An efficient algorithm for mining the top-k high utility itemsets, using novel threshold raising and pruning strategies

Article *in* Knowledge-Based Systems · April 2016

DOI: 10.1016/j.knosys.2016.04.016

CITATIONS
56

READS
690

4 authors, including:

Philippe Fournier Viger
Shenzhen University
339 PUBLICATIONS **7,057** CITATIONS

SEE PROFILE

An efficient algorithm for mining the top-k high utility itemsets, using novel threshold raising and pruning strategies

Quang-Huy Duong[a], Bo Liao[a], Philippe Fournier-Viger[c], Thu-Lan Dam[a,b]

[a] *College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China*
[b] *Faculty of Information Technology, Hanoi University of Industry, Hanoi, Vietnam*
[c] *School of Natural Sciences and Humanities, Harbin Institute of Technology Shenzhen Graduate School, Shenzhen, Guangdong, 518055, China*

Abstract

Top-k high utility itemset mining is the process of discovering the k itemsets having the highest utilities in a transactional database. In recent years, several algorithms have been proposed for this task. However, it remains very expensive both in terms of runtime and memory consumption. The reason is that current algorithms often generate a huge amount of candidate itemsets and are unable to prune the search space effectively. In this paper, we address this issue by proposing a novel algorithm named kHMC to discover the top-k high utility itemsets more efficiently. Unlike several algorithms for top-k high utility itemset mining, kHMC discovers high utility itemsets using a single phase. Furthermore, it employs three strategies named RIU, CUD, and COV to raise its internal minimum utility threshold effectively, and thus reduce the search space. The COV strategy introduces a novel concept of *coverage*. The concept of coverage can be employed to prune the search space in high utility itemset mining, or to raise the threshold in top-k high utility itemset mining, as proposed in this paper. Furthermore, kHMC relies on a novel co-occurrence pruning technique named EUCPT to avoid performing costly join operations for calculating the utilities of itemsets. Moreover, a novel pruning strategy named TEP is proposed for reducing the search space. To evaluate the performance of the proposed algorithm, extensive experiments have been conducted on six datasets having various characteristics. Results show that the proposed algorithm outperforms the state-of-the-art TKO and REPT algorithms for top-k high utility itemset mining both in terms of memory consumption and runtime.

Keywords: High utility itemset mining, Top-k mining, Threshold raising strategies, Co-occurrence pruning, Transitive extension pruning, Coverage

1. Introduction

Frequent itemset mining (FIM) [1, 2, 3, 4] is a fundamental research topic in data mining. It consists of discovering the frequent itemsets appearing in a transactional database. A frequent itemset is a group of items having a support (occurrence frequency) that is no less than a user-specified minimum support threshold. Numerous algorithms have been developed to discover frequent itemsets efficiently [5, 6, 7, 8, 9, 10, 11, 12]. Most of them are based on two well-known representative algorithms: Apriori [1] and FP-Growth [4]. The Apriori algorithm employs a level-wise candidate generation-and-test approach. A drawback of Apriori-based algorithms is that they scan the database multiple times and can generate a huge amount of candidates. This greatly reduces the performance of Apriori-based algorithms. The FP-Growth algorithm relies on an efficient tree structure named FP-Tree, which is a compact representation of the database. To discover frequent itemsets, FP-Growth first builds an FP-Tree, and then uses a pattern-growth approach to discover the frequent itemsets directly from the FP-tree, without generating candidates. FP-Growth-based

Email addresses: huydqyb@gmail.com (Quang-Huy Duong), boliao@yeah.net (Bo Liao), philfv@hitsz.edu.cn (Philippe Fournier-Viger), lanfict@gmail.com (Thu-Lan Dam)

algorithms generally outperform Apriori-based algorithms. In FIM, the downward closure property [1, 4] is a well-known property, used to reduce the search space. This property states that if an itemset is infrequent, all its supersets are also infrequent. It is used by almost all FIM algorithms to prune the search space. FIM has a wide range of applications, and has been applied to various types of databases such as transactional databases [13], streaming databases [14], uncertain databases [15], and time-series databases [16]. Some representative applications of FIM are web analysis [17] and bioinformatics [18].

An important limitation of FIM is that in real-world applications, specifying the minimum support threshold required by FIM algorithms is not an easy task for users, since users often don't know what threshold values are the best for their requirements [19]. But choosing an appropriate threshold value is crucial since it affects the number of frequent itemsets found by FIM algorithms. If the threshold is set too high, few frequent itemsets will be found, and thus many interesting itemsets will be missed. But if the threshold is set too low, a huge number of frequent itemsets will be obtained, which is both time and space consuming, and users may find it difficult to analyze a large amount of frequent itemsets. To find an appropriate threshold value, a user may thus need to run a FIM algorithm several times. To solve this problem, a new research direction was proposed named top-k FIM [6, 20, 21, 22, 19, 23]. In top-k FIM, instead of specifying a minimum threshold value, the user must specify a parameter k indicating the number of itemsets to be found. A top-k FIM algorithm then returns the k itemsets having the highest supports.

A second important limitation of FIM is that it assumes that all items have the same importance (e.g. unit profit or weight) and that items may not appear more than once in each transaction. But these assumptions often do not hold in real applications. For example, items in a transactional database may have different unit profits, and items may have non binary purchase quantities in transactions. Besides, in real-life, retailers are often more interested in finding the itemsets that yield a high profit than those bought frequently. Thus, top-k FIM does not satisfy the requirement of users who want to discover itemsets having high utilities (e.g. generating a high profit).

To address these important issues, the problem of top-k FIM was recently redefined as the problem of top-k high utility itemset mining. Mining high utility itemsets (HUIs) [24, 25, 26, 27, 28, 29, 30] is an important research problem in data mining, which has a wide range of applications. It consists of discovering itemsets having utilities (e.g. profit) no less than a user-specified minimum utility threshold, that is high utility itemsets. The problem of top-k high utility itemset mining consists of finding the k itemsets having the highest utilities in a transactional database. Top-k high utility itemset mining is an important data mining task that is useful in many domains. For example, it can be used to find the k sets of products that are the most profitable when sold together, in retail stores. Top-k high utility itemset mining is harder than top-k FIM, since the downward closure property used in FIM to prune the search space, does not hold in high utility itemset mining. To circumvent this issue, overestimation methods [24, 28] such as the transaction weighted utility (TWU) have been proposed.

Although several algorithms have been designed for top-k high utility itemset mining, it remains a very expensive task in terms of runtime and memory consumption. It is thus an important research problem to design more efficient algorithms. Despite that top-k high utility itemset mining is inspired by the traditional problem of top-k FIM, the methods used in top-k FIM cannot be directly applied to solve the problem of top-k high utility itemset mining. To design an efficient top-k high utility itemset mining algorithm, additional issues need to be considered such as designing effective search space pruning techniques by considering information about the utilities of itemsets, and also how to raise the internal minimum utility threshold effectively to reduce the number of candidates.

To address the need for a more efficient top-k high utility itemset mining algorithm, this paper proposes a novel algorithm, called top-k High utility itemset Mining using Co-occurrence pruning (kHMC). It introduces several novel ideas to discover top-k high utility itemsets efficiently. The contributions of this paper are summarized as follows:

1. An efficient algorithm is proposed for mining the top-k high utility itemsets in transactional databases. The proposed algorithm relies on two efficient strategies for pruning the search space. The first strategy is an improved co-occurrence pruning strategy that eliminates a large number of join operations, and thus prunes a large part of the search space. This strategy is implemented using a novel co-occurrence

pruning structure with threshold (EUCST). The second strategy is named transitive extension pruning (TEP). It reduces the search space using a novel upper-bound on the utilities of itemsets.

2. The proposed algorithm relies on the utility-list structure [31]. To further improve the performance of discovering high utility itemsets using utility-lists, a low complexity utility-list construction procedure is proposed. Moreover, the proposed algorithm utilizes a strategy for abandoning utility-list construction early.
3. As previously mentioned, a key challenge in top-k high utility itemset mining is how to raise the internal minimum utility threshold effectively while searching for the top-k high utility itemsets, to reduce the search space. To address this challenge, the proposed algorithm utilizes several strategies, named RIU, COV, and CUD, for initializing and dynamically adjusting its internal minimum utility threshold. The COV strategy is based on a novel concept of coverage. The concept of coverage can be employed to prune the search space in high utility itemset mining, or to raise the threshold as introduced in this paper. This article proves that by using these strategies, no top-k HUIs will be missed, and demonstrates that these strategies are effective at raising the internal minimum utility threshold close to the optimal value.
4. Extensive experimental evaluations are conducted on both real and synthetic datasets to evaluate the proposed techniques. Results show that the proposed algorithm is faster and consumes less memory than the state-of-the-art TKU, REPT, and TKO algorithms for top-k high utility itemset mining.

The paper is organized as follows. Section 2 briefly reviews related work on (top-k) high utility itemset mining. Section 3 presents preliminaries and formally defines the problem of top-k high utility itemset mining. Section 4 proposes an improved strategy for pruning the search space based on item co-occurrence information. Section 5 presents two techniques for improving the efficiency of utility-list construction. Section 6 introduces a strategy for reducing the search space using a novel concept of transitive extension upper-bound utility. Section 7 proposes the concept of coverage. Section 8 presents the three threshold raising strategies used in the designed algorithm. Then, section 9 describes the proposed kHMC algorithm, to mine the top-k high utility itemsets, which incorporates all the techniques presented in previous sections. Section 10 presents an extensive experimental evaluation. Finally, section 11 draws the conclusion and discusses future work.

2. Related work

This section briefly reviews studies related to high utility itemset mining and top-k high utility itemset mining.

2.1. High utility itemset mining

Several algorithms have been proposed for high utility itemset mining. Two-Phase [26], an Apriori-based algorithm, employs an upper-bound called the Transaction Weighted Utilization (TWU) to restore the downward closure property for mining high utility itemsets. According to the TWU model, an itemset having a TWU lower than the minimum utility threshold is not a high utility itemset, as well as all its supersets. The TWU model allows to reduce the search space. However, a disadvantage of using the TWU model is that the TWU is a loose upper-bound on the utilities of itemsets and thus a huge amount of candidates still need to be considered to discover the high utility itemsets. The IIDS [30] algorithm improves upon Two-Phase by introducing a pruning strategy named Isolated Itemset Discarding Strategy (IIDS). But IIDS still finds HUIs by scanning the database numerous times using a level-wise approach. As a solution to this problem, the IHUP algorithm was proposed based on the FP-Growth algorithm and the TWU model [25]. The Two-Phase, IIDS, and IHUP algorithms, are said to be two-phase algorithms since they discover high utility itemsets in two phases. They first find a set of candidate HUIs in a given database (Phase I). Then, they scan the database to calculate the exact utility of each candidate and keep only those having utilities no less than the minimum utility threshold (Phase II). Despite having introduced new techniques to discover high utility itemsets, these algorithms still suffer from the problem of generating

too many candidates due to the use of the TWU model. A solution to the above issue was proposed in the UP-Growth algorithm [24]. This latter reduces the overestimation performed by the TWU model by using four strategies named DGU, DGN, DLU, and DLN. UP-Growth relies on a tree structure called UP-Tree, for mining high utility itemsets. A UP-Tree is constructed by performing two database scans. Then, candidates are generated directly from the tree. But the exact utility of each candidate is calculated by performing an additional database scan. UP-Growth+ [28] is an improved version of UP-Growth that introduces two strategies named DNU and DNN to further decrease the overestimations on utilities. BAHUI [32], PB [33], and MU-Growth [34] are recent two-phase algorithms which provide a slight improvement over Two-Phase or UP-Growth in terms of execution time. However, running these algorithms is still very time consuming.

Recently, algorithms have been proposed to mine HUIs in a single phase. HUI-Miner (High Utility Itemset Miner) was proposed by [31]. It introduces a new data structure called utility-list to maintain the information required for calculating the utilities of itemsets and prune the search space. Utility-lists are a vertical structure that is similar to the tid lists used by the Eclat algorithm [35] for mining frequent itemsets. Once HUI-Miner has constructed utility-lists for single items, it can create the utility-list of any itemset without scanning the database, and can derive all HUIs and their exact utilities from these utility-lists. The search space of itemsets is explored by HUI-Miner using a depth-first search rather than a breadth-first search. Furthermore, it avoids repeatedly scanning the database, thanks to the use of utility-lists. Experimental results have shown that HUI-Miner outperforms IHUP [25], UP-Growth [24], and UP-Growth+ [28]. But HUI-Miner still has to perform a costly join operation to construct the utility-list of each itemset considered by its search procedure. To reduce the number of join operations, the FHM [36] algorithm was designed. It integrates a strategy for pruning the search space using information about itemset co-occurrences. It was shown to reduce the number of join operations by up to 95%.

2.2. Top-k high utility itemset mining

An important limitation of high utility itemset mining is that in real applications, specifying the minimum utility threshold is not an easy task for users, as they often don't know what threshold value is the best for their requirements [37, 38, 39]. But choosing an appropriate threshold value is crucial since it directly influences the number of high utility itemsets found by the algorithms. If the threshold is set too high, few itemsets will be found, and thus many interesting itemsets will be missed. But if the threshold is set too low, a huge number of itemsets will be obtained, which is time and space consuming, and users may find it difficult to analyze a large amount of high utility itemsets. To address this issue, the task of top-k high utility itemset mining was proposed, by combining the concept of top-k pattern mining from FIM with high utility itemset mining. In top-k high utility itemset mining, the user must specify a parameter k, representing the number of patterns to be found, instead of specifying a minimum utility threshold. The task of top-k high utility itemset mining is useful in many domains. For example, it can be used to find the k sets of products that are the most profitable when sold together, in retail stores.

Several top-k high utility-itemset mining algorithms have been proposed [37, 38, 39]. They typically follow a same general process. A top-k high utility itemset mining algorithm initially sets an internal minimum utility threshold to zero. Then, during the mining process, the algorithm automatically raises the threshold using various strategies, to reduce the search space. But it is important that the algorithm uses appropriate strategies to raise the threshold, to ensure that no top-k high utility itemsets are missed. A top-k high utility itemset mining algorithm searches for itemsets by using a search strategy. When a potential itemset is found, it is inserted into a list of itemsets R_k, ordered by their utilities. The minimum utility threshold is then raised to the k-*th* largest utility value in R_k. Then, itemset(s) in R_k not respecting the minimum utility threshold anymore are removed from R_k. Thereafter, the new threshold value is used to continue the search for more itemsets (by searching for itemsets having utilities no less than the threshold). The algorithm then continues searching for more itemsets until no itemsets are produced by the search strategy. When the algorithm terminates, it has found the top-k high utility itemsets. The final value of the internal minimum utility threshold is called the *boundary threshold value*.

The efficiency of a top-k high utility itemset mining algorithm is not only determined by its data structure(s) and search strategy. But it also largely depends on how fast the algorithm can raise its internal minimum utility threshold to prune the search space. The major challenges in top-k HUI mining are thus

to design effective strategies for initializing and raising the internal minimum utility threshold, and effective data structures and strategies for reducing the search space.

Several algorithms [40, 37, 41] have been proposed for top-k high utility itemset mining. The first one is TKU [38], which extends the UP-Growth algorithm [24, 28] with several efficient threshold raising strategies for top-k high utility itemset mining. Recently, another algorithm, called *Raising threshold with Exact and Pre-calculated utilities* (REPT) [37] was proposed for top-k high utility itemset mining. REPT relies on the UP-Tree [24, 28] structure, just like TKU. However, REPT uses different threshold raising strategies. In Phase I, two strategies called *Pre-evaluation with Utility Descending order* (PUD) and *Real Item Utilities* (RIU) are used. In Phase 2, two strategies named *Raising by Support Descending order* (RSD) and *Sorting candidates and raising the threshold by Exact and Pre-calculated utilities of candidates* (SEP) are applied. A drawback of REPT is that besides parameter k, users also have to set a value for a novel parameter N, used by the RSD strategy. Choosing a value for N that provides good performance is difficult for users [39]. The strategies proposed in TKU and REPT can raise their internal minimum utility thresholds effectively. But these algorithms still generate a huge amount of candidates because they are two-phase algorithms. They thus repeatedly scan the database to obtain the exact utilities of candidates and identify the actual top-k high utility itemsets. This process is very expensive, especially when working with a huge number of candidates or datasets with long transactions and many items. Recently, a one-phase algorithm named TKO was proposed [39]. It was shown to generally outperform TKU and REPT.

This paper proposes a novel algorithm named kHMC to mine the top-k high utility itemsets efficiently. kHMC is a top-k high utility itemset mining algorithm that uses the utility-list structure [31] to extract the top-k high utility itemsets. kHMC is a one-phase algorithm, unlike TKU and REPT. The kHMC algorithm prunes the search space using two novel strategies called *Estimated Utility Co-occurrence Pruning Strategy with Threshold* (EUCPT) and *Transitive Extension Pruning strategy* (TEP). The top-k high utility itemsets are discovered by raising an internal minimum utility threshold, initially set to zero. The key differences between the proposed algorithm and previous algorithms are the following. First, the proposed EUCPT pruning strategy utilizes information about item co-occurrences to prune the search space. Second, the TEP strategy prunes the search space using a novel upper-bound. Third, kHMC also introduces a novel utility-list construction procedure, having a complexity lower than the one used in HUI-Miner and FHM. Fourth, an early abandoning strategy (EA strategy) is used during utility-list construction to stop building a utility-list if it cannot result in a top-k high utility itemset. These novelties considerably reduce runtime and memory consumption for discovering the top-k high utility itemsets. Besides, the kHMC algorithm also relies on three threshold raising strategies. The first one is named *raising the threshold by Real Item Utilities* (RIU), and is applied during the first database scan. The second and third ones are named *raising the threshold based on Co-occurrence with Utility Descending order* (CUD) and *raising the threshold based on COVerage with utility descending order* (COV), and are applied during the second database scan. These strategies are very efficient at raising the internal minimum utility threshold close to the boundary threshold value. Furthermore, the paper introduces a novel concept of coverage. This latter can be employed to prune the search space in high utility itemset mining, or to raise an internal minimum utility threshold in top-k high utility itemset mining as proposed in this paper.

3. Preliminaries and Problem Definition

This section presents preliminaries related to high utility itemset mining, and defines the problem of top-k high utility itemsets mining.

Let I=$\{i_1; i_2; \ldots ; i_m\}$ be a set of items, such that each item $i_j \in I$ is associated with a positive number $p(i_j)$, called its *external utility*, which represents the relative importance (e.g. unit profit) of item i_j to the user. An itemset X is a set of l distinct items $\{i_1; i_2; \ldots ; i_l\}$ such that $X \subseteq I$. An itemset containing l items is said to be of length l, and to be a l-itemset. In the rest of this paper, for the sake of brevity, each itemset will be denoted by the concatenation of its items. For example, the itemset $\{x; y; z\}$ will be denoted as xyz. Similarly, the union of two itemsets X and Y will be denoted as XY or $X \cup Y$. A transaction T_d is a subset of I and has a unique identifier d, called its transaction identifier, or Tid. A transaction database D = $\{T_1; T_2; \ldots ; T_n\}$ is a set of transactions. Each item i_j in a transaction T_d $(1 \leq d \leq n)$ is associated with

a quantity $q(i, T_d)$ called the *local utility* (e.g. purchase quantity) of item i_j in T_d. For example, Figure 1 shows a transaction database D containing five transactions (T_1, T_2, T_3, T_4, and T_5), which will be used as running example. In this database, the set of items I in D is $\{a; b; c; d; e; f; g\}$. The external utilities of these items are respectively 5, 2, 1, 2, 3, 1, and 1. The itemset ac appears in transactions T_1, T_2, and T_3. The items a, c, e, and g, respectively have internal utilities (e.g. purchase quantities) of 2, 6, 2, and 5, in transaction T_2.

Definition 1. The utility of an item i in a transaction T_d is defined as $u(i, T_d) = q(i, T_d) \times p(i)$, where $q(i, T_d)$ is the internal utility (purchase quantity) of item i in transaction T_d, and $p(i)$ is the external utility (unit profit) of item i. For example, in Figure 1, $u(a, T_1) = 1 \times 5 = 5$, $u(c, T_1) = 1 \times 1 = 1$.

Definition 2. The utility of an itemset X in a transaction T_d is denoted as $u(X, T_d)$ and defined as $u(X, T_d) = \sum_{i \in X} u(i, T_d)$. For example, in Figure 1, $u(ac, T_1) = 1 \times 5 + 1 \times 1 = 6$, and $u(ac, T_2) = 2 \times 5 + 6 \times 1 = 16$.

Definition 3. The utility of an itemset X in a database D is defined as $u(X) = \sum_{X \subseteq T_d \wedge T_d \in D} u(X, T_d)$. For example, in Figure 1, $u(ac) = u(ac, T_1) + u(ac, T_2) + u(ac, T_3) = 6 + 16 + 6 = 28$.

Definition 4. The utility of a transaction T_d is denoted as $TU(T_d)$, and is calculated as $u(T_d, T_d)$. For example, in Figure 1, $TU(T_1) = 8$, $TU(T_2) = 27$.

Definition 5. Let $\succ$ be any total order on items from I, and X be an itemset. An *extension* of X is an itemset Xy that is obtained by appending an item y to X, such that $y \succ i, \forall i \in X$. For example, in Figure 1, assume that items in transactions are ordered by the lexicographical order. The itemset acd is an extension of the itemset ac.

Given a user-specified minimum utility threshold *min_util*, if the utility of an itemset X is no less than *min_util*, then X is said to be a high utility itemset. Otherwise, X is said to be a low utility itemset. The problem of high utility itemset mining is to discover all high utility itemsets. The problem of top-k high utility itemset mining is to discover the k itemsets having the highest utilities, where k is a parameter set by the user. In FIM, the downward closure property is used to prune the search space. But in high utility itemset mining, the utility measure does not follow this property (the utility is neither monotonic nor anti-monotonic). To restore the downward closure property in utility mining, the transaction-weighted downward closure property (TWDC) was proposed [26].

Definition 6. The transaction-weighted utilization (TWU) of an itemset X in a database D is denoted as $TWU(X)$ and defined as $TWU(X) = \sum_{T_d \in D \wedge X \subseteq T_d} TU(T_d)$.

Property 1 (Overestimation [26])**.** The TWU of an itemset X is no less than its utility, that is $TWU(X) \geq u(X)$.

Property 2 (Transaction-weighted downward closure property [26])**.** Let there be two itemsets X and Y. If $Y \subseteq X$, then $TWU(Y) \geq TWU(X)$. Thus, the TWU measure is downward close (anti-monotonic). Most HUI mining algorithms use the TWU values of itemsets as an overestimation on their utilities and the utilities of their supersets, to prune the search space.

Example 1. Consider the database of Figure 1. The utility of itemset c in transaction T_1 is $u(c, T_1) = 1 \times 1 = 1$. The utility of itemset cd in T_1 is $u(cd, T_1) = u(c, T_1) + u(d, T_1) = 1 \times 1 + 1 \times 2 = 1 + 2 = 3$. Because the itemset cd appears in transactions T_1, T_3, and T_4, its utility is calculated as $u(cd) = u(c) + u(d) = u(c, T_1) + u(c, T_3) + u(c, T_4) + u(d, T_1) + u(d, T_3) + u(d, T_4) = 1 + 1 + 3 + 2 + 12 + 6 = 25$. The utility of transaction T_1 is $TU(T_1) = u(a, T_1) + u(c, T_1) + u(d, T_1) = 5 + 1 + 2 = 8$. The transaction-weighted utilization of itemset f is $TWU(f) = TU(T_3) = 5 + 4 + 1 + 12 + 3 + 5 = 30$. Figure 2 shows the TU values of all transactions in D, and the TWU values of all items. Suppose that the parameter k is set to 3. The set of top-k high utility itemsets for the running example is {*bcde*:40, *bde*:36, *bcd*:34}, where the number beside each itemset indicates its utility. For $k = 3$, the boundary threshold value is 34.

TID	Transaction	TU
T_1	(a,1), (c,1), (d,1)	8
T_2	(a,2), (c,6), (e,2), (g,5)	27
T_3	(a,1), (b,2), (c,1), (d,6),(e,1),(f,5)	30
T_4	(b,4), (c,3), (d,3), (e,1)	20
T_5	(b,2), (c,2), (e,1), (g,2)	11

Item	a	b	c	d	e	f	g
External utility	5	2	1	2	3	1	1

Figure 1: An example transaction database (top) and the corresponding external utilities of items (bottom).

Item Name	a	b	c	d	e	f	g
TWU	65	61	96	58	88	30	38

TID	1	2	3	4	5
TU	8	27	30	20	11

Figure 2: The transaction utilities and TWU values for the database of Figure 1.

4. Effective Search Space Pruning using item Co-occurrence Information

A key challenge in HUI mining is to design effective search space pruning techniques to discover HUIs efficiently. This section proposes an improved search space pruning strategy for top-k high utility itemset mining, which relies on information about item co-occurrences to reduce the search space. This proposed strategy is named Estimated Utility Co-occurrence Pruning Strategy with Threshold (EUCPT). It is an improvement of the EUCP strategy introduced in the FHM algorithm [36]. The following paragraphs first describe the EUCP strategy, and then present the proposed EUCPT strategy.

4.1. The Estimated Utility Co-occurrence Pruning Strategy (EUCP)

The EUCP [36] pruning strategy relies on a structure called the Estimated Utility Co-occurrence Structure (EUCS), which is defined as follows:

Definition 7. The Estimated Utility Co-occurrence Structure of a database D is denoted as $EUCS_D$ and defined as $EUCS_D = \{(a; b; TWU(ab)) \in I^* \times I^* \times \mathbb{R}^+\}$, where I^* is the set of all items having a TWU no less than *min_util* in D.

The EUCS has been defined for the problem of HUI mining, where the user has to specify a *min_util* threshold. But in top-k HUI mining, no such threshold is defined by the user. Instead, top-k HUI mining algorithms rely on an internal minimal utility threshold that is initially set to zero, and increased during the search for itemsets using threshold raising strategies, as it will be explained in the next section. As other efficient algorithms for HUI mining, the proposed algorithm only scans the database twice to compute the TU and TWU values of all items and 2-itemsets, to construct the EUCS structure. The first database scan calculates the TWU of each single item, as well as the utility of each single item. The TWU values of items are used to establish a total order on items appearing in the transactional database, which is the order of ascending TWU values. The TWU values of items will be used by the threshold raising strategies presented in section 8. The second database scan constructs the EUCS. During this scan, items in transaction are reordered using the ascending order of TWU values. For instance, consider the database D of Figure 1. Items are reordered by ascending order of TWU values, and the EUCS is implemented as a triangular matrix, as shown in Figure 3.

The EUCP strategy employs the following property to reduce the search space.

Item	f	g	d	b	a	e	c
TWU	30	38	58	61	65	88	96

TID	**Ordered Transaction**
T_1	(d,1), (a,1), (c,1)
T_2	(g,5), (a,2), (e,2), (c,6)
T_3	(f,5), (d,6), (b,2), (a,1), (e,1), (c,1)
T_4	(d,3), (b,4), (e,1), (c,3)
T_5	(g,2), (b,2), (e,1), (c,2)

Item	f	g	d	b	a	e
g	0					
d	30	0				
b	30	11	50			
a	30	27	38	30		
e	30	38	50	61	57	
c	30	38	58	61	65	88

Figure 3: Items are reordered by ascending order of TWU values (top). Transactions are sorted according to this order (bottom-left). The EUCS matrix (bottom-right).

Property 3 (Pruning)**.** Let there be an itemset X. If $TWU(X) < min\_util$, then for any extension Y of X, $u(XY) < min\_util$.

Proof. By Property 2, $TWU(XY) \leq TWU(X)$. Because $TWU(X) < min\_util$, it follows that $TWU(XY) < min\_util$. Furthermore, $u(XY) \leq TWU(XY)$ by Property 1. Therefore $u(XY) < min\_util$. □

Property 3 is employed by the EUCP strategy to directly prune an itemset Pxy and its transitive extensions. If there is a tuple $(x; y; TWU(xy))$ in the EUCS such that $TWU(xy) < min\_util$, then any supersets of xy such as Pxy, and its transitive extensions, have utilities lower than min_util. Thus, Pxy and its transitive extensions do not need to be explored by the search process.

The EUCP strategy is efficient because it can reduce the number of join operations performed by utility-list based algorithms, and thus the number of candidates that they generate. It was shown that the number of candidates generated can be pruned by up to 95% using this strategy [36]. Hence, a considerable performance gain can be achieved using this strategy.

4.2. The Estimated Utility Co-occurrence Pruning Strategy with Threshold (EUCPT)

A simple optimization of the EUCS was mentioned in the authors' implementation. By implementing the EUCS using hashmaps instead of a triangular matrix, the EUCS only stores tuples such that TWU $\neq 0$. As a result, fewer items are kept in memory, and this implementation is thus more memory efficient than the original implementation. This paper introduces an improvement of the EUCS, named EUCST, which keeps fewer items in memory than the EUCS. The design of the EUCST is based on the observation that only tuples having TWU values greater than min_util are used by the EUCP strategy to prune the search space. Thus, the EUCST only stores tuples for itemsets that have a TWU greater than min_util. As a result, less tuples are stored in the proposed structure, and it is more memory efficient. Moreover, because the EUCST is smaller, the time spent for searching the structure, when using the structure to prune join operations, is also reduced. The proposed pruning strategy, called EUCPT, utilizes the designed EUCST structure to avoid join operations.

For example, consider that min_util is set to 31. Using the proposed EUCST, all tuples having a TWU less than 31 will be eliminated, as shown in Figure 4(a). In this figure, all tuples colored in brown have a TWU that is less than 31, and are thus removed. The resulting EUCST keeps less tuples than the corresponding EUCS, as shown in Figure 4(b).

The construction of the EUCST is performed during the second database scan. For each transaction T_d, each pair of items in T_d is considered. If there is no tuples for that pair of items in the EUCST, a tuple is inserted in the EUCST, where the TWU value of the pair of items is set to the transaction utility of T_d, that is $TU(T_d)$. Otherwise, the TWU value of the tuple in the EUCST is increased by $TU(T_d)$.

For example, consider the database of Figure 1. At first, the transaction T_1 is processed, and it is found that $TU(T_1) = 8$. Items in that transaction are d, a, and c. The algorithm combines each item with each other to create the tuples $\{(d, a, 8), (d, c, 8), (a, c, 8)\}$, which are inserted in the EUCST (see Figure 5 - left). The transaction T_2 is then processed, and it is found that $TU(T_2) = 27$. T_2 is composed of four items

Item	f	g	d	b	a	e
g	0					
d	30	0				
b	30	11	50			
a	30	27	38	30		
e	30	38	50	61	57	
c	30	38	58	61	65	88

Item	g	d	b	a	e
g					
d					
b		50			
a		38			
e	38	50	61	57	
c	38	58	61	65	88

Figure 4: The (a) EUCS and (b) EUCST structures

g, a, e, and c. The tuples that are constructed using these four items are: $\{(g, a, 27)$, $(g, e, 27)$, $(g, c, 27)$, $(a, e, 27)$, $(a, c, 27)$, $(e, c, 27)\}$. Because the pair (a, c) already exists in the EUCST with a TWU value of 8, this value is updated to $8 + 27 = 35$. The others pairs do not exist in the EUCST. They are thus inserted into the EUCST without updating any tuples. The resulting EUCST after processing T_1 and T_2 is shown in Figure 5 - right. The same process is repeated for the next transactions. The result is the EUCS presented in Figure 4(a). Then, all tuples having a TWU less than *min_util* (colored in brown in Figure 4(a)) are removed from the EUCS, and the EUCST shown in Figure 4(b) is obtained.

Item	f	g	d	b	a	e
g						
d						
b						
a			8			
e						
c			8		8	

Item	f	g	d	b	a	e
g						
d						
b						
a		27	8			
e		27			27	
c		27	8		35	27

Figure 5: The construction process of the EUCST

The EUCST is constructed quickly and occupies a small amount of memory. It is thus an efficient data structure for mining high utility itemsets.

5. Efficient Utility-list Construction

In utility-list based algorithms such as HUI-Miner, FHM, and the proposed algorithm, a key operation is the construction of utility-lists. The next subsection thus proposes an improved utility-list construction procedure that has a lower complexity than the one used in previous algorithms. Moreover, the following subsection presents an early abandoning strategy to avoid completely constructing utility-lists when specific conditions are met. These optimizations greatly reduce the runtime and memory consumption of the proposed algorithm.

5.1. An Efficient Utility-lists Construction Procedure

The proposed kHMC algorithm utilizes the utility-list structure proposed in the HUI-Miner algorithm [31] to mine high utility itemsets directly, using a single phase. The utility-list structure is used to maintain information about the utilities of itemsets. The utility of an itemset can be quickly calculated by joining utility-lists of smaller itemsets. Utility-lists are briefly introduced thereafter by the following definitions and properties:

Definition 8 (Utility-list)**.** Let $\prec$ be any total order on items from I. The utility-list of an itemset X in a database is denoted as $UL(X)$, and contains a set of tuples having the form $(tid; iutil; rutil)$, such that $UL(X) = \bigcup_{X \in T_{tid}} (tid; u(X, T_{tid}), \sum_{i \in X \wedge x \prec i} u(i, T_{tid}))$.

Definition 9 (Remaining utility [31]). The set of items appearing after an itemset X in a transaction T is defined as $\{i|i \in T \wedge \nexists j \in X, i \prec j\}$. The remaining utility of an itemset X in a transaction T is denoted as $RU(X,T)$ and defined as the sum of the utilities of all items appearing after X in T.

Property 4 (Sum of iutil values). Let there be an itemset X. X is a high utility itemset if $\sum_{ul \in UL(X)} ul.iutil \geq min\_util$. Otherwise, it is a low utility itemset.

Property 5 (Sum of iutil and rutil values [31]). Let X be an itemset. If the sum of *iutil* and *rutil* values in the utility-list of X is less than *min_util*, all extensions of X and their transitive extensions are low utility itemsets.

As proposed in the HUI-Miner algorithm, the utility-list of an itemset can be constructed by intersecting the utility-lists of some of its subsets. For example, let P, Px, and Py be itemsets, such that Px and Py are extensions of P with items x and y, respectively. The utility-list of Pxy is obtained by applying Algorithm 1. This procedure is the same in FHM [36] and HUI-Miner [31]. The implementation of this procedure in the FHM algorithm can be obtained as part of the SPMF data mining library [42]. The main idea of this procedure is the following. Two utility-lists $UL(x)$ and $UL(y)$ are intersected as follows. For each element in $UL(x)$, a binary search is performed to check whether the element exists in $UL(y)$ or not. Therefore, the complexity of the procedure is $O(mlogn)$, where m and n are $|UL(x)|$ and $|UL(y)|$, respectively. Because all Tids in a utility-list are ordered, the algorithm for identifying transactions that are common to both utility-lists by comparing the Tids in the two utility-lists can be improved. This paper introduces an intersection procedure for constructing utility-lists having a complexity of $O(m+n) < O(mlogn)$. The pseudo-code is presented in Algorithm 2. Note that if $UL(P)$ is not empty then the Tid list in $UL(Pxy)$ is always a subset of the Tid list in $UL(P)$. In this case, the complexity of the intersection procedure is only $O(|UL(P)|)$.

Algorithm 1 Construct utility-list

Input:
$UL(P)$, the utility-list of itemset P;
$UL(Px)$, the utility-list of itemset Px;
$UL(Py)$, the utility-list of itemset Py
Output: $UL(Pxy)$, the utility-list of itemset Pxy.

1: $UL(Pxy)$ = NULL;
2: **for each** (tuple $ex \in UL(Px)$) **do**
3: **if** ($\exists ey \in UL(Py)$ and $ex.tid = ey.tid$) **then**
4: **if** ($UL(P)$ is not empty) **then**
5: Search element $e \in UL(P)$ such that $e.tid = ex.tid$;
6: $exy \longleftarrow$ *(ex.tid; ex.iutil + ey.iutil - e.iutil; ey.rutil)*;
7: **else**
8: $exy \longleftarrow$ *(ex.tid; ex.iutil + ey.iutil; ey.rutil)*;
9: **end if**
10: $UL(Pxy) \longleftarrow UL(Pxy) \cup exy$;
11: **end if**
12: **end for**
13: **return** $UL(Pxy)$;

5.2. A Strategy for Abandoning Utility-list Construction Early

Furthermore, an additional strategy called EA (Early Abandoning) is integrated in the proposed kHMC algorithm, to obtain better performance for utility-list construction. This strategy consists of abandoning the construction of a utility-list if a specific condition is met.

The kHMC algorithm, just like HUI-Miner, utilizes Property 4 to determine if an itemset X is a high utility itemset (an itemset X is a high utility itemset if $\sum_{ul \in UL(X)} ul.iutil \geq min\_util$). Moreover, kHMC utilizes the following property [43].

Algorithm 2 iConstruct Algorithm

Input:
$UL(P)$, the utility-list of itemset P;
$UL(Px)$, the utility-list of itemset Px;
$UL(Py)$, the utility-list of itemset Py

Output: $UL(Pxy)$, the utility-list of itemset Pxy.

1: $UL(Pxy)$ = NULL; Let i, j = 0;
2: Let total = $UL(Px)$.sumIutils + $UL(Py)$.sumIutils + $UL(Px)$.sumRutils + $UL(Py)$.sumRutils; *//for the EA strategy*
3: **while** (i < $UL(Px).size$ and j < $UL(Py).size$) **do**
4: **if** ($UL(Px)$[i].tid = $UL(Py)$[j].tid) **then**
5: $UL(Pxy) \longleftarrow$ ($UL(Px)$[i].tid; $UL(Px)$[i].iutil + $UL(Py)$[j].iutil; $UL(Py)$[j].rutil);
6: i = i + 1; j = j + 1;
7: **else if** ($UL(Px)$[i].tid < $UL(Py)$[j].tid) **then**
8: total = total - ($UL(Px)$[i].iutil + $UL(Px)$[i].rutil);
9: i = i + 1;
10: **else**
11: total = total - ($UL(Py)$[j].iutil + $UL(Py)$[j].rutil);
12: j = j + 1;
13: **end if**
14: **if** (total < *min_util*) **then**
15: **return** null; *//EA strategy*
16: **end if**
17: **end while**
18: **if** ($UL(P) \neq \varnothing$ and $UL(Pxy) \neq \varnothing$) **then**
19: Let i = j = 0;
20: **while** (i < $UL(Pxy).size$ and j < $UL(P).size$) **do**
21: **if** ($UL(Pxy)$[i].tid = $UL(P)$[j].tid) **then**
22: $UL(Pxy)$[i++].iutil -= $UL(P)$[j].iutil;*//Subtract utilities of tuples existing in prefix* $UL(P)$
23: **end if**
24: j = j + 1;
25: **end while**
26: **end if**
27: **return** $UL(Pxy)$;

Property 6 (LA property)**.** Given two itemsets X and Y, if the value $\sum_{\forall T_i \in D} U(X,T_i) + RU(X,T_i) - \sum_{\forall T_j \in D, X \subseteq T_j \wedge Y \nsubseteq T_j} U(X,T_j) + RU(X,T_j) < min\_util$, then $\forall X' \supseteq X \wedge Y' \supseteq Y, X'Y' \notin HUIs$.

Proof. The proof of this property can be found in [43]. For the convenience of the reader, this proof is provided thereafter. We have:

$$min\_util > \sum_{\forall T_i \in D} U(X,T_i) + RU(X,T_i) - \sum_{\forall T_j \in D, X \subseteq T_j \wedge Y \nsubseteq T_j} U(X,T_j) + RU(X,T_j) =$$
$$\sum_{\forall T_i \in D, X \subseteq T_i \wedge Y \subseteq T_i} U(X,T_i) + RU(X,T_i) + \sum_{\forall T_i \in D, X \subseteq T_i \wedge Y \nsubseteq T_i} U(X,T_i) + RU(X,T_i)$$
$$- \sum_{\forall T_j \in D, X \subseteq T_j \wedge Y \nsubseteq T_j} U(X,T_j) + RU(X,T_j) =$$
$$\sum_{\forall T_i \in D, X \subseteq T_i \wedge Y \subseteq T_i} U(X,T_i) + RU(X,T_i)$$

On the other hand, we also have:

$$
\begin{aligned}
& U(X'Y') = \sum_{X'Y' \subseteq T_i \in D} U(X'Y', T_i) \leq \sum_{X'Y' \subseteq T_i \in D} U(XY, T_i) + RU(XY, T_i) \\
\Rightarrow \quad & U(X'Y') \leq \sum_{\forall T_i \in D, X \subseteq T_i \wedge Y \subseteq T_i} U(XY, T_i) + RU(XY, T_i) \\
\Rightarrow \quad & U(X'Y') \leq \sum_{\forall T_i \in D, X \subseteq T_i \wedge Y \subseteq T_i} U(X, T_i) + RU(X, T_i) < min\_util
\end{aligned}
$$

Therefore, $X'Y' \notin HUIs$. □

Thus, for any itemset X, extensions of X will be explored if (1) the pruning condition of Property 5 is satisfied ($\sum_{ul \in UL(X)}(ul.iutil + ul.rutil) \geq min\_util$), and (2) Property 6 is not satisfied. The proposed algorithm adopts the LA property in its EA strategy. The EA strategy is applied as follows. In the utility-list construction process, if the condition of Property 6 is satisfied, it is concluded that extensions of X should not be explored, as they will be low utility itemsets. Thus, the utility-list construction process is immediately stopped. This saves the time and memory that would be used for creating and storing the utility-list. This process is detailed in Algorithm 2. At first, the variable *total* is set to the sum of *iutil* and *rutil* values in the utility-lists of Px and Py (Line 2). Then, the value of *total* is decreased for each tuple in $UL(Px)$ that has no corresponding tuple in $UL(Py)$, or vice versa, by the sum of *iutil* and *rutil* values in the current tuple (Line 8, 11). Lines (14-16) check if *total* falls below *min_util*. If yes, the construction of the utility-list of Pxy is stopped and extensions of Pxy will not be explored. This stopping criterion is employed by the proposed algorithm during the construction of all utility-lists for itemsets of size greater than one. Thus, this strategy reduces the runtime and memory consumption of the proposed algorithm.

6. Transitive Extension Pruning Strategy (TEP)

Liu et al. [31] proposed a pruning strategy based on Property 5 to reduce the search space using the sum of *iutil* and *rutil* values in the utility-list of an itemset Y. If this sum is less than *min_util*, then the itemset Y and its transitive extensions are low utility itemsets. This strategy is efficient at pruning the search space and is thus integrated in the proposed kHMC algorithm. However, a drawback of this pruning strategy is that it can only be applied to an itemset Y after the utility-list of Y has been fully constructed. But constructing a utility-list is an expensive operation. It is thus desirable to design pruning strategies that can prune an itemset Y and its transitive extensions before the utility-list of Y is constructed. The designed EUCPT strategy, presented in Section 4, is one such strategy. It relies on information about item co-occurrences to reduce the search space. In this section, we propose a novel strategy for pruning an itemset Y and its transitive extensions before its utility-list is constructed, by adapting Property 5. This strategy is called Transitive Extension Pruning (TEP).

Recall that kHMC mines high utility itemsets using a depth-first search. For any itemset X, kHMC generates larger itemsets by recursively appending items one at a time to X according to the $\succ$ order. Consider an itemset Y that is an extension of an itemset X, formed by appending an item y to X, that is $Y = Xy$. The TEP strategy is used to prune the itemset Y and its transitive extensions, when the search procedure is considering itemset X (thus before the utility-list of Y is constructed). This novel strategy introduced in kHMC relies on a new upper-bound on the utility of itemsets and their extensions, named the transitive extension upper-bound utility. This upper-bound is defined as follows.

Definition 10. (Transitive extension upper-bound utility) Let be an itemset X and an item $y \succ i, \forall i \in X$. The transitive extension upper-bound utility of itemset X with respect to item y is denoted as $teu(X, y)$ and is defined as $teu(X, y) = \sum_{ul \in UL(X)} ul.iutil + \sum_{ul \in UL(y)} ul.rutil + u(y)$.

This upper-bound can be calculated very efficiently. In the proposed kHMC algorithm, utility-lists of single items are created during the initial database scans. Thus, the second term $\sum_{ul \in UL(y)} ul.rutil$ and the

third term $u(y)$ are precalculated in advance for each item y. The proposed transitive extension upper-bound utility is used for pruning the search space based on the following properties and lemma.

Property 7. (Overestimation using the transitive extension upper-bound utility) Let be an itemset X and an item $y \succ i, \forall i \in X$. The relationship $teu(X, y) \geq u(Xy)$ holds.

Property 8. Let there be an itemset X and an item y, such that $y \succ i, \forall i \in X$. The value $teu(X, y)$ is no less than the sum of *iutil* and *rutil* values in the utility-list of Xy, i.e., $teu(X, y) \geq \sum_{ul \in UL(Xy)}(ul.iutil + ul.rutil)$.

Proof. We have:

$$
\begin{aligned}
teu(X,y) &= \sum_{ul \in UL(X)} ul.iutil + \sum_{ul \in UL(y)} ul.rutil + u(y) \\
&= \sum_{X \subseteq T_d \wedge T_d \in D} u(X, T_d) + \sum_{y \in T_d \wedge T_d \in D} RU(y, T_d) + u(y) \\
&= \sum_{X \subseteq T_d \wedge y \in T_d \wedge T_d \in D} u(X, T_d) + \sum_{X \subseteq T_d \wedge y \notin T_d \wedge T_d \in D} u(X, T_d) + \sum_{y \in T_d \wedge T_d \in D} RU(y, T_d) + \sum_{y \in T_d \wedge T_d \in D} u(y, T_d) \\
&= \sum_{X \subseteq T_d \wedge y \in T_d \wedge T_d \in D} [u(X, T_d) + u(y, T_d)] + \sum_{X \subseteq T_d \wedge y \notin T_d \wedge T_d \in D} u(X, T_d) + \sum_{y \in T_d \wedge T_d \in D} RU(y, T_d) \\
&\quad + \sum_{y \in T_d \wedge X \nsubseteq T_d \wedge T_d \in D} u(y, T_d) \\
&\geq \sum_{Xy \subseteq T_d \wedge T_d \in D} u(Xy, T_d) + \sum_{y \in T_d \wedge T_d \in D} RU(y, T_d) \\
&\geq \sum_{Xy \subseteq T_d \wedge T_d \in D} u(Xy, T_d) + \sum_{Xy \subseteq T_d \wedge T_d \in D} RU(y, T_d) = \sum_{ul \in UL(Xy)} (ul.iutil + ul.rutil)
\end{aligned}
$$

□

Lemma 1. (Pruning the search space using the transitive extension upper-bound utility) Let there be an itemset X and an item $y \succ i, \forall i \in X$. If $teu(X, y) < min\_util$, then the single item extension Xy and its transitive extensions are low utility.

Proof. Since $teu(X, y) \geq \sum_{ul \in UL(Xy)}(ul.iutil + ul.rutil)$ (by Property 8), and $teu(X, y) < min\_util$, it follows that $\sum_{ul \in UL(Xy)}(ul.iutil + ul.rutil) < min\_util$. Therefore, by Property 5, the itemset Xy and its transitive extensions are low utility. □

The above lemma is used by the TEP strategy to prune the search space. For a given itemset X, if $teu(X, y)$ is less than *min_util*, then Xy and all its transitive extensions are low utility itemsets, and can thus be pruned. The TEP strategy can greatly reduce the number of extensions to be explored for mining high utility itemsets using utility-lists. As it will be shown in the experimental evaluation of this paper, if the proposed kHMC algorithm applies the TEP strategy after applying the EUCPT strategy, kHMC can prune up to 21% more candidates from the search space.

7. The Concept of Coverage

This section introduces a novel concept called coverage, for mining high utility itemsets. In this paper, it is used in the proposed kHMC algorithm in the context of top-k high utility itemset mining but it could also be used in other high utility itemset mining algorithms. It is defined as follows.

Definition 11 (Tidset of an itemset [44])**.** The Tidset of an itemset X is denoted as $g(X)$ and defined as the set of transaction identifiers (Tids) of transactions containing X. Thus, $g(X) = \{tid \mid X \subseteq T_{tid} \wedge T_{tid} \in D\}$.

Definition 12 (Coverage)**.** Let i and j be two single items ($i, j \in I$). The item j is said to cover i if $g(i) \subseteq g(j)$. The coverage of the item i is denoted as Ç(i) and defined as Ç(i) = $\{j \in I, g(i) \subseteq g(j)\}$.

Property 9. Let i, j be two single items such that $j \in$ Ç(i). The utility of the itemset ij is greater than the utility of itemset i, that is $u(ij) \geq u(i) + |g(i)| \times p(j)$.

Proof. If $j \in$ Ç(i), then $g(i) \subseteq g(j)$. Hence, $g(ij) = g(i) \cap g(j) = g(i)$.

$$\begin{aligned} u(ij) &= \sum_{ij \subseteq T_d \wedge T_d \in D} u(ij, T_d) = \sum_{T_d \in g(ij)} u(ij, T_d) \\ &= \sum_{T_d \in g(ij)} u(i, T_d) + u(j, T_d) = \sum_{T_d \in g(i)} u(i, T_d) + \sum_{T_d \in g(i)} u(j, T_d) \\ &= u(i) + \sum_{T_d \in g(i)} q(j, T_d) \times p(j) \geq u(i) + \sum_{T_d \in g(i)} p(j) \\ &\geq u(i) + |g(i)| \times p(j) \end{aligned}$$

□

Example 2. In the example database shown in Figure 1, item d appears in transactions T_1, T_3, and T_4. Thus, $g(d) = \{1, 3, 4\}$. Since the item c appears in all transactions, $g(c) = \{1, 2, 3, 4, 5\}$. Furthermore, since $g(d) \subseteq g(c)$, we have that item $c \in$ Ç(d).

Lemma 2. (Relationship between TWU and coverage) Let i, j be two single items in a database D ($i, j \in I$). We have that $j \in$ Ç(i) if and only if $TWU(i) = TWU(ij)$.

Proof. Let $g(x\overline{y})$ denote the set of tids of transactions containing x, but not y. The TWU of the itemset ij is defined by the following formula.

$$\begin{aligned} TWU(ij) &= \sum_{T_d \in D \wedge ij \subseteq T_d} TU(T_d) = \sum_{T_d \in g(ij)} TU(T_d) \\ &= \sum_{T_d \in g(ij)} TU(T_d) + \sum_{T_d \in g(i\overline{j})} TU(T_d) - \sum_{T_d \in g(i\overline{j})} TU(T_d) \\ &= \sum_{T_d \in g(i)} TU(T_d) - \sum_{T_d \in g(i\overline{j})} TU(T_d) \\ &= TWU(i) - \sum_{T_d \in g(i\overline{j})} TU(T_d) \end{aligned}$$

Therefore,

$$\begin{aligned} & TWU(ij) = TWU(i) \\ \Leftrightarrow \quad & \sum_{T_d \in g(i\overline{j})} TU(T_d) = 0 \Leftrightarrow g(i\overline{j}) = \varnothing \\ \Leftrightarrow \quad & g(i) \subseteq g(j) \Leftrightarrow j \in \text{Ç}(i) \end{aligned}$$

□

The above lemma provides a simple way of calculating the coverage of any single item $i \in I$ using the TWU of single items, and the EUCST structure, which have been presented in the previous section.

Example 3. Consider the database of Figure 1. Figure 3 gives the TWU values of all single items in that database, and illustrates the corresponding EUCST. Consider the item g. We have that $TWU(g) = 38$. In the EUCST, there are two tuples containing g, which are $(g, c, 38)$ and $(g, e, 38)$. The TWU values stored in these tuples are equal to $TWU(g) = 38$. Hence, the coverage of item g is Ç(g) = $\{c, e\}$.

Definition 13 (Coverage itemset)**.** Let there be an itemset X. X is said to be a coverage itemset if and only if $X = pp_1p_2...p_r, p_l \in \text{Ç}(p)(1 \leq l \leq r)$.

Lemma 3 (Utility of a coverage itemset)**.** Let there be a coverage itemset $X = pp_1p_2...p_r$. The utility of X can be calculated using the following formula:

$$u(X) = \textstyle\sum_{1 \leq i \leq r} u(pp_i) - (r-1) \times u(p)$$

Proof. Because $p_i \in \text{Ç}(p)$, we have that $g(p) \subseteq g(p_i)$. Therefore, $g(pp_i) = g(p)$, and $g(pp_1p_2...p_r) = g(p)$.

$$\begin{aligned}
u(X) &= u(pp_1p_2...p_r) = \sum_{T_d \in g(p)} u(pp_1p_2...p_r, T_d) \\
&= \sum_{T_d \in g(p)} [u(p, T_d) + u(p_1, T_d) + ... + u(p_r, T_d)] \\
&= \sum_{T_d \in g(p)} [[(u(p, T_d) + u(p_1, T_d)) + ... + (u(p, T_d) + u(p_r, T_d)] - (r-1) \times u(p, T_d)] \\
&= \sum_{T_d \in g(p)} [[(u(p, T_d) + u(p_1, T_d)) + ... + (u(p, T_d) + u(p_r, T_d))] - (r-1) \times \sum_{T_d \in g(p)} u(p, T_d)] \\
&= \sum_{1 \leq i \leq r} \sum_{T_d \in g(p)} [u(p, T_d) + u(p_i, T_d)] - (r-1) \times u(p) \\
&= \sum_{1 \leq i \leq r} u(pp_i) - (r-1) \times u(p)
\end{aligned}$$

□

Example 4. Consider the database of Figure 1. The coverage of item g is $\text{Ç}(g) = \{e, c\}$, the utility of coverage itemset gec can be calculated as $u(gec) = u(ge) + u(gc) - u(g) = 16 + 15 - 7 = 24$.

Using the above lemma, the utility of any coverage itemset X can be obtained directly by adding the utilities of 2-itemsets and subtracting the utility of a 1-itemset. Thus, the utility of X can be obtained without performing utility-list intersections or additional database scans. This calculation only requires to precalculate the coverage of each item, and the utilities of 1-itemsets and 2-itemsets. This lemma can be applied to effectively prune the search space for the problem of top-k high utility itemset mining, and could be applied in other problems. In this work, this lemma is introduced to be used in a novel and efficient threshold raising strategy.

Lemma 4 (Utility of a superset using the coverage)**.** Let there be an item p, and $\text{Ç}(p)$ be the coverage of p. Furthermore, let there be an itemset $Y \subset \text{Ç}(p)$, and an item $p_i \in \text{Ç}(p)$ such that $p_i \notin Y$. We have that $u(p \cup Y \cup p_i) = u(p \cup Y) + u(pp_i) - u(p)$.

Proof. Assume that the itemset Y is of the form $Y = p_1p_2...p_r$. Hence, $p \cup Y \cup p_i = pp_1p_2...p_rp_i$. By Lemma 3, we have that:

$$\begin{aligned}
u(p \cup Y \cup p_i) &= u(pp_1p_2...p_rp_i) \\
&= \sum_{1 \leq j \leq r} u(pp_j) + u(pp_i) - r \times u(p) \\
&= (\sum_{1 \leq j \leq r} u(pp_j) - (r-1) \times u(p)) + u(pp_i) - u(p) \\
&= u(p \cup Y) + u(pp_i) - u(p)
\end{aligned}$$

□

Property 10. Let there be a single item p, and Ç(p) be its coverage. Furthermore, consider an itemset $Y \subseteq$ Ç(p), and an item $p_i \in$ Ç(p) such that $p_i \notin Y$. We have that $u(p \cup Y) < u(p \cup Y \cup p_i)$.

Proof. By Lemma 4, we have $u(p \cup Y \cup p_i) - u(p \cup Y) = u(pp_i) - u(p)$. According to Property 9, $u(pp_i) - u(p) \geq |g(p)| \times p(p_i) > 0$. □

Property 11. Let there be a single item p, and Ç(p) be its coverage. Furthermore, consider an itemset $Y \subseteq$ Ç(p), and an itemset X such that $X \cap Y = \varnothing$ and $p \notin X$. If $p \cup X$ is a high utility itemset, then $p \cup X \cup Y$ is also a high utility itemset.

Proof. Because Y is a subset of Ç(p), it follows that $g(p \cup X \cup Y) = g(p \cup X)$. The utility of the itemset $p \cup X \cup Y$ is calculated as:

$$\begin{aligned} u(p \cup X \cup Y) &= \sum_{T_d \in D} u(p \cup X \cup Y, T_d) \\ &= \sum_{T_d \in g(p \cup X)} [u(p \cup X, T_d) + u(Y, T_d)] \\ &= \sum_{T_d \in g(p \cup X)} u(p \cup X, T_d) + \sum_{T_d \in g(p \cup X)} u(Y, T_d) \\ &= u(p \cup X) + \sum_{T_d \in g(p \cup X)} u(Y, T_d) > u(p \cup X) \end{aligned}$$

If $p \cup X$ is a high utility itemset, then $u(p \cup X) \geq min\_util$. Thus, $u(p \cup X \cup Y) > u(p \cup X) \geq min\_util$. Therefore, $p \cup X \cup Y$ is a high utility itemset. □

Note that a *coverage itemset* is a closed itemset as defined in FIM. Closed itemsets are a subset of the set of all itemsets. Mining closed itemsets generally reduces the execution time in FIM. Several algorithms were proposed to mine closed itemsets [12]. In this paper, the concept of coverage is proposed to mine the top-k high utility itemsets. Novel definitions and properties are introduced in this paper related to the concept of coverage, which are specific to high utility itemset mining. In particular, the next section introduces a novel threshold raising strategy relying on the concept of coverage, to mine the top-k high utility itemsets.

8. Three Effective Threshold Raising Strategies

The proposed kHMC algorithm is designed to mine the top-k high utility itemsets. It initially sets an internal *min_util* threshold to zero, and employs three effective threshold raising strategies to raise the threshold to higher values. The next subsections present these three strategies.

8.1. The Real Item Utilities Threshold Raising Strategy

The first threshold raising strategy is named RIU [37]. It is applied immediately after the first database scan. The utilities of all items are calculated during the first database scan. The utility of an item i in a database D is denoted as $riu(i)$, and is calculated by the formula $\sum_{T_d \in D} u(i, T_d)$. The RIU strategy utilizes the utilities of items to raise the value of the *min_util* threshold, as follows.

Let I={i_1; i_2; . . . ; i_m} be the set of items in database D, and $R = \{riu_1, riu_2, ..., riu_m\}$ be the list of utilities of items in I, ordered by descending TWU values. Let riu_k denote the k-th largest value in R. The RIU strategy consists of raising the *min_util* threshold to the value $riu_k (1 \leq k \leq m)$. This new value will then be used as *min_util* threshold by the mining process, until the threshold is increased to a larger value by another threshold raising strategy.

Example 5. Consider the database of Figure 1, and that $k = 3$. After the first database scan, the utilities of items in D have been calculated (see Figure 6). The third largest value in the list R is 16. Therefore, the value of *min_util* is increased to 16. This new *min_util* value will then be used by the mining process. A second database scan is then done using the new *min_util* value, to construct the EUCST.

Item	a	b	c	d	e	f	g
Utility	20	16	13	20	15	5	7

Figure 6: Utilities of items in database D.

8.2. The Co-occurrence Threshold Raising Strategy

The second threshold raising strategy is named CUD (Co-occurrence Utility Descending order). It is a novel strategy that is applied after the RIU strategy and the second database scan. CUD increases *min_util* using the utilities of 2-itemsets stored in the EUCST. As mentioned previously, the EUCST structure employed by the proposed algorithm is very efficient for pruning the search space. The EUCST is used to avoid performing join operations for constructing utility lists. It is designed to store and quickly retrieve the TWU values of pairs of items. Each tuple in the EUCST contains a pair of items having a TWU no less than *min_util*, that may thus be a high utility itemset. Hence, the utility of the 2-itemsets stored in the EUCST can be considered for raising the threshold. The utility of each entry in the EUCST can be calculated easily during the construction of the EUCST, while performing the second database scan. The benefits of the CUD strategy are that: (i) it uses the same structure as the main pruning strategy, (ii) it is easy to calculate the utility of pairs of items during the second database scan, (iii) it is efficient to store this information in terms of memory and runtime using hash maps, and (iv) the utilities of 2-itemsets help raising the value of the *min_util* threshold closer to the boundary to obtain the top-k high utility itemsets. The structure for storing the utilities of pairs of items is called the CUD utility Matrix (CUDM). It is built at the same time as the EUCST structure. When the algorithm reads a transaction T_d, the utility of each 2-itemset xy appearing in the transaction is increased in the CUDM by the value $u(x, T_d) + u(y, T_d)$.

Lemma 5. Let $H_D(k)$ denote the top-k high utility itemsets in a database D. Let CUD_k be the k-th largest utility in the CUDM. We have that $u(X) \geq CUD_k, \forall X \in H_D(k)$.

Proof. Let $CUDM_k = \{X \in CUDM, u(X) \geq CUD_k\}$. Assume that $\exists Y \in H_D(k), u(Y) < CUD_k$. We have that $\forall X \in CUDM_k, u(X) \geq CUD_k > u(Y)$. Therefore, $CUDM_k \subseteq H_D(k)$. Furthermore, $Y \in H_D(k)$. This is in contradiction with the statement that $H_D(k)$ is the set of top-k high utility itemsets. Thus, $\nexists Y \in H_D(k), u(Y) < CUD_k$, which means that $u(X) \geq CUD_k, \forall X \in H_D(k)$. □

This lemma is used in the proposed CUD threshold raising strategy. After the utilities of all pairs of items have been calculated by constructing the CUDM, the *min_util* threshold is raised to the k-th highest value in the CUDM.

Example 6. Consider the example database depicted in Figure 1, and that $k = 3$. The CUD strategy first reads the transaction T_1, which contains three items: d, a, and c. Thus, three 2-itemsets are stored in the CUDM, i.e. da, dc, and ac, having the following utilities: $u(da, T_1) = 7, u(dc, T_1) = 3$, and $u(ac, T_1) = 6$. Next, the transaction T_2 is processed. The 2-itemsets in T_2 are: ga, ge, gc, ae, ac, and ec, having the following utilities: 15, 11, 11, 16, 16, and 12. Because the itemset ac is already in the CUDM with a value of 6, its utility in the CUDM is updated to $6 + 16 = 22$. All the other transactions are processed in the same way. The result is shown in Table 1. The third largest value in the CUDM is 27. Therefore, the CUD strategy raises *min_util* to 27.

8.3. The Coverage Threshold Raising Strategy

An important observation made in this paper is that the coverage relation between single items can be used to raise the *min_util* threshold using the properties proposed in the previous section. This subsection presents such a threshold raising strategy, based on the concept of coverage. This strategy is named COV. It relies on a structure named COVerage List (COVL), which is a list of utility values. The construction of the COVL is done as follows. Initially, all values stored in the CUDM are inserted in the COVL. Then, for each single item $i \in I$, the COV strategy inserts the combinations of i with all subsets of its coverage Ç(i) in the COVL. The construction of the COVL is completed when all items have been processed. The

Table 1: The utilities in the CUDM

Item	f	g	d	b	a	e
g						
d	17					
b	9	6	30			
a	10	15	24	9		
e	8	16	24	25	24	
c	6	15	25	22	28	27

Algorithm 3 CoverageContruct Procedure

Output: The COVL.

1: **for** each (item $x \in I$) **do**
2: $I(x).COV \longleftarrow \emptyset$;
3: **for** (each item $y \in I$ and $y \succ x$) **do**
4: **if** $(EUCST(x,y) = TWU(x))$ **then**
5: $I(x).COV \longleftarrow I(x).COV \cup y$;
6: **end if**
7: **end for**
8: **end for**

detailed algorithm for constructing the coverage list is given in Algorithm 3. Let COV_k denote the k-th highest value in the COVL.

Lemma 6. Let $H_D(k)$ denote the utilities of the top-k high utility itemsets in a database D. Let COV_k be the k-th highest utility value in COVL. We have that $u(X) \geq COV_k, \forall X \in H_D(k)$.

Proof. This lemma can be easily proven in a similar way to the proof of lemma 5 for the CUD strategy, and presented in section 8.2. □

The proposed COV strategy consists of raising the *min_util* threshold to COV_k.

9. The Proposed kHMC Algorithm

This section presents the proposed kHMC algorithm, for mining the top-k high utility itemsets. kHMC employs the efficient EUCST structure to prune the search space using the EUCPT strategy. This strategy can avoid numerous join operations for constructing utility-lists. It was shown that this strategy can eliminate up to 95% of the candidates [36]. Moreover, the designed TEP strategy is also applied for reducing the search space. In addition, threshold raising strategies RIU, CUD, and COV are used to raise the *min_util* threshold. The threshold is also dynamically raised by the kHMC algorithm as the utilities of itemsets in the search space are calculated. In addition, the proposed procedure that constructs utility-lists with a complexity of $O(m+n)$ and an early abandoning strategy are introduced in the proposed algorithm (described in Section 5).

The main procedure of kHMC is shown in Algorithm 4. kHMC first scans the database to obtain the TWU and utilities (RIU) of single items (Line 2). The algorithm then sorts the list of single items in I by ascending order of TWU (Line 3), and raises *min_util* using the RIU strategy (Line 4). kHMC then scans the database again to build the EUCST and the CUDM (Line 5). The threshold raising strategy CUD is then applied (Lines 6-7). The coverage relation is then constructed by applying the *CoverageContruct* procedure (Line 8). The threshold raising strategy COV is then applied (Line 9). The EUCST is finally created using the current *min_util* value (Line 10). Then, the *Search* procedure is applied to search for high utility candidates, to be inserted into R_{topk} (Line 11), and the real top-k high utility itemsets from R_{topk} are output (Line 12). The main steps of the *Search* procedure are the following. For each itemset Px in

ExtensionsOfP having a utility no less than *min_util*, Px is added to the set of candidates R_{topk}. Then, *min_util* is raised to the k-th highest utility in R_{topk}, and each itemset having a utility lower than that value is removed from R_{topk} (Lines 2-8). If this condition is satisfied (Line 9), extensions of Px will be considered by the search procedure. Px will be combined with each other itemset Py in *ExtensionsOfP* such that $y \succ x$ to form larger itemsets, if there is a tuple in the EUCST for x and y such that this tuple's utility value is no less than *min_util* (Lines 12-13) and the extension upper-bound utility of itemset X by extension Xy is no less than *min_util* (Lines 14-15). The utility-list of the resulting itemset Pxy is constructed by applying the *iConstruct* procedure (Line 17). If the utility-list is not empty (Line 18), then Pxy and its extensions will be considered by the search procedure (Line 19). Then, the *Search* procedure is recursively called with Px. The procedure stops if no extensions can be formed. By the proposed lemmas, it can be easily seen that the algorithm is correct and complete for mining the top-k high utility itemsets.

10. Experimental Evaluation

This section presents an extensive experimental evaluation to evaluate the performance of the proposed kHMC algorithm, including its pruning strategies and threshold raising strategies. The performance of kHMC is compared to the state-of-the-art REPT [37] and TKO [39] algorithms. REPT and TKO were recently proposed for mining top-k high utility itemsets, and they were shown to be more efficient than other state-of-the-art algorithms such as TKU [38], UP-Growth [24], and UP-Growth+ [28]. Experiments were performed on a computer equipped with a 64 bit Core i3, 2.4 GHz Intel Processor and 4 GB of RAM, running the Windows 7 operating system. All of the algorithms were implemented in Java, and both real and synthetic datasets were used in the experiments. The characteristics of the real datasets used in the experiments are shown in Table 2, where #Trans, #Items, and #Avg indicate the number of transactions, the number of distinct items, and the average transaction length, respectively.

Table 2: Details of the datasets.

Dataset	#Trans	#Items	#Avg	Type
Accidents	340,183	468	33.8	Dense
Mushroom	8,124	119	23.0	Dense
Retail	88,162	16,470	10.3	Sparse
Chess	3196	75	37	Dense
Connect	67,557	129	43.0	Dense
Kosarak	990,002	41,270	8.1	Large

Accidents, Mushroom, Retail, Chess, Connect, and Kosarak were obtained from the FIMI Repository [45]. These datasets are real datasets with synthetic utility values. The internal utility values have been generated using a uniform distribution in the [1, 10] interval, and the external utility values have been generated using a Gaussian (normal) distribution [42], as in previous work [24, 28].

10.1. Evaluation of the EUCST structure

This section presents experimental results related to the proposed EUCST structure. For each dataset, the algorithm was applied with various values of k. The number of item pairs in the EUCST and the memory required to store these pairs were measured after applying the various threshold raising strategies.

Tables 3-8 show the pair count and the maximum memory usage for each dataset, after applying the RIU strategy, and after applying the CUD and COV strategies. When the parameter k is increased, the internal minimum utility threshold is raised to lower values, and the item pair count and memory usage are slightly larger. It can be seen that the memory usage and pair count for all datasets is quite low. From these results, it can be inferred that the EUCST after raising the threshold using the CUD and COV strategies is much more efficient than the EUCS, as it requires less memory and stores less item pairs. Thus, using the designed threshold strategies RIU, CUD, and COV, the EUCST contains fewer item pairs and consumes less memory. This considerably improves the overall performance of the proposed algorithm.

Algorithm 4 kHMC Algorithm

Input: a transaction database D and the value k
Output: the top-k high utility itemsets

1: Global variables: $min\_util \longleftarrow 0$; $R_{topk} \longleftarrow \varnothing$;
2: Scan D to calculate the TWU and real item utility (RIU) of each item;
3: Let I be the list of single items sorted by ascending order of TWU;
4: $min\_util \longleftarrow$ the k-th highest utility value in RIU; //RIU strategy
5: Scan D to build the utility-list of each item $i \in I$, the EUCST structure, and the CUDM structure;
6: $cud \longleftarrow$ the k-th highest utility value in the CUDM;
7: if ($min\_util < cud$) then $min\_util \longleftarrow cud$; //CUD strategy
8: **CoverageContruct**();
9: Raise min_util to COV_k; //COV strategy
10: Remove all entries lower than min_util in the EUCST ;
11: **Search** ($\varnothing$, I, EUCST);
12: Output the k itemsets having the highest utilities in R_{topk}

Procedure Search()
Input : P: an itemset, $ExtensionsOfP$: a set of extensions of P, and the EUCST structure
Output : a set of top-k high utility itemset candidates

1: **for** (each itemset $Px \in ExtensionsOfP$) **do**
2: **if** ($SUM(UL(Px).iutils) \geq min\_util$) **then**
3: $R_{topk} \longleftarrow R_{topk} \cup Px$;
4: **if** ($R_{topk}.size \geq k$) **then**
5: $min\_util \leftarrow$ the k-th highest utility in R_{topk};
6: keep only the k itemsets having the highest utilities in R_{topk};
7: **end if**
8: **end if**
9: **if** ($SUM(UL(Px).iutils) + SUM(UL(Px).rutils) \geq min\_util$) **then**
10: $ExtensionsOfPx \longleftarrow \varnothing$;
11: **for** (each itemset $Py \in ExtensionsOfP$ and $y \succ x$) **do**
12: **if** ($EUCST(x, y) \geq min\_util$) **then**
13: $Pxy \longleftarrow Px \cup Py$;
14: **if** ($teu(Px, y) < min\_util$) **then**
15: **continue**;
16: **end if**
17: $UL(Pxy) \longleftarrow$ **iConstruct**(P, Px, Py);
18: **if** ($UL(Pxy) \neq NULL$) **then**
19: $ExtensionsOfPx \longleftarrow ExtensionsOfPx \cup Pxy$;
20: **end if**
21: **end if**
22: **end for**
23: **Search** $(Px, ExtensionsOfPx, EUCST)$;
24: **end if**
25: **end for**

10.2. Evaluation of the Threshold Raising Strategies

The designed kHMC algorithm includes three effective threshold raising strategies named RIU, CUD, and COV. RIU is also used in the REPT algorithm [37], while CUD and COV are two novel strategies introduced in this paper.

To evaluate the effectiveness of the threshold raising strategies, an experiment was performed where the RIU, CUD, and COV strategies were compared with the PUD and RSD strategies used by the REPT

Table 3: Pair count and memory usage of the EUCST after applying the RIU, CUD, and COV threshold raising strategies on Accidents.

	RIU		*CUD and COV*	
k	Memory Size (MB)	Pair Count	Memory Size (MB)	Pair Count
1	0.9	5,268	0.29	1,330
5	1.08	6,340	0.34	1,552
10	1.46	8,534	0.41	1,791
50	2.96	17,341	0.65	2,411
100	4.32	25,276	0.85	2,864
500	8.01	46,815	1.52	4,963
1000	8.01	46,815	1.81	6,850

Table 4: Pair count and memory usage of the EUCST after applying the RIU, CUD, and COV threshold raising strategies on Mushroom.

	RIU		*CUD and COV*	
k	Memory Size (MB)	Pair Count	Memory Size (MB)	Pair Count
1	0.31	1,789	0.14	702
5	0.4	2,312	0.15	737
10	0.42	2,432	0.16	763
50	0.56	3,242	0.22	1,109
100	0.61	3,527	0.24	1,204
500	0.61	3,527	0.36	1,930
1000	0.61	3,527	0.43	2,357

Table 5: Pair count and memory usage of the EUCST after applying the RIU, CUD, and COV threshold raising strategies on Retail.

	RIU		*CUD and COV*	
k	Memory Size (MB)	Pair Count	Memory Size (MB)	Pair Count
1	0.03	171	0.0054	3
5	1.76	10,113	0.18	17
10	46.13	266,503	4.92	114
50	226.7	1,309,201	24.67	2,306
100	344.61	1,989,460	37.74	4,403
500	503.27	2,904,279	57.14	17,978
1000	551.74	3,183,355	64.94	34,022

Table 6: Pair count and memory usage of the EUCST after applying the RIU, CUD, and COV threshold raising strategies on Chess.

	RIU		*CUD and COV*	
k	Memory Size (MB)	Pair Count	Memory Size (MB)	Pair Count
1	0.31	1,791	0.24	1,368
5	0.39	2,276	0.26	1,424
10	0.39	2,276	0.27	1,475
50	0.42	2,445	0.28	1,550
100	0.44	2,582	0.3	1,652
500	0.44	2,582	0.36	2,025
1000	0.44	2,582	0.38	2,182

algorithm. The strategies were applied on each dataset, while varying the parameter k, and the threshold values obtained by each threshold raising strategy were measured. In this experiment, obtaining a higher

Table 7: Pair count and memory usage of the EUCST after applying the RIU, CUD, and COV threshold raising strategies on Connect.

	RIU		*CUD and COV*	
k	Memory Size (MB)	Pair Count	Memory Size (MB)	Pair Count
1	0.57	3,309	0.34	1,832
5	0.69	4,052	0.38	2,050
10	0.75	4,401	0.41	2,200
50	1.01	5,917	0.46	2,403
100	1.14	6,659	0.5	2,569
500	1.17	6,827	0.61	3,247
1000	1.17	6,827	0.71	3,886

Table 8: Pair count and memory usage of the EUCST after applying the RIU, CUD, and COV threshold raising strategies on Kosarak.

	RIU		*CUD and COV*	
k	Memory Size (MB)	Pair Count	Memory Size (MB)	Pair Count
1	0.0343	210	0.00047	2
5	103.65	666,208	0.0859	488
10	792.85	5,098,733	0.712	4,301
50	2,061.98	13,261,732	57.29	366,867
100	2,715.41	17,464,485	119.50	766,746
500	3,927.17	27,187,241	384.15	2,468,287

threshold value is better. Note that to be able to see the influence of each threshold raising strategy, each strategy is run independently in this experiment, unlike in [37] where the threshold values obtained by a strategy depends on prior strategies. Thus, in this experiment, the k-th value obtained by each strategy is displayed. Besides, note that for this experiment, the value of N required by the RSD strategy is respectively set to 200, 100, 1000, 30, 50, and 2000, for the Accidents, Mushroom, Retail, Chess, Connect, and Kosarak datasets as suggested in [37].

Tables 9-14 show the threshold values obtained by applying each threshold raising strategy. It can be observed that as the parameter k is increased, the threshold can be raised to larger values. Furthermore, it can be seen that the CUD and COV strategies raise the threshold to very high values. The more the threshold is raised, the more the search space is reduced, and as a result less itemsets need to be considered to find the top-k high utility itemsets. This directly influences the runtime and memory consumption of the algorithms. Besides, it can be observed that the RSD and PUD strategies are special cases of the CUD strategy. The CUD strategy covers all cases of the RSD and PUD strategies. This is the reason why the CUD strategy can raise the threshold higher than the RSD and PUD strategies. Applying the CUD strategy is simple and inexpensive in terms of runtime and memory, as described in Section 8.2. The three datasets Mushroom, Connect, and Chess are dense datasets. Thus, items in these datasets have large coverage sets. For this reason, the COV strategy can raise the threshold higher than the CUD strategy. Figures 7-10 compare results of the CUD, COV, PUD, and RSD threshold raising strategies on the Chess, Connect, Retail, and Mushroom datasets. It can be observed that the CUD and COV strategies raise the threshold higher than the PUD and RSD strategies, and by more than 100 times, in some cases.

10.3. Evaluation of the TEP Strategy

This section presents experimental results related to the evaluation of the proposed TEP pruning strategy. To evaluate the effectiveness of the TEP strategy, an experiment was performed on the Accidents, Retail, and Mushroom datasets, while varying the parameter k, and measuring the number of candidates generated by applying the TEP strategy. Two versions of kHMC named kHMC$_{Tep}$ and kHMC$_{NoTep}$ were prepared, where the TEP strategy is respectively activated and deactivated. Table 15 compares the number of candidates

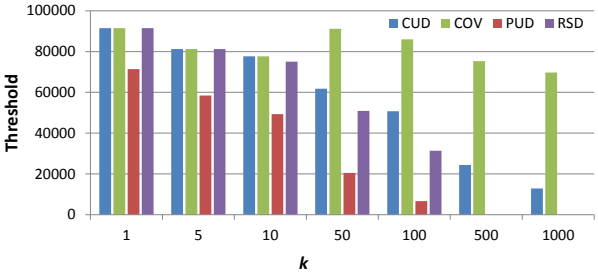

Figure 7: Threshold values obtained on Chess.

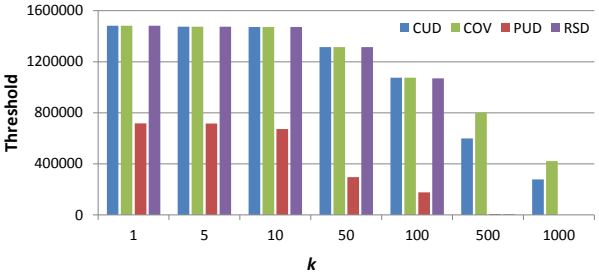

Figure 8: Threshold values obtained on Connect.

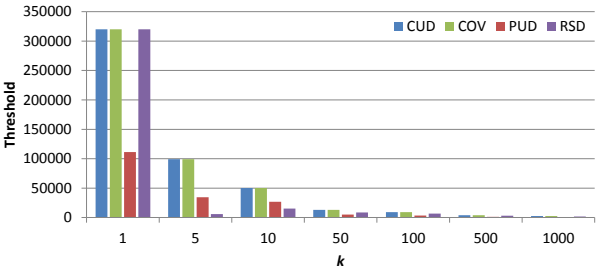

Figure 9: Threshold values obtained on Retail.

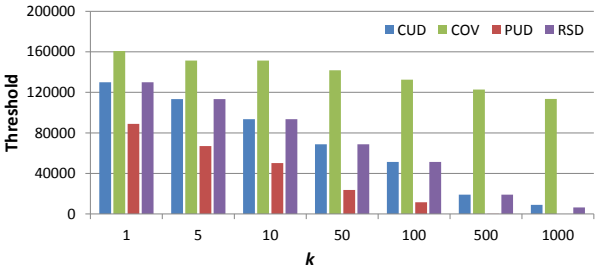

Figure 10: Threshold values obtained on Mushroom.

Table 9: Threshold values obtained on Accidents.

Algorithm		kHMC			REPT(N=200)	
k	*RIU*	*CUD*	*COV*	*PUD*	*RIU*	*RSD*
1	5,589,192	9,320,084	9,320,084	7,653,891	5,589,192	9,320,084
5	3,325,842	7,853,805	7,853,805	5,681,678	3,325,842	7,853,805
10	1,905,782	6,969,478	6,969,478	4,823,295	1,905,782	6,969,478
50	428,860	4,836,703	4,836,703	1,388,712	428,860	4,836,703
100	142,277	3,730,529	3,730,529	649,572	142,277	3,730,529
500		1,326,872	1,326,872	12,964		1,326,872
1000		695,739	695,739	642		695,739

Table 10: Threshold values obtained on Mushroom.

Algorithm		kHMC			REPT(N=100)	
k	*RIU*	*CUD*	*COV*	*PUD*	*RIU*	*RSD*
1	88,312	129,885	160,823	88,979	88,312	129,885
5	53,631	113,298	151,424	66,980	53,631	113,298
10	40,998	93,606	151,298	50,286	40,998	93,606
50	7,949	68,770	141,789	23,638	7,949	68,770
100	410	51,340	132,567	11,510	410	51,340
500		19,139	122,777	193		19,070
1000		8,993	113,495			6,401

generated by kHMC$_{Tep}$ and kHMC$_{NoTep}$ on the Accidents, Retail, and Mushroom datasets. It can be observed that kHMC$_{Tep}$ considers up to 21% less candidate than kHMC$_{NoTep}$.

10.4. Evaluation of the Efficiency of kHMC

This section compares the efficiency of the proposed kHMC algorithm with REPT [37] and TKO [39], for the Accidents, Retail, Mushroom, and Kosarak datasets. These datasets have been selected because they

Table 11: Threshold values obtained on Retail.

Algorithm			kHMC		REPT(N=1000)	
k	*RIU*	*CUD*	*COV*	*PUD*	*RIU*	*RSD*
1	278,199	320,051	320,051	111,343	278,199	320,051
5	82,271	99,222	99,222	34,495	82,271	5,981
10	20,846	50,168	50,168	26,733	20,846	15,259
50	8,080	13,017	13,017	5,109	8,080	8,551
100	5,724	9,136	9,136	3,374	5,724	6,803
500	2,052	3,986	3,986	1,158	2,052	3,140
1000	1,274	2,547	2,547	728	1,274	1,811

Table 12: Threshold values obtained on Chess.

Algorithm			kHMC		REPT(N=30)	
k	*RIU*	*CUD*	*COV*	*PUD*	*RIU*	*RSD*
1	52,038	91,500	91,500	71,367	52,038	91,500
5	35,290	81,247	81,247	58,467	35,290	81,247
10	32,676	77,676	77,676	49,314	32,676	75,017
50	6,623	61,813	91,197	20,512	6,623	50,902
100		50,759	86,024	6,653		31,380
500		24,384	75,264			
1000		12,814	69,714			

Table 13: Threshold values obtained on Connect.

Algorithm			kHMC		REPT(N=50)	
k	*RIU*	*CUD*	*COV*	*PUD*	*RIU*	*RSD*
1	742,554	1,483,020	1,483,020	717,790	742,554	1,483,020
5	737,272	1,474,578	1,474,578	716,340	737,272	1,474,578
10	674,164	1,472,118	1,472,118	672,756	674,164	1,472,118
50	109,341	1,314,648	1,314,648	295,545	109,341	1,314,648
100	7,156	1,075,593	1,075,593	176,082	7,156	1,070,541
500		599,088	800,558	6,942		5,941
1000		278,367	421,406			

Table 14: Threshold values obtained on Kosarak.

Algorithm			kHMC		REPT(N=2000)	
k	*RIU*	*CUD*	*COV*	*PUD*	*RIU*	*RSD*
1	6,617,618	8,912,841	8,912,841	8,519,820	6,617,618	8,912,841
5	972,156	2,023,210	2,023,210	1,535,670	972,156	2,023,210
10	396,823	1,290,328	1,290,328	635,846	396,823	1,290,328
50	94,648	352,209	352,209	185,291	94,648	185,291
100	53,707	231,239	231,239	117,103	53,707	231,239
500	16,149	87,496	87,496	35,981	16,149	87,496

include dense, sparse, and large datasets, and thus represent the main types of datasets used in real-life applications.

To discover the top-k high utility itemsets, REPT scans the database three times. In Phase I, REPT performs two database scans to generate a set of candidates. In Phase II, REPT scans the database again to calculate the exact utility of each candidate. This operation is very expensive. In this experiment, REPT is executed with various values for the parameter N of the RSD strategy. The notation $REPT(N = x)$

Table 15: Number of candidates generated by kHMC with/without the TEP strategy.

	Accidents		*Mushroom*		*Retail*	
k	$kHMC_{NoTep}$	$kHMC_{Tep}$	$kHMC_{NoTep}$	$kHMC_{Tep}$	$kHMC_{NoTep}$	$kHMC_{Tep}$
10	10,228	8,470	3,643	2,876	13	13
50	13,451	11,388	4,982	4,099	2,667	2,461
100	18,440	16,024	5,743	4,870	159,264	158,899
200	22,957	20,265	6,188	5,281	1,136,541	1,135,946
500	29,919	26,933	90,74	7,960	4,588,796	4,587,425
700	32,861	29,789	10,552	9,378	6,588,211	6,586,878
1000	38,013	34,796	11,620	10,524	8,963,480	8,961,775

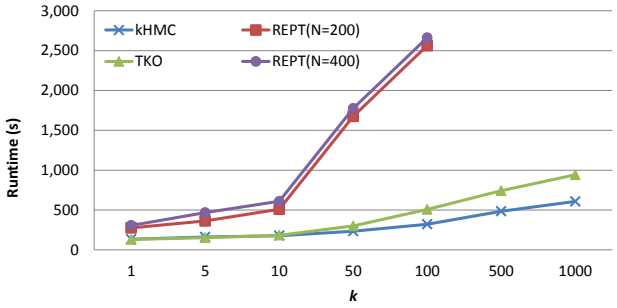

Figure 11: Runtimes on Accidents.

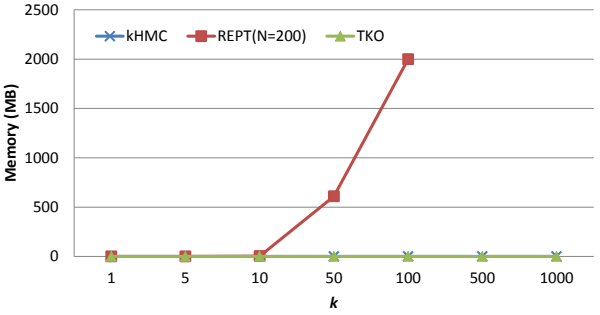

Figure 12: Memory used for candidates on Accidents.

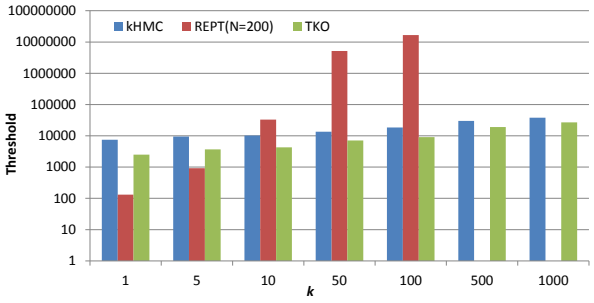

Figure 13: Number of potential candidates on Accidents.

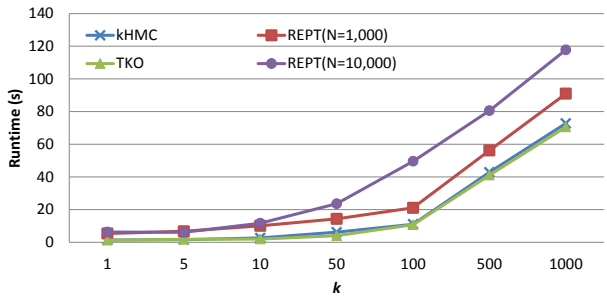

Figure 14: Runtimes on Retail.

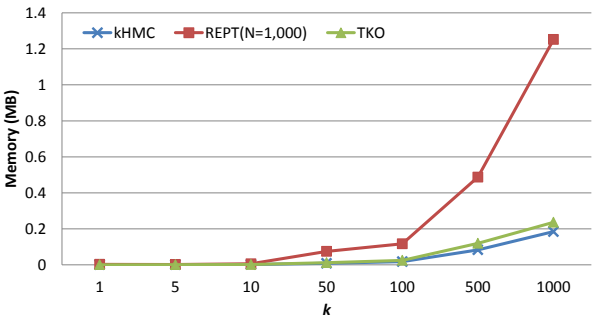

Figure 15: Memory used for candidates on Retail.

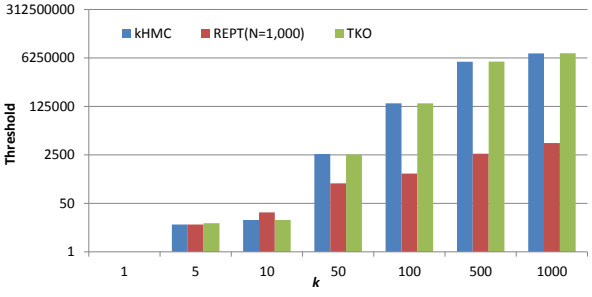

Figure 16: Number of potential candidates on Retail.

represents REPT with N=x. The proposed kHMC algorithm scans the database twice. The utility of an itemset is calculated directly by adding the utilities in its utility-list. In TKO, authors integrate the strategies RUC, RUZ, and EPB [39] for pruning the search space.

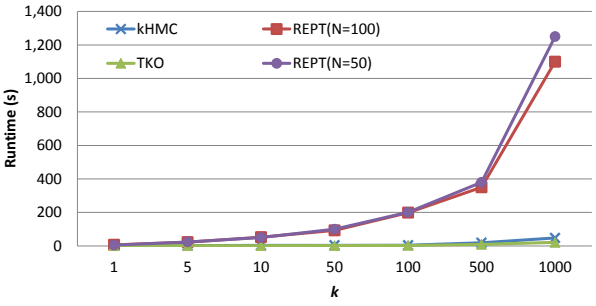

Figure 17: Runtimes on Mushroom.

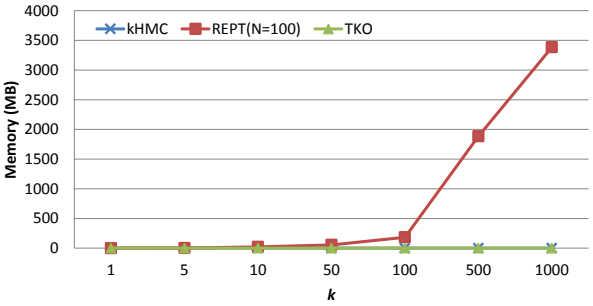

Figure 18: Memory used for candidates on Mushroom.

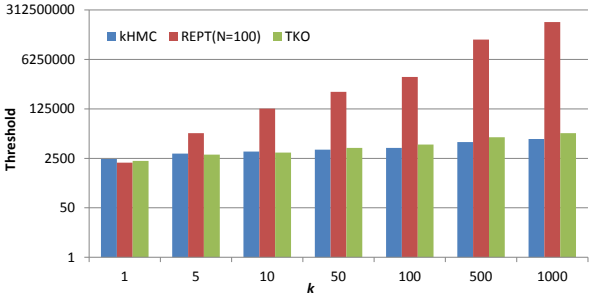

Figure 19: Number of potential candidates on Mushroom.

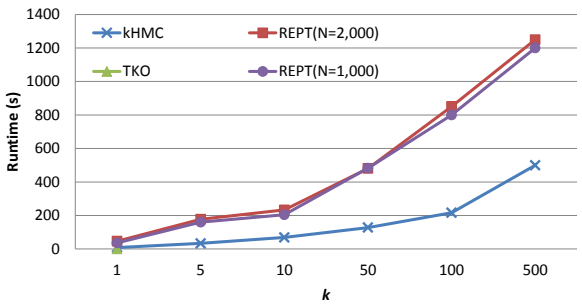

Figure 20: Runtimes on Kosarak.

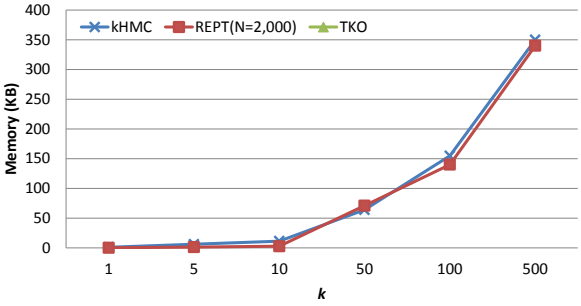

Figure 21: Memory used for candidates on Kosarak.

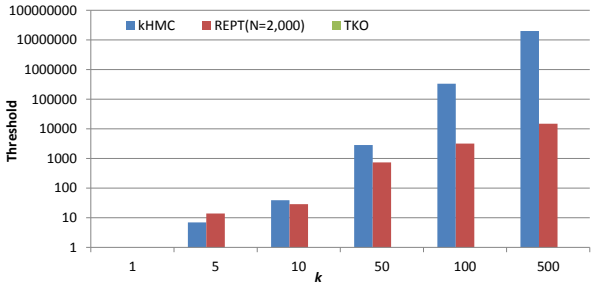

Figure 22: Number of potential candidates on on Kosarak.

Figures 11-22 show the runtime, the memory used to store potential candidates, and the number of candidates visited in the search space, for each algorithm and dataset. Detailed results for the Accidents dataset are shown in Figures 11-13. It can be observed that when k is small ($1 \leq k \leq 10$), the runtime and memory consumption are similar for all algorithms, and that for small values of k (k=1), the REPT algorithm is slightly slower than kHMC and TKO. The reason is that kHMC has to apply the CUD and COV threshold raising strategies, and this takes slightly more time than the PUD and RSD strategies, but the difference is very small. When k is set to larger values ($50 \leq k \leq 1{,}000$), the runtime and memory consumption of REPT are much larger than kHMC. In particular, when k is set to values greater than 100, REPT fails to provide a result (it is considered that an algorithm fails in this experiment if it takes more than 10,000 seconds to terminate). It can also be observed that kHMC outperforms TKO.

Figures 14-16 display results for the Retail dataset. It can be seen that kHMC consumes less memory than REPT. In terms of runtime, kHMC and TKO have similar performance. They perform slightly better than REPT because they generate a small number of candidates. For $N = 10{,}000$, REPT spends more time for enumerating 2-itemsets consisting of promising items from each transaction by using the RSD strategy

than with N = 1,000, so $REPT(N = 10,000)$ is the worse.

Detailed results for the Mushroom dataset are shown in Figures 17-19. Results for Mushroom are similar to the results for the Accidents dataset. When k is set to small values, the performance of REPT is similar to kHMC, but when k is set to larger values, the difference becomes clear and large. This is explained as follows. For dense datasets such as Mushroom and Accidents, when k is set to small values, the threshold is raised to high values, and as a result, the number of candidates is small. Therefore, algorithms are fast and require a small amount of memory. But when k is set to large values, the threshold cannot be raised as high. In this case, the REPT algorithm generates a huge amount of candidates (millions), while kHMC employs co-occurrence pruning and TEP pruning, and as a result the number of candidates generated by kHMC remains small. Moreover, because REPT is a two phase algorithm, it has to scan the database to get the utilities of all (millions of) candidates. This process is very expensive in terms of runtime. It becomes especially inefficient for large values of k and N. On the other hand, in kHMC, the utilities of itemsets are directly and easily calculated using utility-lists. The performance of TKO is good and similar to kHMC for this dataset.

Figures 20-22 display results for the large Kosarak dataset. When k is set to small values ($1 \leq k \leq 50$), the threshold can be raised high and the number of candidates is small for both kHMC and REPT. On overall, the runtime of kHMC is less than the runtime of REPT for this dataset. When k is set to large values ($k > 50$), the number of candidates generated by kHMC grows fast (more than a million) because the number of items in Kosarak is huge (approximately 40,000 items), and hence the threshold cannot be raised to high values. But the runtime of kHMC is still less than the runtime of REPT. This result is explained as follows. Although the number of candidates generated by REPT is less than the number of candidates generated by kHMC, REPT needs to scan the database to calculate the exact utilities of all candidates in Phase II. This is very expensive since Korasak has a huge amount of transactions and a large number of items. Another reason is that kHMC is a one phase algorithm. Thus, it obtains the utilities of itemsets by performing utility-list intersections rather than scanning the database. Therefore, REPT takes more time than kHMC to mine the top-k high utility itemsets. In contrast, TKO cannot be run on Kosarak since it runs out of memory because of its EPB strategy, which requires to maintain multiple possible branches of the search space into memory. The EPB strategy explores the most promising branches of the search space first, to find itemsets having a high utility early. Hence, results show that the number of candidates generated by TKO is slightly less than kHMC but TKO requires additional time to sort and maintain the set of promising branches. Furthermore, the proposed method for intersecting utility-lists has a lower complexity than the one used in TKO. For these reasons, the runtime of kHMC is less than TKO. Besides, experimental results show that the performance of REPT can greatly vary for different values of N. The selection of N has a major influence on the RSD strategy in REPT, especially on large datasets. However, it may be difficult for users to select appropriate values for the parameter N of REPT. This drawback of REPT was also observed by Tseng et al. [39]. In conclusion, the previously described experiments show that kHMC outperforms REPT and TKO.

10.5. Evaluation of the Scalability

Finally, the scalability of the algorithms is compared under different parameter settings. In these experiments, k is fixed to 5,000. Synthetic datasets have been generated using the IBM data generator [39] and the database size is varied from 100K to 500K transactions. Figure 23 shows the characteristics of the synthetic datasets, where the number of transactions x is varied from 100K to 500K transactions. For this experiment, the parameter N of REPT is set to 1,000. Figure 24 shows the runtime of the algorithms w.r.t to the database size. Note that both REPT and TKO failed to run on synthetic datasets (it is considered that an algorithm fails if it terminates in more than 10,000 seconds or run out of memory). Results show that the proposed algorithm has good scalability under different parameter settings.

11. Conclusion

This paper proposed an efficient algorithm named kHMC for top-k high utility itemset mining. The kHMC algorithm introduces several novel ideas. First, it includes an improved pruning strategy named

Description	Setting
- Number of transactions	xK
- Average transaction length	10
- Number of items	5000

Figure 23: Synthetic datasets characteristics.

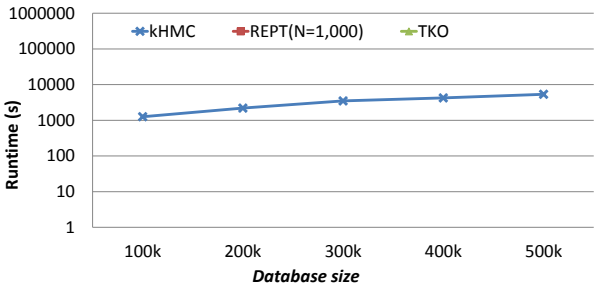

Figure 24: Scalability evaluation of the algorithms.

EUCPT to reduce the number of join operations and prune the search space. Second, a novel and efficient pruning strategy named TEP was proposed for reducing the search space by pruning transitive extensions. Third, a novel intersection procedure for constructing a utility-list in $O(m + n)$ time complexity was introduced. Fourth, an early abandoning strategy applied during the construction of utility-lists was also presented. These techniques not only help to find the top-k high utility itemsets effectively, but they also reduce the runtime and memory consumption of the algorithm considerably. But major challenges in top-k high utility itemset mining are also to design effective strategies for initializing and raising the internal minimum utility threshold during the mining process. Thus, in the proposed kHMC algorithm, three additional strategies are employed to raise the threshold close to the optimal boundary threshold value. In particular, a new threshold raising strategy based on a novel concept of coverage was proposed. The concept of coverage was shown to be useful for top-k high utility itemset mining and may be used in other problems related to high utility pattern mining. An extensive experimental evaluation was conducted to confirm the effectiveness and the high efficiency of the proposed algorithm for finding the top-k high utility itemsets.

Although the proposed algorithm was shown to outperform previous algorithms in terms of both runtime and memory, there is still room for improvement, and there are several opportunities for future work. In particular, the concept of coverage is employed in the new COV threshold raising strategy. The coverage should be studied and applied to prune the search space in other high utility pattern mining problems such as closed and maximal high utility itemset mining. This is an interesting possibility for future work.

Acknowledgements

This work was partially funded by the National Natural Science Foundation of China (Grant Nos. 61472124, 61202109, 61472126, 61402400).

References

[1] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, VLDB (1994) 487–499.
[2] T. Calders, B. Goethals, Mining all non-derivable frequent itemsets, Principles of Data Mining and Knowledge Discovery. 6th European Conference, PKDD 2002. Proceedings, 2002.
[3] J. Han, H. Cheng, D. Xin, X. Yan, Frequent pattern mining: current status and future directions, Data Mining and Knowledge Discovery 15 (1) (2007) 55–86.
[4] J. W. Han, J. Pei, Y. W. Yin, Mining frequent patterns without candidate generation, Sigmod Record 29 (2) (2000) 1–12.
[5] R. Agrawal, T. Imielinski, A. Swami, Mining association rules between sets of items in large databases, SIGMOD Record 22 (2) (1993) 207–16.
[6] Y. L. Cheung, A. W. C. Fu, Mining frequent itemsets without support threshold: With and without item constraints, IEEE Transactions on Knowledge and Data Engineering 16 (9) (2004) 1052–1069.
[7] J. Han, J. Pei, Y. Yin, R. Mao, Mining frequent patterns without candidate generation: A frequent-pattern tree approach, Data Mining and Knowledge Discovery 8 (1) (2004) 53–87.
[8] M. J. Zaki, K. Gouda, Fast vertical mining using diffsets, Proceedings of the 2003 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2003.
[9] G. Liu, H. Lu, W. Lou, Y. Xu, J. Yu, Efficient mining of frequent patterns using ascending frequency ordered prefix-tree, Data Mining and Knowledge Discovery 9 (2) (2004) 249–274.

[10] T.-L. Dam, K. Li, P. Fournier-Viger, Q.-H. Duong, An efficient algorithm for mining top-rank-k frequent patterns, Applied Intelligence (2016) 1–16.
[11] G. Grahne, J. F. Zhu, Fast algorithms for frequent itemset mining using fp-trees, IEEE Transactions on Knowledge and Data Engineering 17 (10) (2005) 1347–1362.
[12] J. W. Han, J. Y. Wang, Y. Lu, P. Tzvetkov, Mining top-k frequent closed patterns without minimum support, 2002 IEEE International Conference on Data Mining, Proceedings, 2002.
[13] P. Jian, H. Jiawei, L. Hongjun, S. Nishio, T. Shiwei, Y. Dongqing, H-mine: fast and space-preserving frequent pattern mining in large databases, IIE Transactions 39 (6) (2007) 593–605.
[14] U. Yun, G. Lee, K. H. Ryu, Mining maximal frequent patterns by considering weight conditions over data streams, Knowledge-Based Systems 55 (2014) 49–65.
[15] J. C.-W. Lin, W. Gan, P. Fournier-Viger, T.-P. Hong, V. S. Tseng, Efficient algorithms for mining high-utility itemsets in uncertain databases, Knowledge-Based Systems 96 (2016) 171–187.
[16] M. A. Nishi, C. F. Ahmed, M. Samiullah, B.-S. Jeong, Effective periodic pattern mining in time series databases, Expert Systems with Applications 40 (8) (2013) 3015–3027.
[17] W. Yao-Te, A. J. T. Lee, Mining web navigation patterns with a path traversal graph, Expert Systems with Applications 38 (6) (2011) 7112–22.
[18] A. Deepak, D. Fernandez-Baca, S. Tirthapura, M. J. Sanderson, M. M. McMahon, Evominer: frequent subtree mining in phylogenetic databases, Knowledge and Information Systems 41 (3) (2014) 559–590.
[19] A. Salam, M. S. H. Khayal, Mining top-k frequent patterns without minimum support threshold, Knowledge and Information Systems 30 (1) (2012) 57–86.
[20] J. Y. Wang, J. W. Han, Y. Lu, P. Tzvetkov, Tfp: An efficient algorithm for mining top-k frequent closed itemsets, IEEE Transactions on Knowledge and Data Engineering 17 (5) (2005) 652–664.
[21] S.-C. Ngan, T. Lam, R. C.-W. Wong, A. W.-C. Fu, Mining n-most interesting itemsets without support threshold by the cofi-tree, International Journal of Business Intelligence and Data Mining 1 (1) (2005) 88–106.
[22] A. W.-c. Fu, R. W.-w. Kwong, J. Tang, Mining n-most interesting itemsets, in: Foundations of Intelligent Systems, Springer, 2010, pp. 59–67.
[23] J.-j. Shi, Y.-q. Miao, Top-k frequent closed item set mining in microarray data, Computer Engineering 37 (2) (2011) 60–2.
[24] V. S. Tseng, C.-W. Wu, B.-E. Shie, P. S. Yu, Up-growth: an efficient algorithm for high utility itemset mining (2010).
[25] C. F. Ahmed, S. K. Tanbeer, J. Byeong-Soo, L. Young-Koo, Efficient tree structures for high utility pattern mining in incremental databases, Knowledge and Data Engineering, IEEE Transactions on 21 (12) (2009) 1708–1721.
[26] Y. Liu, W. K. Liao, A. Choudhary, A two-phase algorithm for fast discovery of high utility itemsets, Vol. 3518 of Lecture Notes in Artificial Intelligence, 2005, pp. 689–695.
[27] H. Yao, H. J. Hamilton, C. J. Butz, A foundational approach to mining itemset utilities from databases, Proceedings of the Fourth Siam International Conference on Data Mining, 2004.
[28] V. S. Tseng, B.-E. Shie, C.-W. Wu, P. S. Yu, Efficient algorithms for mining high utility itemsets from transactional databases, IEEE Trans. Knowl. Data Eng. 25 (8) (2013) 1772–1786.
[29] C.-W. Lin, T.-P. Hong, W.-H. Lu, An effective tree structure for mining high utility itemsets, Expert Systems with Applications 38 (6) (2011) 7419–7424.
[30] Y.-C. Li, J.-S. Yeh, C.-C. Chang, Isolated items discarding strategy for discovering high utility itemsets, Data and Knowledge Engineering 64 (1) (2008) 198–217.
[31] M. Liu, J. Qu, Mining high utility itemsets without candidate generation, in: 21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012, 2012, pp. 55–64.
[32] W. Song, Y. Liu, J. Li, Bahui: Fast and memory efficient mining of high utility itemsets based on bitmap, International Journal of Data Warehousing and Mining 10 (1) (2014) 1–15.
[33] G.-C. Lan, T.-P. Hong, V. S. Tseng, An efficient projection-based indexing approach for mining high utility itemsets, Knowledge and Information Systems 38 (1) (2014) 85–107.
[34] U. Yun, H. Ryang, K. H. Ryu, High utility itemset mining with techniques for reducing overestimated utilities and pruning candidates, Expert Systems with Applications 41 (8) (2014) 3861–3878.
[35] M. J. Zaki, Scalable algorithms for association mining, IEEE Transactions on Knowledge and Data Engineering 12 (3) (2000) 372–390.
[36] P. Fournier-Viger, C. Wu, S. Zida, V. S. Tseng, FHM: faster high-utility itemset mining using estimated utility co-occurrence pruning, in: Foundations of Intelligent Systems - 21st International Symposium, ISMIS 2014, Roskilde, Denmark, June 25-27, 2014. Proceedings, 2014, pp. 83–92.
[37] H. Ryang, U. Yun, Top-k high utility pattern mining with effective threshold raising strategies, Knowledge-Based Systems 76 (2015) 109–126.
[38] C. Wu, B. Shie, V. S. Tseng, P. S. Yu, Mining top-k high utility itemsets, in: The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012, 2012, pp. 78–86.
[39] V. S. Tseng, C.-W. Wu, P. Fournier-Viger, P. S. Yu, Efficient algorithms for mining top-k high utility itemsets, Knowledge and Data Engineering, IEEE Transactions on 28 (1) (2015) 54–67.
[40] C. Raymond, Y. Qiang, S. Yi-Dong, Mining high utility itemsets, Third IEEE International Conference on Data Mining, 2003.
[41] H. Xiong, M. Brodie, S. Ma, TOP-COP: Mining TOP-K strongly correlated pairs in large databases, IEEE International Conference on Data Mining, 2006, pp. 1162–1166.
[42] P. Fournier-Viger, A. G. Penalver, T. Gueniche, A. Soltani, C.-W. Wu, V. S. Tseng, Spmf: a java open-source pattern mining library, Journal of Machine Learning Research (JMLR) (2014) 15:3389–3393.

[43] S. Krishnamoorthy, Pruning strategies for mining high utility itemsets, Expert Systems with Applications 42 (5) (2015) 2371–2381.
[44] V. S. Tseng, C. Wu, P. Fournier-Viger, P. S. Yu, Efficient algorithms for mining the concise and lossless representation of high utility itemsets, IEEE Trans. Knowl. Data Eng. 27 (3) (2015) 726–739.
[45] Frequent itemset mining implementations repository (2012).
URL http://fimi.cs.helsinki.fi