets within the current window, as depicted in Fig. 5. The *minutil* is rapidly raised by calculating the utility values of 2-itemsets among (generalized) items within the current window. In this structure, there are no ancestor–descendant relationships among the 2-itemsets. The BUHT structure achieves this by storing the *IU* and *GWU* values of 2-itemsets within the current window and specific batches. The *IU* value of 2-itemset is placed into a non-increasing priority queue when its *GWU* value satisfies the *minutil*. A new *minutil* is derived through Optimization 2. When the window slides, it is only necessary to remove the old batches from the structure, scan the information of the new batch, and add the utility of 2-itemsets from the new batch. This effectively maintains the utilities of 2-itemsets within the entire window without scanning the entire window.

4.3 TKCCLHM algorithm

In this subsection, the workflow of the TKCCLHM algorithm for mining Top-K constrained cross-level itemsets using a sliding window is described. The main workflow of the algorithm is depicted in Fig. 6.

The main process of the TKCCLHM algorithm is shown in the pseudocode of Algorithm 1. The algorithm requires six input parameters: (1) Taxonomy τ; (2) Data stream (*DS*);

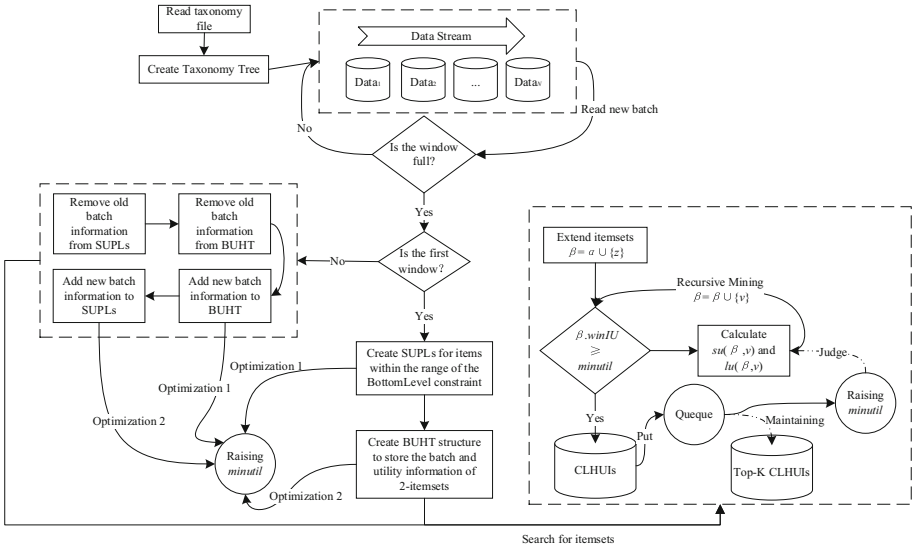


Fig. 6 Workflow of the TKCCLHM algorithm

(3) Window size (*WinSize*); (4) Batch size (*BatchSize*); (5) user-defined parameter K; (6) BottomLevel constraint range. In the end, the algorithm returns the Top-K CLHUIs for each window.

First, the algorithm reads the taxonomy file of items and creates a taxonomy tree. For the given data stream, it scans the transaction sets on the data stream and reads the transaction sets based on the specified *WinSize* and *BatchSize* (lines 1–6). As the window slides, when the number of batches within the window equals the specified *WinSize*, the algorithm processes the information within the current window (lines 12–26). When the number of batches within the window exceeds the specified *WinSize*, it is necessary to remove the old batch information from the data structures (lines 7–10). The process of handling the old batch includes removing the transaction sets from the current window, as well as removing the *IU* and *GWU* values of itemsets and other information related to the old batch from the SUPL structure and BUHT. At this point, these structures store the common batch information between the previous window and the current window. With the addition of new batch transaction data, the transaction sets within the current window are updated (line 11). The algorithm initializes the utility-bin array within the current window directly based on the common batch information stored in the SUPL structure. Using the hierarchical relationships of items in the taxonomy tree and the user-defined BottomLevel constraint, it only calculates the local utility upper bound for items within the specified BottomLevel range. Generalized items that are not within the constraint range will not be included in the transactions. If the current window is the first window, the SUPL for 1-itemsets needs to be created. Otherwise, only the item SUPL structure needs to be updated by scanning the latest batch and adding the utility information of items in the new batch to the SUPL structure (line 14). Subsequently, the threshold is raised to the K-th highest utility value among the 1-itemsets within the current window using optimization strategy 1 and the $Secondary_{W_s}(\alpha)$ is obtained (lines 16–17). The BUHT is used to store the utility information of 2-(generalized) itemsets that meet the BottomLevel requirements within the current window. If the current window is the first window, the entire window needs to be

scanned to calculate the utility information of all 2-itemsets and create the BUHT. Otherwise, only the latest batch needs to be scanned to record the *GWU* and *IU* values of the 2-itemsets within the current window in the BUHT. The utility values of the 2-itemsets that satisfy the *minutil* are added to the priority queue Q_2 (lines 18–19). According to the *GWU* values and hierarchical relationships of (generalized) items within $Secondary_{W_s}(\alpha)$, the total order of items within the current window is calculated. Then, items that do not satisfy the *minutil* and empty transactions are removed from the transactions, and the items are re-sorted according to the order (lines 20–21). Optimization strategy 2 is applied to raise the threshold to the utility value of the K-th highest utility 2-itemsets within the current window (line 22). The utility-bin array is used to continue calculating the subtree utility of the 1-itemsets within the current window and filtering out $Primary_{W_s}(\alpha)$ using the updated *minutil*. Later, Algorithm 1 initiates a recursive traversal by calling the Search method to explore the extension itemsets of the item α until all items in $Primary_{W_s}(\alpha)$ have been searched. Finally, the algorithm outputs the Top-K CLHUIs with the highest utility values within the current window. As the window slides, the TKCCLHM algorithm continues mining Top-K CLHUIs in the subsequent windows.

Algorithm 1 The TKCCLHM algorithm

Input: τ : a taxonomy, *DS*: a data stream, *WinSize*: number of batches in Window, *BatchSize*: a number of transactions in Batch, *K*: user-defined parameter, [*bottom_k*, *top_k*]: itemsets limit the BottomLevel scope
Output: Top-K cross-level high utility itemsets of BottomLevels in [*bottom_k*, *top_k*]

1 Scan τ and *DS*
2 Create taxonomyTree
3 While there is stream of transactions do
4 Initialize a priority queue Q, Q_1, Q_2, BUHT, SUPLs of item, CurrentBatch, CurrentWin, *minutil*
5 Read *DS* and add transactions to currentBatch according to *BatchSize*
6 ++currentBatch;
7 If currentBatch.Size > *WinSize* then
8 CurrentWin.removeOldBatch(OldBatchID)
9 SUPL.removeOldBatch(OldBatchID)
10 BUHT.removeOldBatch(OldBatchID)
11 Add CurrentBatch to CurrentWin according to *WinSize*
12 If currentWin.Size ≥ *WinSize* then
13 Initialize *minutil*, $\alpha = \emptyset$
14 Use a utility-bin array to calculate $u(\alpha, W_s)$ and $lu_{W_s}(\alpha, z)$ for $z \in AI^*$, build or update SUPLs of each item by scanning currenWin ,where BottomLevel(z)∈[*bottom_k*, *top_k*]
15 Put the utility values of the all 1-itemsets in current window into Q_1
16 $minutil \leftarrow max(minutil, Q_1[K])$
17 $Secondary_{W_s}(\alpha) \leftarrow \{z | z \in W_s \wedge lu_{W_s}(\alpha, z) \geq minutil\}$
18 Use the BUHT structure to store the all 2-itemsets utility information by scanning currenWin
19 Put the utility values of the 2-itemsets in current window into Q_2
20 Let $\prec$ be the total order of *GWU* and *Level* on $Secondary_{W_s}(\alpha)$
21 Scan CurrenWin to remove each item$\notin Secondary(\alpha)$, sorted items in each transaction according to $\prec$ and delete empty transactions
22 $minutil \leftarrow max(minutil, Q_2[K])$
23 Use a utility-bin array to calculate $su_{W_s}(\alpha, z)$ for $z \in AI^*$and update SUPLs
24 $Primary_{W_s}(\alpha) \leftarrow \{z | z \in Secondary_{W_s}(\alpha) \wedge su_{W_s}(\alpha, z) \geq minutil\}$
25 $Search(\alpha, \alpha - W_s, Primary_W(\alpha), Secondary_{W_s}(\alpha), Q, minutil)$
26 Output Top-K CLHUIs
27 End While

The search process of Algorithm 2 requires a total of six parameters: (1) the itemset α, which is initially empty; (2) the projection dataset $\alpha - W_s$ of the itemset α within the current window, which is generated by TransProjection in the SUPL structure of α; (3) the primary items of the itemset α, $Primary_{W_s}(\alpha)$; (4) the secondary items of the itemset α, $Secondary_{W_s}(\alpha)$; (5) the set that stores the first K CLHUIs with the largest utility values in the current window, initially empty; (6) the *minutil* within the current window.

The depth-first search process starts with the extension itemset α. First, an item z is taken from the $Primary_{W_s}(\alpha)$, and a new extended itemset β is generated by combining z with α. Since the generation of ancestor–descendant extensions is not allowed in CLHUIs, a constraint is applied to the secondary items of α, resulting in the constrained secondary items called $Secondary^*_{W_s}(\alpha)$. Subsequently, the transaction set of the extended item β is projected

where the empty transactions and the descendants of item z are removed named $\beta - W_s$, and the utility value of β in the current window is computed to determine whether the extended itemset β is a CLHUI. If the utility of β is greater than or equal to the current *minutil*, it is added to the priority queue Q to maintain the Top-K CLHUIs with the highest utility values in the current window, and the current *minutil* is updated. The program further scans $\beta - W_s$ to calculate the local utility and subtree utility of the extended 1-itemset β regarding the item $v \in Secondary^{*}_{W_s}(\alpha)$ in W_s, using the updated *minutil* to derive $Primary_{W_s}(\beta)$ and $Secondary_{W_s}(\beta)$ to determine whether the item has a need to be further extended, and then recursively calls the Search function to further extend the itemset β.

Algorithm 2 Search

Input: α: an itemset, $\alpha - W_s$:the α projected database, $z \in Primary_{W_s}(\alpha)$:the primary items of α,$Secondary_{W_s}(\alpha)$: the secondary items of α, Q: a non-increasing priority queue Q of K size, *minutil*
Output: Top-K CLHUIs in W_s

1 Foreach item $z \in Primary_{W_s}(\alpha)$ do
2 $\beta = \alpha \cup \{z\}$, $Secondary^{*}_{W_s}(\alpha) = \{y | y \in Secondary_{W_s}(\alpha) \wedge y \notin Desc(z, \tau)\}$
3 Scan $\alpha - W_s$ to calculate $u(\beta, W_s)$, create $\beta - W_s$, and delete $i \in Desc(z, \tau)$ in $\beta - W_s$
4 If $u(\beta, W_s) \geq minutil$ then put β into Q
5 $minutil \leftarrow max(minutil, Q[K])$
6 Scan $\beta - W_s$ to calculate $lu_{W_s}(\beta, v)$ and $su_{W_s}(\beta, v)$ for $v \in Secondary^{*}_{W_s}(\alpha)$ by utility-bin array
7 $Primary_{W_s}(\beta) \leftarrow \{v | v \in Secondary^{*}_{W_s}(\alpha) \wedge su_{W_s}(\beta, v) \geq minutil\}$
8 $Secondary_{W_s}(\beta) \leftarrow \{v | v \in Secondary^{*}_{W_s}(\alpha) \wedge lu_{W_s}(\beta, v) \geq minutil\}$
9 $Search(\beta, \beta - W_s, Primary_{W_s}(\beta), Secondary_{W_s}(\beta), Q, minutil)$
10 End Foreach
11 Return Top-K CLHUIs in W_s

4.4 Example analysis

This subsection gives an example of the overall flow of the TKCCLHM algorithm, assuming that the taxonomy tree has been created and the rest of the parameters take the following values: $WinSize = 3$, $BatchSize = 2$, $K = 5$, and the BottomLevel constraint range is [1, 2].

When the new batch is fully read and the window W_1 is full, the *minutil* in W_1 is initialized to 0. Based on the taxonomy tree and the BottomLevel constraint range, it is known that $BottomLevel(X) \notin [1, 2]$, so the generalized item X is not introduced. Apart from this node, the local utility values of (generalized) items in W_1 are calculated, and the SUPLs are created. Taking items Z, g, and Y as examples, their SUPL structures are shown in Fig. 7. The *GWU* values and *IU* values of all 1-itemsets in W_1 are shown in Tables 3 and 5, respectively. The utilities of all 1-itemsets are stored in queue $Q_1 = [56,56,50,48,30,18,14,12,8,6]$, and sorted to obtain $Q_1[5] = 30$. According to Optimization 1, the *minutil* is raised to 30, $Secondary_{W_1}(\varnothing) = \{a, b, c, d, e, g, V, Y, U, Z\}$. The BUHT structure is created to store all the 2-itemsets that satisfy the BottomLevel constraint condition, as shown in Fig. 8. How-

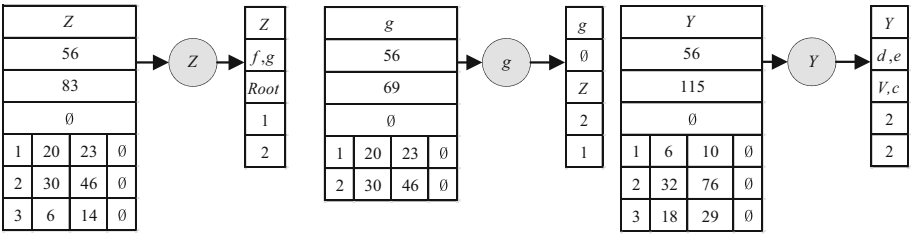


Fig. 7 SUPL structures of items Z, g and Y

Table 5 Utilities of the 1- itemsets in W_1

Item	a	b	c	d	e	f	g	V	Y	U	Z
Utility	30	18	8	14	12	6	50	48	56	26	56

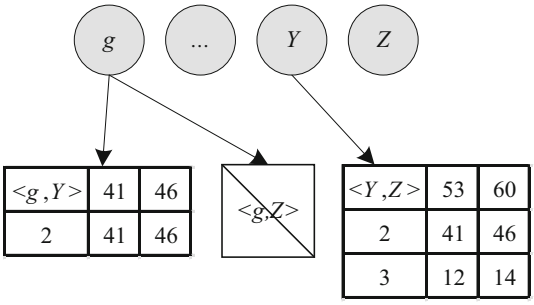


Fig. 8 BUHT structure

ever, since item g is an ancestor of item d, the 2-itemset $\{g,Z\}$ is not created in the BUHT structure.

The transaction data are processed by removing the item f that does not satisfy the *minutil*. The remaining items are sorted in descending order based on their utility values and levels ($Z \prec g \prec Y \prec U \prec d \prec e \prec c \prec V \prec a \prec b$). Using Optimization 2, the K-th largest utility value of the 2-itemsets is calculated using $Q_2 = [79,71,66,58,56,\ldots]$, and the threshold is raised to $Q_2\,[5] = 58$. Next, the subtree utility values of all (generalized) items in the $Secondary_{W_1}(\varnothing) = \{a, b, c, d, e, g, V, Y, U, Z\}$ are calculated to obtain the $Primary_{W_1}(\varnothing) = \{Z, g, Y, U, d, c\}$. Additionally, the SUPL structure of the items is updated.

First, the algorithm takes an item from $Primary_{W_1}(\varnothing)$ for depth-first search to generate the extended itemset Z as shown in Fig. 9. Since the utility value of itemset Z in W_1 is 56, Z is not a CLHUI and the *minutil* is not updated at this time. By calculating the local utility upper

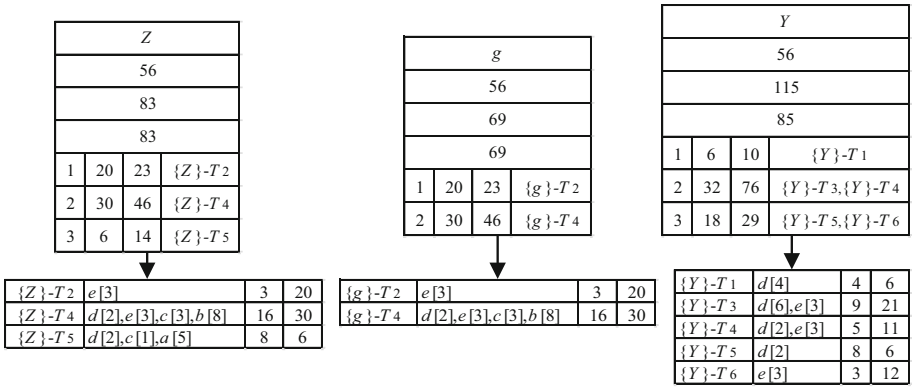


Fig. 9 Updated SUPL structures

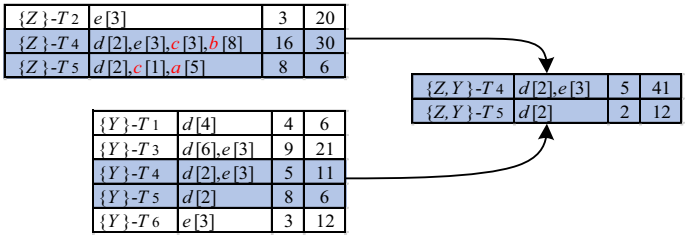


Fig. 10 Projection diagram of the itemset {*Z*,*Y*}

bound and the subtree utility upper bound of the item Z, $Primary_{W_1}(Z) = \{Y, U, d, e\}$ and $Secondary_{W_1}(Z) = \{Y, U, d, e, c, V\}$ are obtained. Therefore, item Z can be continued to be extended into a 2-itemset. Z is first extended by Y to generate the itemset {*Z*,*Y*}, which further generates the projected transaction sets for the 2-itemset {*Z*,*Y*} based on the projected window for item Z as shown in Fig. 10. In the projected window of item Z, there are descendant nodes of the generalized item Y, namely items a, b, c. Therefore, when creating the projected transactions for {*Z*,*Y*}, the descendant nodes of Y should be removed from the original transactions. At this point, the prefix utility of the projected transactions for {*Z*,*Y*} is the utility value of the itemset {*Z*, *Y*} in this transaction, which is equivalent to the prefix utility of item Z plus the utility values of the descendant nodes of Y. Since the utility value of item {*Z*,*Y*} in window W_1 is 53, the itemset is not a CLHUI, and the *minutil* remains unchanged. Finally, after the recursive process, the Top-K CLHUIs in window W_1 are {*Y*,*U*:79}, {*c*,*V*,*U*:79}, {*V*,*U*:71}, {*U*,*Z*:66}, {*Y*,*U*,*Z*:60}. As the window continues to slide forward, the transactions with batch number 1 are removed from the window, SUPL, and BUHT. Partial fields are updated, and the transactions with batch number 3 are added to window W_2. The subsequent operations in window W_2 are similar to those in the previous window. After the recursive process, the Top-K CLHUIs in window W_2 are {*Y*,*U*,*Z*:101}, {*d*,*Y*,*Z*:98}, {*V*,*U*,*Z*:97}, {*d*,*Y*,*Z*:94}, {*Y*,*Z*:88}.

4.5 TKCCLHM analysis

4.5.1 Complexity

The time consumption of TKCCLHM primarily occurs in the following steps:① The TKCCLHM algorithm scans the category information file, which contains X rows. In this category file, the general item count is denoted as |*I*|, and the leaf item count is represented as |*GI*|. The time complexity of scanning the category file is $O(X)$. Subsequently, it creates category nodes for each item based on the category file. Since each node can have at most one parent node, setting the level information for nodes takes O(|*GI*| +|*I*|) time. Assigning the "BottomLevel" attribute to nodes involves traversing all leaf nodes in the worst-case scenario and updating information along the ancestor chain of each node. Therefore, the time complexity is $TN_1 = O(|I|*L)$, where L denotes the number of levels from leaf nodes to the root node. ② During the process of scanning transaction data within the window, where w is the total number of transactions in the entire window, q is the number of transactions in the latest batch, and y is the number of items contained in each transaction (including generalized items and leaf items), assuming all are within the hierarchy range, constructing SUPL requires scanning the current window once. Therefore, the time for building the list in the first window is $TN_2 =$

$O(wy)$, and for subsequent windows, constructing and updating the list is $TN'_2 = O(qy)$. For the BUHT structure construction within the first window, it requires scanning the window once. Hence, the time consumed for BUHT in the first window is $TN_3 = O(wy^2)$. For subsequent windows, constructing and updating this structure is $TN'_3 = O(qy^2)$. ③ The threshold raising strategy employs a non-decreasing priority queue of length K to store data. Assuming that there are n utility values of items satisfying the *minutil* each time, the time complexity of storing them in the priority queue and adjusting their order is $TN_4 = O(n\log k)$.④ For each itemset z in the primary itemset of α, the TKCCLHM algorithm needs to perform window projection for itemset β within a time complexity of $O(wy\log y)$. Subsequently, it evaluates the utility value of itemset β and maintains a non-decreasing priority queue of length K. The time complexity for this step is TN_4, which is $O(n\log k)$, where n represents the size of itemset β. Finally, the computation of the local utility and subtree utility upper bound values for itemset v with respect to itemset β incurs a time complexity of $O(w(y + yw_1))$, where w_1 represents the set of ancestor nodes for items in the transactions, and q denotes the size of the primary itemset collection. Therefore, the time complexity of this process is $TN_5 = [O(wy\log y) + O(\log k) + O(w(y + yw_1))]*q$. ⑤ In the remaining sliding windows, time is required to handle old batches and update the SUPL and BUHT structures. In the worst-case scenario, where all items are present in the transactions within the window, the time consumption is $TN_6 = O(|GI| + |I| + p)$, with p representing the size of the BUHT structure. Assuming that the data stream flows into H windows before stopping, the total time complexity is $TN_1 + TN_2 + TN_3 + TN_4 + TN_5 + (H\text{-}1)(TN'_2 + TN'_3 + TN_4 + TN_5 + TN_6)$.

The space consumption of TKCCLHM primarily occurs in the following steps: ① Creation of the taxonomy tree. ② SUPL structure. ③ BHUT structure. ④ utility-bin array. Assuming each node occupies space N, the space occupied by the taxonomy tree is $O[(|GI| + |I|)*N]$. In the SUPL construction part for itemsets, assuming a list structure occupies space M, and the number of distinct items in the entire window is v, the space occupied by constructing the SUPL structure is $O(v*M)$. During the mining process, if the SUPL generated for itemsets of size i consumes space X, and assuming that the BUHT structure consumes memory S and utility-bin array structure occupies Q, the total space complexity is determined as $O[(|GI| + |I|)*N + (v + i)*M + S + Q]$.

4.5.2 Correctness and completeness

Algorithm 1 has a loop invariant in line 14. Its task is to traverse the transactions in the current window to calculate the utility of items, and create the SUPL structure. The loop invariant can be described as follows: If there are transactions in the array, then unprocessed transactions exist in the subarray $list[i] \sim list[winsize]$. Initialization: Initially, $i = 1$, so the loop invariant holds with the array being $list[i] \sim list[winsize]$. Maintenance: Define i as the current loop variable. At the beginning of each iteration, if there are transactions in the array, the items within these transactions not yet used to create the SUPL structure, and there are still unread transactions, then they are present in the subarray from $list[i]$ to $list[winsize]$. As i increases with each iteration, the loop invariant remains true before the next iteration. Termination: The loop terminates when all transactions in the current window have been read and processed.

The threshold raising process in Algorithm 1 is a looping procedure that maintains the Top-K items in a priority queue. Its task is to process itemsets meeting the *minutil*, append them to queue Q, and arrange them according to their utility values. The following loop invariant exists: The items before position i in queue Q are sorted, and $Q[1\ldots i]$ contains the Top-i itemsets in terms of utility values under the current threshold condition. Initialization: At the beginning, Q is an empty queue. The loop invariant holds during the initialization

phase. Maintenance: Assuming it holds before the current iteration, which means that the items in $Q[1\ldots i-1]$ are sorted, during the current iteration i, the current itemset is added to Q while ensuring that Q remains sorted in non-decreasing order of utility values. If adding the itemset exceeds the size of Q beyond K, the itemset with the lowest utility value is removed. By the conclusion of the current iteration, $Q[1\ldots i]$ is sorted. Termination: When the loop ends, it means that all itemsets have been traversed, and those satisfying the threshold have been added to Q while maintaining the order property.

The loop process in the 3 line of Algorithm 2 is tasked with finding items z in each projected transaction α-T from the projected window α-W of the prefix itemset α, which can be extended with the itemset α to generate the extended itemset $\beta = \alpha \cup \{z\}$. The generated projected transaction β–T is then added to the projected window of the extended itemset β, denoted as β–W. This results in the following loop invariant: When traversing the i-1th transaction in α-W, β-W includes all projected transactions β-T that contain item z found in the preceding i-1 transactions in α-W. Initialization: Before the first iteration, β–W is empty, and the loop invariant holds during initialization. Maintenance: Assuming the loop invariant holds before the current iteration, which means that β–W contains all projected transactions β–T have found in the $i-1$ transactions of α–W that contain item z, during the current iteration i, the algorithm traverses the i-th transaction in α–W and checks if item z is present. If it is, a new projected transaction β–T is created and added to β–W. Consequently, after the current iteration, β–W contains all the projected transactions β–T that include item z found in the first i transactions of α-W. Termination: When the last transaction in α–W has been traversed, the loop terminates, at which point the complete projection window for the extended itemset β is obtained.

The outer loop process in lines 1–10 of Algorithm 2 is responsible for evaluating whether the itemset β, generated by extending itemset α, satisfies the current *minutil*. Thus, a loop invariant exists: The items in the subarray $primaryof\alpha[0\ldots i-1]$ have been processed, meaning they have already been extended with prefix items to form β and assessed for satisfying the current *minutil*. Initialization: Prior to the first iteration, none of the items in the subarray $primaryof\alpha[0\ldots i-1]$ have been processed, indicating that they have not yet been extended with the prefix itemset and evaluated for compliance with the current *minutil*. Therefore, the loop invariant holds during initialization. Maintenance: Assuming the loop invariant holds before the current iteration, which means that items in the subarray $primaryof\alpha[0\ldots i-1]$ have been processed, during the current iteration, the algorithm processes $primaryof\alpha[i]$. After processing $primaryof\alpha[i]$, the items in the subarray $primaryof\alpha[0\ldots i]$ have been processed. Therefore, the loop invariant holds during the maintenance phase. Termination: When $i > primaryof\alpha.size$-1, the subarray $primaryof\alpha$ has been traversed completely. All extensible itemsets β have been checked for satisfaction of the current *minutil*. Hence, the loop invariant holds during the termination phase.

5 Experiments and results

In this section, we compare the proposed algorithm with the state-of-the-art algorithm TKC in the field of mining CLHUIs. All experiments are conducted on a computer running Ubuntu 16.04.6 LTS operating system with an Intel(R) Xeon(R) Gold 6154 CPU operating at a frequency of 3.00 GHz and 256 GB of memory. The experiments are conducted using six sparse and dense datasets with taxonomies. The basic characteristics of the datasets are shown in Table 6. In the table header, the attributes from left to right represent the dataset

Table 6 Basic characteristics of the datasets

Datasets	|D|	T_{avg}	|I|	|GI|	*MaxLevel*	*MaxBottomLevel*	Type
Liquor	90,826	10.28	8595	78	7	3	Sparse
Fruithut	181,970	3.58	1265	43	4	2	Sparse
Foodmart	54,537	4.60	1568	167	6	4	Sparse
Connect	67,557	43.00	129	40	4	3	Dense
Chess	3,196	37.00	75	30	3	3	Dense
Mushroom	8,124	23.00	119	31	2	2	Dense

name, the number of transactions in the dataset, the average transaction length, the number of non-generalized items, the number of generalized items, the maximum hierarchy level, the maximum BottomLevel in the taxonomy, and the dataset type. The datasets Foodmart and Fruithut contain built-in taxonomies; while, the rest of the datasets have synthetic taxonomies. The datasets are obtained from the SPMF website and the GitHub website [11]. The experiments compare the performance of the proposed TKCCLHM algorithm with the TKC algorithm, which is currently the state-of-the-art algorithm for mining Top-K CLHUIs from static transaction data. Additionally, three variants of TKCCLHM are designed in this paper: (1) TKCCLHM (without optimization), where the algorithm does not use any minimum utility threshold raising strategies. (2) TKCCLHM (with optimization 1), where the algorithm only uses threshold raising strategy 1. (3) TKCCLHM, where the algorithm uses both threshold raising strategies 1 and 2 simultaneously.

In this paper, two variants of the TKC algorithm are designed: (1) TKC (with optimization 2), where the TKC algorithm applies both threshold raising strategies 1 and 2. (2) TKC (Stream), which is an improvement of the TKC algorithm for data stream environments. The experiments are mainly divided into four parts: (1) evaluating the effect of different K values on algorithm performance; (2) evaluating the effect of BottomLevel constraints on algorithm performance; (3) evaluating the effect of different window sizes on algorithm performance; (4) scalability experiments. To ensure the stability of the experimental results, the data presented in this section is the average of six test runs. By varying the relevant parameters, the performance of different algorithms are compared by running them on six datasets until the algorithm became too slow, run out of memory, or an obvious winner is observed.

5.1 Effect of different *K* values on algorithm performance

Due to the proposed algorithm being the first algorithm for mining Top-K CLHUIs in a data stream environment, there is no comparable algorithm with similar performance to compare it with. The method to evaluate its reliability is to compare it with the existing TKC algorithm for processing static transaction data. Therefore, this experiment adopts the method of single-batch execution, where the sliding window size is set to 1 and the batch size is set to the number of transactions in the dataset. The algorithm's BottomLevel range is set to the full range of the taxonomy, i.e., $[1, MaxBottomLevel(\tau)]$. The runtime variations of the algorithms in the static data environment on the dataset under different K values are shown in Fig. 11. The number of candidates generated by all the algorithms with different K-value settings is shown in Table 7. The experimental results show that the proposed algorithm

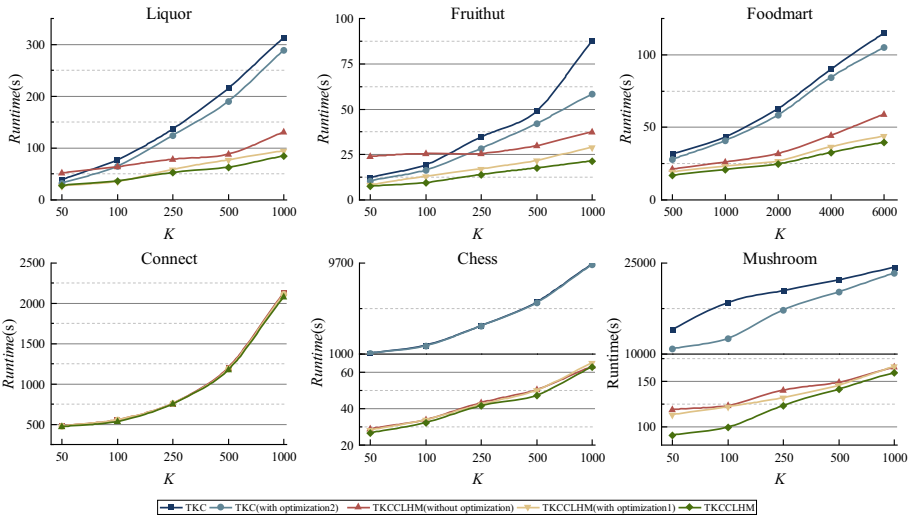


Fig. 11 Runtime comparison for different values of K in a static environment

outperforms the TKC algorithm in terms of runtime on both sparse and dense datasets. It is worth mentioning that the TKC algorithm did not display experimental results on the Connect dataset because it exceeded a runtime of 10 h. When the value of K is low, the initial *minutil* of the algorithm tends to be high. As the value of K increases, the *minutil* decreases, leading to an increase in the number of candidates generated, and consequently, the runtime of the algorithm increases. The experiments demonstrate that the inclusion of threshold optimization strategies has a significant impact on the performance of the algorithm. Each introduction of a threshold optimization strategy reduces the time consumption and the number of candidates generated. In the sparse datasets, the three variants of the TKCCLHM algorithm exhibit a reduction in the number of candidates by 1–2 orders of magnitude. The inclusion of threshold optimization strategy 2 in the TKC algorithm results in a noticeable decrease in the number of candidates across all datasets. The TKCCLHM algorithm performs exceptionally well on dense datasets such as Connect, Chess, and Mushroom, being 2–3 orders of magnitude faster than the TKC algorithm. The main reasons for this are as follows: (1) It employs transaction projection techniques to reduce the size of transaction sets by projecting within the window, thus reducing transaction scanning costs. (2) Instead of using join operations on utility-lists, the algorithm utilizes utility-bin array to calculate the upper bounds of itemsets in linear time. (3) It applies local utility and subtree utility upper bounds to prune the search space, which are two tighter bound compared to the remaining utility upper bound used by the TKC algorithm. As a result, a significant number of unpromising itemsets can be quickly filtered out during the mining process. On the Fruit dataset, when $K < 250$, it is normal for the TKCCLHM (without optimization) algorithm, which does not use any threshold optimization strategy, to have lower performance compared to the TKC (with optimization 2) and TKC(Stream) algorithms that utilize threshold optimization strategies. The reason for this is that the algorithm without any utility threshold raising strategy cannot initially reduce the size of the primary and secondary items. Therefore, it has a larger search space, resulting in longer runtime as it needs more time to filter itemsets. When K is low, the corresponding *minutil* is also higher. Even though the local utility and subtree utility upper bounds are more

Table 7 Number of candidates for different values of K in a static environment

Dataset	K	TKC	TKC(with optimization2)	TKCCLHM(without optimization)	TKCCLHM(with optimization1)	TKCCLHM
Liquor	50	13,655	4509	12,113	1352	319
	100	43,516	14,847	14,054	2526	674
	250	220,227	81,498	19,793	6353	1837
	500	604,672	216,544	28,622	12,628	3813
	1000	1,500,217	578,612	46,656	25,763	8256
Fruithut	50	7253	3271	69,852	799	259
	100	20,863	8127	73,003	1681	528
	250	109,337	29,793	82,431	5462	1472
	500	355,540	82,627	99,432	14,419	3245
	1000	17,941,237	199,488	154,342	154,069	7511
Foodmart	500	256,116	109,118	999,472	11,016	3510
	1000	521,285	255,608	1,255,230	19,830	7619
	2000	2,285,013	601,628	1,614,408	47,645	16,865
	4000	4,836,788	1,338,323	2,998,120	967,649	37,720
	6000	9,065,766	2,156,863	3,411,277	1,250,539	59,422
Connect	50	–	–	3451	3172	2881
	100	–	–	5629	5358	4797
	250	–	–	12,600	12,600	11,057
	500	–	–	25,696	25,696	23,405
	1000	–	–	53,607	53,607	49,343
Chess	50	419,768	419,398	13,080	12,412	10,302
	100	719,823	719,223	17,486	17,479	13,059

Table 7 (continued)

Dataset	*K*	TKC	TKC(with optimization2)	TKCCLHM(without optimization)	TKCCLHM(with optimization1)	TKCCLHM
	250	1,509,237	1,508,159	27,497	27,497	19,302
	500	2,479,295	2,477,650	40,246	40,246	27,854
	1000	4,079,831	4,077,231	62,551	62,551	44,181
Mushroom	50	33,344,721	12,983,166	297,640	215,722	101,939
	100	55,928,045	16,514,263	326,338	296,011	121,829
	250	72,801,491	32,490,471	391,147	391,147	211,778
	500	77,521,428	40,557,355	480,598	480,598	303,825
	1000	81,623,883	48,731,745	602,864	602,864	413,479

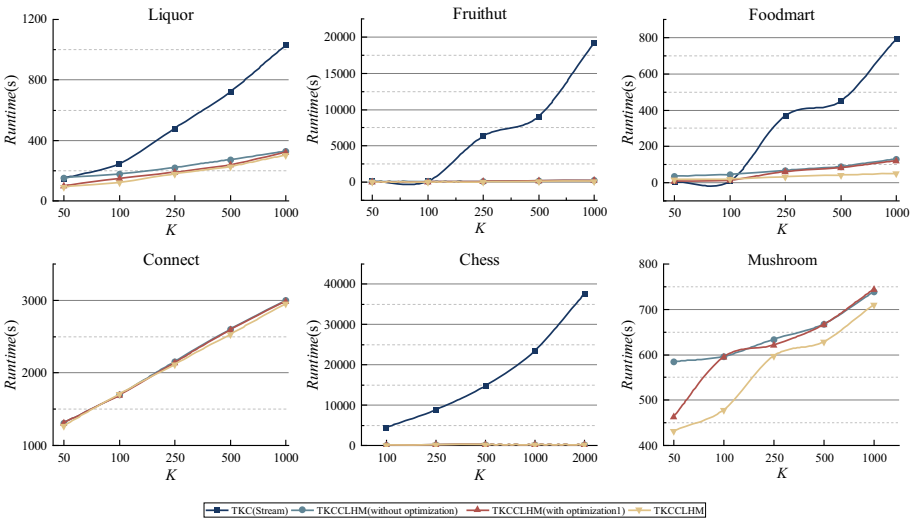


Fig. 12 Runtime comparison for different values of K over data stream

compact compared to using the remaining utility upper bound, the pruning effect is not very significant at this stage. However, as K increases, the compact upper bounds can eliminate more intermediate itemsets early in the process, and their advantage becomes more evident. Additionally, the remaining utility pruning strategy in the TKC algorithm incorrectly prunes some itemsets, leading to inaccurate mining results. For example, in the Foodmart dataset, when $K = 50$, the TKC algorithm misses the CLHUI {1674, 1721}, and when $K = 100$, it misses the CLHUIs {1673, 1721} and {1659, 1721}.

Afterward, the proposed algorithm in this paper is compared to the variant TKC(Stream) of the algorithm TKC in a data stream environment. The window sizes are fixed at 5, 5, 3, 3, 5, and 3 for the datasets Liquor, Fruithut, Foodmart, Connect, Chess, and Mushroom, respectively. The batch sizes are set at 1000, 1000, 500, 500, 500, and 500 for the respective datasets. The time comparison of the algorithms as the value of K increases is shown in Fig. 12. The memory consumption and the number of generated candidates for the four algorithms on the datasets are shown in Table 8.

The results indicate that, across the six datasets, the runtimes and the number of candidates generated by the TKCCLHM algorithm and its three variants are significantly lower than those of the TKC(Stream) algorithm. For sparse datasets, the TKCCLHM algorithm is up to approximately 2.39 times, 291.31 times, and 14.33 times faster than the TKC(Stream) algorithm on the Liquor, Fruithut, and Foodmart datasets, respectively, and the number of generated candidates is reduced by 2–3 orders of magnitude. On dense datasets, the runtime of the TKC(Stream) algorithm exceeds 10 h for the Connect and Mushroom datasets, whereas the TKCCLHM algorithm is at least 27.48 times and 82.42 times faster on these datasets, respectively, and 202.09 times faster on the Chess dataset. Additionally, the effectiveness of the two threshold raising strategies used in the TKCCLHM algorithm can be observed from the number of generated candidates. Compared to the TKCCLHM(without optimization) algorithm that does not use threshold raising strategies and the TKCCLHM(with optimization1) algorithm that only uses the threshold raising strategy for utilities of 1-itemsets, the proposed TKCCLHM algorithm uses both the 1-itemsets and 2-itemsets threshold raising

Table 8 Performance comparison for different values of K over data stream

Algorithms		TKC(Stream)		TKCCLHM(without optimization)		TKCCLHM(with optimization1)		TKCCLHM	
Dataset	K	Memory	Candidates	Memory	Candidates	Memory	Candidates	Memory	Candidates
Liquor	50	2085.4	1,174,344	8211.8	484,233	8219	115,444	9613.8	26,757
	100	2244.2	3,746,507	8577	655,893	8245.6	225,626	10,182.2	58,370
	250	222.6	18,865,018	8179.8	1,127,375	6589.2	635,074	8920.6	160,700
	500	2404.6	50,642,336	10,139	1,829,236	10,132	1,558,891	10,168.2	344,269
	1000	8650.6	105,383,260	10,204	3,093,624	10,181.8	3,067,865	8133.6	820,745
Fruithut	50	1716.2	1,336,078	2382.6	57,241,928	1553.4	141,868	2145	46,075
	100	1921	4,032,108	2497.8	64,501,739	2085.6	315,388	2074.6	94,462
	250	6048	7,767,976,070	2328.4	74,209,815	2208.2	73,829,685	2211.8	262,162
	500	5849.2	11,532,057,934	2340.2	99,342,634	2249.4	99,340,591	2505.6	608,117
	1000	6248.2	21,906,161,388	2416.8	125,425,299	2319.8	125,425,299	2550.2	1,506,340
Foodmart	50	514.4	314,186	1500.4	15,416,602	1035.4	94,075	2020.8	38,004
	100	514.8	970,390	1692	21,101,615	1289	189,217	2098.6	72,263
	250	3786.8	611,368,262	1944	34,426,564	1768.4	34,174,152	3676.8	175,362
	500	3917.4	808,022,934	1773.6	43,465,944	1521.6	43,352,269	2368.4	394,323
	1000	4337.8	1,498,304,355	1495.8	55,240,749	1628.4	55,205,434	3354.8	917,324
Connect	50	–	–	4160	429,420	4121.6	410,733	4121.6	378,099
	100	–	–	4116.6	722,992	4119	712,398	3508.6	645,131
	250	–	–	4119.2	1,690,297	4119	1,690,297	4104.2	1,548,575
	500	–	–	4121.4	3,433,168	4120.8	3,433,168	4452.6	3,196,718
	1000	–	–	4251.4	6,940,989	4246.4	6,940,989	4128.4	6,493,047

Table 8 (continued)

Algorithms		TKC(Stream)		TKCCLHM(without optimization)		TKCCLHM(with optimization1)		TKCCLHM	
Dataset	*K*	Memory	Candidates	Memory	Candidates	Memory	Candidates	Memory	Candidates
Chess	100	1427	2,149,248	2077.4	49,289	2077.8	49,275	2071.6	37,635
	250	1425.2	4,451,407	2097.2	114,105	2080.6	77,043	2076	55,589
	500	1425.6	7,429,245	2105.4	114,038	2104.6	114,038	2085.2	83,691
	1000	1429.8	12,212,077	4115.4	178,923	4115.4	178,932	4117.8	132,962
	2000	1452.6	19,880,932	4121.8	289,625	4121.4	289,625	4126.8	240,385
Mushroom	50	–	–	4253.2	20,869,537	4109	9,596,189	4110.8	2,050,306
	100	–	–	4106	21,384,407	4105.6	20,996,860	4111.2	2,712,951
	250	–	–	4106.75	22,635,865	4105.6	22,635,865	4111	3,939,137
	500	–	–	4106.8	24,291,221	4106.8	24,291,221	4109.4	6,748,428
	1000	–	–	4110.8	26,997,632	4110.2	26,997,632	4104.4	10,191,151

strategies to generate the fewest candidates and achieve the best runtime. However, the 1-itemsets threshold raising strategy fails for certain K values in some dense datasets. This is because the average transaction length in dense datasets is generally higher than in sparse datasets, and the frequency of occurrence of the same items in different transactions increases. Assuming the external utility of these items remains the same, their utility increases with the increased frequency. As a result, compared to the items in sparse datasets, these items are more likely to satisfy the *minutil* based on their local utility and subtree utility upper bounds. In the mining process, even with the use of the 1-itemsets threshold raising strategy before the Search procedure, it is unable to reduce the size of the primary and secondary items. Therefore, the performance of the 1-itemsets threshold raising strategy is not satisfactory for dense datasets. However, the 2-itemsets threshold raising strategy raises the threshold to the K-th largest value among all 2-itemsets in the current sliding window, which can reduce the size of the primary items to some extent, and the size of the primary items is directly related to the search space of the algorithm. In terms of memory consumption, as the value of K increases, the relative decrease in the *minutil* leads to an increase in the search space, resulting in an upward trend in memory usage. The TKCCLHM algorithm uses the SUPL structure, which stores a large number of projected transaction sets, and the 2-itemsets threshold optimization strategy requires the use of the BUHT structure to maintain the utility information of all 2-itemsets in the window, resulting in higher memory consumption.

5.2 Effect of BottomLevel constraints on algorithm performance

The TKCCLHM algorithm proposed in this paper allows for flexible mining of CLHUIs by constraining the BottomLevel range of items in the taxonomy tree. The algorithm is executed on the Liquor and Chess datasets, and the mining BottomLevel constraints are set based on the MaxBottomLevel feature of the datasets, resulting in six range segments. The number of CLHUIs, candidates, runtime, and memory usage under different BottomLevel constraint ranges are presented in Table 9.

From Table 9, it can be observed that the TKCCLHM algorithm can mine the Top-K CLHUIs within different BottomLevel ranges. Due to the varying utility of itemsets at different BottomLevels, there are differences in the threshold increment within the specified BottomLevel range under the same K value, resulting in variations in the execution time. It can be noticed that the number of candidates and memory consumption are not consistent. This is because the TKCCLHM algorithm uses the 2-itemsets threshold raising strategy during the mining process, and the BUHT structure stores the utility of 2-itemsets within the BottomLevel constraint range that does not have ancestor–descendant relationships, which consumes a large amount of memory space. Taking the Liquor dataset as an example, compared to mining only the itemsets on $BottomLevel \in [1, 1]$, mining the entire level range of the taxonomy tree results in fewer candidates and should consume less memory space. However, due to the BUHT structure storing the utility information of all 2-itemsets on the taxonomy tree, the overall space usage is larger than the BUHT structure that only stores the 2-itemsets on the $BottomLevel \in [1, 1]$.

5.3 Effect of different window sizes on algorithm performance

Since the algorithm is based on the sliding window model, the *WinSize* is a crucial parameter. Experiments in this subsection primarily compare the performance of algorithms by controlling the *WinSize*. To analyze the effect of this parameter on algorithm performance, the

Table 9 TKCCLHM performance under different BottomLevel constraint ranges

Dataset	Liquor($K = 2000$)				Chess($K = 500$)			
Range	Runtime(S)	Memory (MB)	Candidates	CLHUIs	Runtime(S)	Memory (MB)	Candidates	CLHUIs
[1,3]	84.766	10,136.8	8256	1000	47.035	4131.8	27,854	500
[2,3]	29.906	6630.6	17,247	1000	20.6	4130	44,986	500
[3,3]	2.21	591	34	35	0.244	135	31	31
[1,2]	58.688	7625	7628	1000	501.855	1255	828,110	500
[2,2]	18.077	4178.6	19,920	1000	105.488	4114.4	1,503,949	500
[1,1]	10.213	1418.2	12,166	1000	19.887	4106	223,384	500

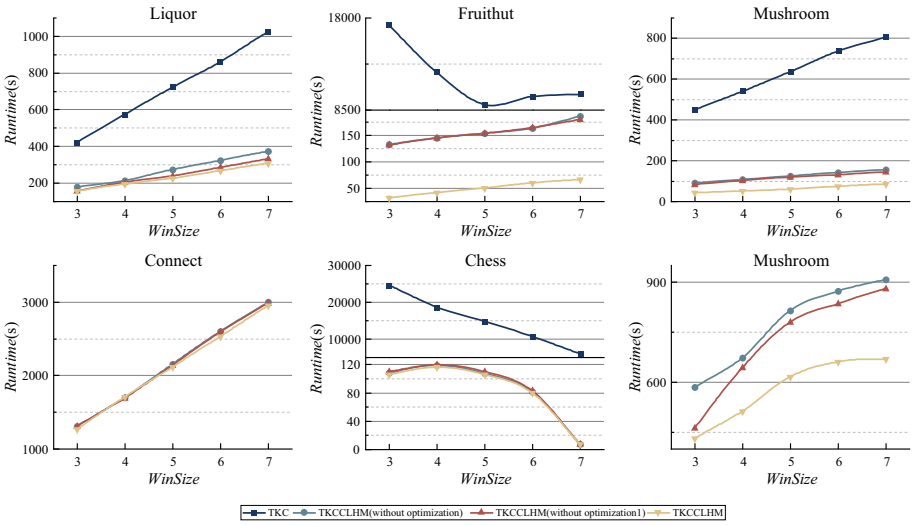


Fig. 13 Comparison of runtime for different window sizes

experiments are conducted on the Liquor, Fruithut, Foodmart, Connect, Chess, and Mushroom datasets. The K values are fixed at 500, 500, 500, 50, 500, and 50, respectively, and the batch sizes are set to 1000, 1000, 500, 500, 500, and 500. The window size varied from 3 to 5. The runtime of the algorithms on the six datasets with different *WinSize* values as shown in Fig. 13. The memory consumption and the number of candidates generated by the four algorithms on the datasets are summarized in Table 10. The experimental results indicate that as the *WinSize* increases, the TKCCLHM algorithm consistently consumes the least amount of time; while, the TKC(Stream) algorithm incurs the highest time cost. For sparse datasets, the TKCCLHM algorithm is significantly faster than the TKC (Stream) algorithm, achieving speedups of approximately 2.33, 548.8, and 9.84 times on the Liquor, Fruithut, and Foodmart datasets, respectively. The average number of candidates generated by TKCCLHM is only 0.695%, 0.004%, and 0.04% of that generated by TKC (Stream). For dense datasets, the runtime of the TKC (Stream) algorithm exceeds 10 h on the Connect and Mushroom datasets so is not included in the figure. In comparison, the TKCCLHM algorithm is at least 27.48 and 82.42 times faster than TKC (Stream) on these two datasets and achieves a speedup of 834.80 times on the Chess dataset. The average number of candidates generated by TKCCLHM is only 1.053% of that generated by TKC(Stream).

As the number of transactions within the window increases, it can be observed that not all algorithms exhibit a consistent upward trend in the generation of candidates. In traditional CLHUIM algorithms, the number of intermediate itemsets is generally proportional to the window size and runtime. This is because a larger window can accommodate more transactions, and itemsets within the window have higher utility compared to smaller windows, making it easier to meet the *minutil*. However, with a fixed value of K, this also leads to a relatively higher *minutil*. Therefore, the pruning conditions of the algorithm are associated with the continuously changing *minutil*. As a result, the number of candidates may not necessarily increase proportionally with the window size or time for the same dataset. Additionally, our proposed TKCCLHM algorithm generates fewer candidates as the window size increases, but the overall runtime increases. This is because a larger window contains

Table 10 Comparison of algorithm performance for different window sizes

Algorithms		TKC(Stream)		TKCCLHM(without optimization)		TKCCLHM(with optimization1)		TKCCLHM	
Dataset	*WinSize*	Memory	Candidates	Memory	Candidates	Memory	Candidates	Memory	Candidates
Liquor	3	1799	49,738,226	6620.6	1,818,794	6604.6	1,714,270	10,157.2	381,535
	4	2166.4	50,712,968	8221	1,825,101	8213.6	1,654,549	9289.8	363,730
	5	2404.6	50,642,336	10,139	1,829,236	10,132	1,558,891	10,168.2	344,269
	6	2529.4	50,230,933	8186.2	1,833,976	8182	1,472,990	7804.4	332,669
	7	2774	50,262,564	8960.8	1,842,168	7813.4	1,450,463	8529.6	326,746
Fruithut	3	6607.2	19,712,337,349	2411.2	99,179,943	2141.4	99,179,506	2081.8	641,979
	4	5737.2	14,871,791,870	2439.4	99,942,628	2322.4	99,941,420	2070.2	621,874
	5	5849.2	11,532,057,934	2340.2	99,342,634	2249.4	99,340,591	2505.6	608,117
	6	6264.8	12,486,100,762	2430.4	98,982,185	2357.6	98,979,123	2203.8	598,635
	7	6400.6	12,741,561,502	2943.6	101,947,747	2415.2	101,943,491	2227.8	591,962
Foodmart	3	3917.4	808,022,934	1773.6	43,465,944	1521.6	43,352,269	2368.4	394,323
	4	3944	912,277,366	1903.8	51,049,068	2083.2	50,880,298	4138.4	383,861
	5	3903.6	977,592,270	1784.6	54,734,350	1509.8	54,495,998	4165.6	376,602
	6	3986.6	1,077,234,652	1617.2	56,615,484	1570.6	55,831,939	5152.2	371,132
	7	3788	1,133,236,414	1720	56,217,835	1754.6	54,380,905	4152	366,015
Connect	3	–	–	4160	429,420	4121.6	410,733	4121.6	378,099
	4	–	–	4160	424,721	4138.4	404,779	4145.4	370,829
	5	–	–	4159.8	423,808	4118.6	402,123	4138.8	367,780
	6	–	–	4126.6	423,263	4123	400,724	4257.2	366,126
	7	–	–	4146.6	422,261	4149.8	398,895	4172.8	364,280

Table 10 (continued)

Algorithms		TKC(Stream)		TKCCLHM(without optimization)		TKCCLHM(with optimization1)		TKCCLHM	
Dataset	*WinSize*	Memory	Candidates	Memory	Candidates	Memory	Candidates	Memory	Candidates
Chess	3	1447.8	19,433,831	2117	195,063	2118.2	195,063	2093.4	141,267
	4	1436	11,278,876	2106	160,293	2107	160,293	2082	115,066
	5	1425.6	7,429,245	2105.4	114,038	2104.6	114,038	2085.2	83,691
	6	1426.2	4,682,412	2099.2	77,321	2098.8	77,321	2086.8	53,537
	7	1423.4	2,479,295	1030.4	35,513	1030.2	35,513	1031.4	30,947
Mushroom	3	–	–	4253.2	20,869,537	4109	9,596,189	4111.8	2,050,306
	4	–	–	4149.8	21,215,308	4108	7,175,092	4082.4	1,973,580
	5	–	–	4124	20,472,542	4107	7,705,016	4117.4	2,218,258
	6	–	–	4142.8	14,995,333	4107.8	7,060,670	4114.4	2,178,976
	7	–	–	4105.6	12,515,811	4108.2	6,506,976	4107.8	1,992,529

a greater variety of items, which requires more time to process the dataset and generate the BUHT structure. It also involves creating more SUPL structures and maintaining additional batch information. In terms of memory usage, the TKCCLHM algorithm consumes more memory space because it stores more projected transaction data and other information. However, due to the significant difference in candidates between TKCCLHM and TKC(Stream) algorithms, particularly on the Fruithut and Foodmart datasets, the memory performance of TKCCLHM is not significantly different from TKC(Stream), and in some cases, TKCCLHM even outperforms it.

5.4 Scalability

In this subsection, scalability experiments are conducted on six datasets to evaluate the performance of the TKCCLHM algorithm and verify that the runtime of the algorithm does not exhibit exponential growth. To analyze the effect of dataset size on algorithm performance, the datasets Liquor, Fruithut, Foodmart, Connect, Chess, and Mushroom are used. The K values are fixed at 500, 500, 500, 50, 500, and 50, and the batch sizes are set at 1000, 1000, 1000, 500, 500, and 500; while, the window size are fixed at 5. The experiment are conducted by varying the dataset size from 20% to 100%. The runtime of the TKC(Stream) algorithm and the TKCCLHM algorithm on the datasets is shown in Fig. 14. The TKC(Stream) algorithm did not display experimental results on the Connect and Mushroom dataset, its performance was not fully shown when the parameter exceeded 40%. This was due to the fact that the algorithm's runtime exceeded 10 h, resulting in incomplete performance results. As the dataset size increases, the algorithms need to process more windows, resulting in an overall increase in runtime. However, from the experimental results, it can be observed that the runtime of the proposed TKCCLHM algorithm grows steadily on the datasets; while, the runtime of the TKC(Stream) algorithm increases at a much higher rate. Furthermore, as the number of transactions increases, the difference between the two algorithms becomes

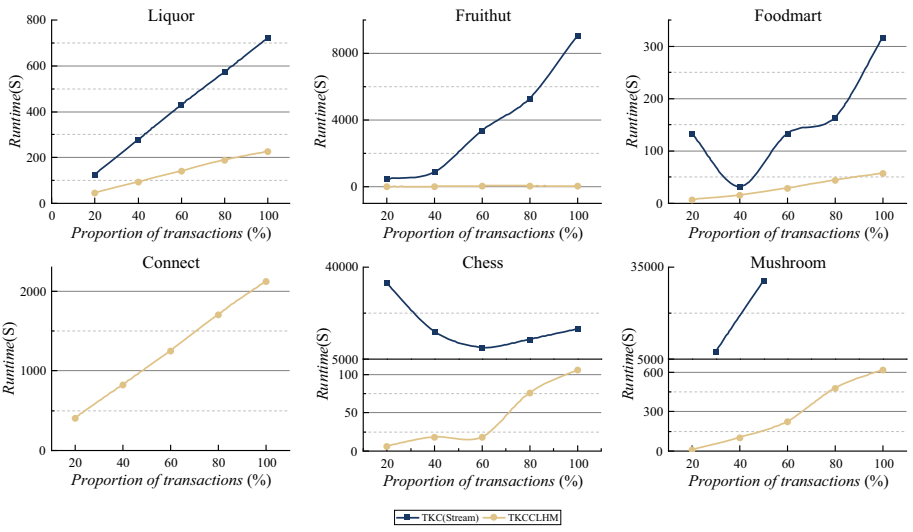


Fig. 14 Scalability comparison of algorithm runtime

larger. Therefore, the proposed TKCCLHM algorithm exhibits good scalability compared to the TKC(Stream) algorithm, with better performance on dense datasets. The main reason is that the TKCCLHM algorithm utilizes transaction projection techniques to extend itemsets in a pattern-growth manner. As the search space increases, the projection transactions of itemsets within the window continuously shrink, reducing the transaction scanning time to some extent. Additionally, the algorithm employs tighter local utility and subtree utility upper bounds, which are computed in linear time, compared to the remaining utility upper bound. In contrast, the TKC(Stream) algorithm utilizes utility-list structures. On dense datasets, the average length of transactions within a window is greater than that in sparse datasets, resulting in increased lengths of utility-lists for itemsets. However, the extension of itemsets in this algorithm requires join operations on utility-lists, leading to more time consumption on dense datasets compared to sparse datasets.

6 Conclusion

This paper introduces the TKCCLHM algorithm, a novel approach to Top-K constrained cross-level high-utility itemset mining based on sliding windows. It addresses the limitations of existing cross-level high-utility itemset mining algorithms, which can only handle static data. Furthermore, current algorithms are unable to explore relationships between items within specific hierarchical ranges. This limitation not only results in returned result sets containing itemsets with significant hierarchical differences, which is unfavorable for subsequent data analysis, but also significantly increases the computational complexity by requiring consideration of the entire taxonomy of items during mining. The algorithm defines the concept of BottomLevel hierarchy to represent items closer to the lower abstract levels in the taxonomy tree, thus controlling the abstraction level of the mined itemsets. By setting appropriate constraint ranges, the mined itemsets can have both sufficient detail and a certain level of generality. This approach allows for more interpretable pattern results without losing important information. The algorithm designs a sliding window-based SUPL structure to store utility information and applies transaction projection techniques to reduce the cost of scanning transactions within the window. Local utility and subtree utility upper bounds are used instead of the remaining utility upper bound to further accelerate the mining process. Additionally, a BUHT structure is designed to store the utilities of 2-itemsets within the range of hierarchical constraints and the *GWU* values. Based on this, the *minutil* raising strategy suitable for general item existence is proposed, and the 1-itemsets threshold raising strategy is applied within the sliding window to accelerate the raising of the *minutil* and prune the search space of itemsets in advance. Through extensive comparative experiments, it has been shown that the proposed algorithm can effectively mine Top-K CLHUIs over data streams while being able to flexibly mine itemsets at specific hierarchical levels.

However, the SUPL structure of the TKCCLHM algorithm is relatively complex, and the BUHT structure requires additional storage for 2-itemsets information, resulting in high memory consumption. In the next steps, we plan to design a compressed list indexing structure based on memory reuse strategies. This will involve considering the merging of identical transaction tuples to reduce memory usage. Our goal is to further enhance the algorithm's efficiency by improving the data structures and designing efficient pruning and threshold optimization strategies. Additionally, the results produced by the TKCCLHM algorithm still contain redundant data. Our future work will focus on extending this algorithm to mine

closed-constrained cross-level highutility itemsets over data streams based on taxonomy with the aim of refining results and eliminating redundancy.

Acknowledgements This work was supported by the National Nature Science Foundation of China (62062004) and the Ningxia Natural Science Foundation Project (2023AAC03315).

Author contributions MH contributed to writing—review & editing, supervision, funding acquisition, and resources. SL contributed to conceptualization, methodology, software, and writing—original draft. ZG contributed to data curation, investigation, and visualization. DM contributed to formal analysis and project administration. AL contributed to validation.

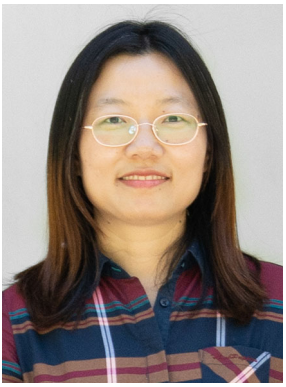
Declarations

Conflict of interest The authors declare that they have no conflict of interest.

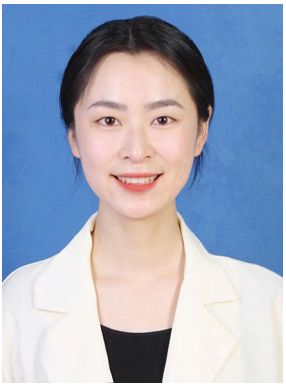
References

1. Han M, Zhang N, Wang L, Li XJ, Cheng HD (2023) Mining closed high utility patterns with negative utility in dynamic databases. Appl Intell 53(10):11750–11767
2. Lin JC-W, Djenouri Y, Srivastava G, Yun U, Fournier-Viger P (2021) A predictive GA-based model for closed high-utility itemset mining. Appl Soft Comput 108:107422
3. Dawar S, Sharma V, Goyal V (2017) Mining top-k high-utility itemsets from a data stream under sliding window model. Appl Intell 47(4):1240–1255
4. Srikant R, Agrawal R (1997) Mining generalized association rules. Futur Gener Comput Syst 13(2–3):161–180
5. Hipp J, Myka A, Wirth R, Güntzer U (2016) A new algorithm for faster mining of generalized association rules. Proceedings of the Principles of Data Mining and Knowledge Discovery: Second European Symposium, PKDD'98 Nantes. Springer, Berlin and Heidelberg, Berlin, pp. 74–82
6. Sriphaew K, Theeramunkong T (2002) A new method for finding generalized frequent itemsets in generalized association rule mining. In: Proceedings of the ISCC 2002 seventh international symposium on computers and communications. CA: IEEE Computer Society, Los Alamitos, pp. 1040–1045
7. Zhong M, Jiang T, Hong Y, Yang XH (2019) Performance of multi-level association rule mining for the relationship between causal factor patterns and flash flood magnitudes in a humid area. Geomat Nat Haz Risk 10(1):1967–1987
8. Baralis E, Cagliero L, Cerquitelli T, Garza P (2012) Generalized association rule mining with constraints. Inf Sci 194:68–84
9. Cagliero L, Chiusano S, Garza P, Ricupero G (2017). Discovering high-utility itemsets at multiple abstraction levels. In: Proceedings of the European conference on advances in databases and information systems. Switzerland: Springer, Cham, pp. 224–234
10. Fournier-Viger P, Wang Y, Lin JC-W, Luna JM, Ventura S (2020) Mining cross-level high utility itemsets. In: Proceedings of the International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, Switzerland: Springer, Cham, pp. 858–871
11. Tung NT, Nguyen LTT, Nguyen TDD, Fournier-Viger P, Nguyen N-T, Vo B (2022) Efficient mining of cross-level high-utility itemsets in taxonomy quantitative databases. Inf Sci 587:41–62
12. Nouioua M, Wang Y, Fournier-Viger P, Lin JC-W, Wu JM-T (2021) Tkc: mining top-k cross-level high utility itemsets. In: Proceedings of the 2020 international conference on data mining workshops. New York, IEEE, pp. 673–682
13. Liu M, Qu J (2012) Mining high utility itemsets without candidate generation. In: Proceedings of the 21st ACM international conference on Information and knowledge management. Maui, HI, USA pp. 55–64
14. Fournier-Viger P, Wu C W, Zida S, Zida S, Tseng VS (2014) FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning. In: Proceedings of the International symposium on methodologies for intelligent systems. Roskilde, Denmark, pp. 83–92
15. Krishnamoorthy S (2015) Pruning strategies for mining high utility itemsets. Expert Syst Appl 42(5):2371–2381
16. Zida S, Fournier-Viger P, Lin JC-W, Wu CW, Tseng VS (2017) EFIM: a fast and memory efficient algorithm for high-utility itemset mining. Knowl Inf Syst 51(2):595–625

17. Peng A Y, Koh Y S, Riddle P (2017) mHUIMiner: a fast high utility itemset mining algorithm for sparse datasets. In: Proceedings of the advances in knowledge discovery and data mining: 21st pacific-asia conference. Jeju, South Korea pp. 196–207
18. Krishnamoorthy S (2017) HMiner: efficiently mining high utility itemsets. Expert Syst Appl 90:168–183
19. Jiang H, Li X, Wang HJ, Wei JH (2022) Cross-level high utility itemset mining algorithms based on data index structure. J Comput Appl 43(7):2220
20. Tung N, Nguyen LT, Nguyen TD, Kozierkiewicz A (2021) Cross-level high-utility itemset mining using multi-core processing. In: Proceedings of the International Conference on Computational Collective Intelligence pp. 467–479
21. Wang Y (2021) Algorithms for cross-level high utility itemset mining. Herbin Institute of Technology
22. Wu CW, Shie B-E, Yu PS, Tseng VS (2012) Mining top-k high utility itemsets. In: Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 78–86
23. Ryang H, Yun U (2015) Top-k high utility pattern mining with effective threshold raising strategies. Knowl-Based Syst 76:109–126
24. Tseng VS, Wu C-W, Fournier-Viger P, Yu PS (2015) Efficient algorithms for mining top-k high utility itemsets. IEEE Trans Knowl Data Eng 28(1):54–67
25. Duong Q-H, Liao B, Fournier-Viger P, Dam TL (2016) An efficient algorithm for mining the top-k high utility itemsets, using novel threshold raising and pruning strategies. Knowl-Based Syst 104:106–122
26. Singh K, Singh SS, Kumar A, Biswas B (2019) TKEH: an efficient algorithm for mining top-k high utility itemsets. Appl Intell 49:1078–1097
27. Krishnamoorthy S (2019) Mining top-k high utility itemsets with effective threshold raising strategies. Expert Syst Appl 117:148–165
28. Sun R, Han M, Zhang CY, Shen MY, Du SY (2021) Mining of top-k high utility itemsets with negative utility. J Intell Fuzzy Syst 40(3):5637–5652
29. Ashraf M, Abdelkader T, Rady S, Gharib TF (2022) TKN: an efficient approach for discovering top-k high utility itemsets with positive or negative profits. Inf Sci 587:654–678
30. Wu R, He Z (2018) Top-k high average-utility itemsets mining with effective pruning strategies. Appl Intell 48(10):3429–3445
31. AHMED C F, TANBEER S K, Jeong B S (2010) Efficient mining of high utility patterns over data streams with a sliding window method. In: Software engineering, artificial intelligence, networking and parallel/distributed computing. Springer, Berlin and Heidelberg, Berlin, pp. 99–113
32. Ryang H, Yun U (2016) High utility pattern mining over data streams with sliding window technique. Expert Syst Appl 57:214–231
33. Baek Y, Yun U, Kim H, Nam H, Kim H, Lin JC-W, Vo B, Pedrycz W (2021) Rhups: mining recent high utility patterns with sliding window–based arrival time control over data streams. ACM Trans Intell Syst Technol (TIST) 12(2):1–27
34. Jaysawal BP, Huang J-W (2020) SOHUPDS: a single-pass one-phase algorithm for mining high utility patterns over a data stream. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing pp. 490–497
35. Cheng H, Han M, Zhang N, Wang L, Li XJ (2021) ETKDS: an efficient algorithm of Top-K high utility itemsets mining over data streams under sliding window model. J Intell Fuzzy Syst 41(2):3317–3338
36. Yun U, Kim D, Yoon E, Fujita H (2018) Damped window based high average utility pattern mining over data streams. Knowl-Based Syst 144:188–205



Meng Han received her PhD degree from Beijing Jiaotong University. She is currently a full professor of school of computer science and engineering, North Minzu University. Her research interest is data mining.



Shujuan Liu received the MS degree with North Minzu University. Her main research interest is data mining.

Zhihui Gao received the MS degree from North Minzu University. Her main research interest is data mining.

Dongliang Mu received the MS degree from North Minzu University. His main research interests include data stream classification.

Ang Li received the MS degree from North Minzu University. His main research interests include data stream classification.