Knowl Inf Syst (2016) 49:597–627
DOI 10.1007/s10115-015-0914-8

REGULAR PAPER

# On efficiently mining high utility sequential patterns

**Jun-Zhe Wang**[1] · **Jiun-Long Huang**[1] · **Yi-Cheng Chen**[2]

Received: 25 November 2014 / Revised: 10 October 2015 / Accepted: 29 December 2015 /
Published online: 11 January 2016
© Springer-Verlag London 2016

**Abstract** High utility sequential pattern mining is an emerging topic in pattern mining, which refers to identify sequences with high utilities (e.g., profits) but probably with low frequencies. To identify high utility sequential patterns, due to lack of downward closure property in this problem, most existing algorithms first generate candidate sequences with high sequence-weighted utilities (SWUs), which is an upper bound of the utilities of a sequence and all its supersequences, and then calculate the actual utilities of these candidates. This causes a large number of candidates since SWU is usually much larger than the real utilities of a sequence and all its supersequences. In view of this, we propose two tight utility upper bounds, prefix extension utility and reduced sequence utility, as well as two companion pruning strategies, and devise HUS-Span algorithm to identify high utility sequential patterns by employing these two pruning strategies. In addition, since setting a proper utility threshold is usually difficult for users, we also propose algorithm TKHUS-Span to identify top-$k$ high utility sequential patterns by using these two pruning strategies. Three searching strategies, guided depth-first search (GDFS), best-first search (BFS) and hybrid search of BFS and GDFS, are also proposed to improve the efficiency of TKHUS-Span. Experimental results on some real and synthetic datasets show that HUS-Span and TKHUS-Span with strategy BFS are able to generate less candidate sequences and thus outperform other prior algorithms in terms of mining efficiency.

✉ Jiun-Long Huang
jlhuang@cs.nctu.edu.tw

Jun-Zhe Wang
jzwang@cs.nctu.edu.tw

Yi-Cheng Chen
ycchen@mail.tku.edu.tw

[1] Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan, ROC

[2] Department of Computer Science and Information Engineering, Tamkang University, New Taipei City, Taiwan, ROC

**Keywords** High utility sequential pattern · High utility sequential pattern mining · Top-$k$ high utility sequential pattern · Utility mining

## 1 Introduction

Sequential pattern mining, which is to discover sequences with high frequencies in a sequence database (SDB), plays an important role in data mining. Many algorithms have been proposed to address this problem, and sequential pattern mining has wide real-life applications, such as market basket analysis, Web click log analysis and network traffic analysis [5,7–9,16,19,22,24]. However, without considering the information such as selling quantity and profit in a SDB, classical sequential pattern mining algorithms may lose some sequences with low frequencies but with high utilities (e.g., profits). For example, in a consumer electronics retail store, the unit profit of selling a LCD television is much higher than that of selling an extension cord, but the selling quantity of LCD televisions is usually much smaller than that of extension cords. Hence, sequential pattern mining algorithms may filter out LCD televisions and obtain some patterns related to extension cords. Therefore, how to mine sequential patterns catching business or users' interest is an important research topic nowadays.

To address this problem, some studies argued that both selling quantity and unit profit of each item should be considered together. In high utility sequential pattern mining [3,26], each item in the database has an external utility (e.g., unit profit) and an internal utility (e.g., selling quantity). The utility of a sequence represents its importance, which can be measured in terms of profit (e.g., sum of product of selling quantity and unit profit of each item in the sequence) or other information which business or users concern. Therefore, high utility sequential pattern mining refers to find out sequences in a sequence database with utilities no less than a user-specified minimum utility threshold.

However, mining high utility sequential patterns is challenging due to the following two reasons. First, the utility of a sequence is neither monotone nor anti-monotone. Therefore, most techniques used in sequential pattern mining algorithms, which rely on the anti-monotonicity of support to prune the search space, cannot be directly applied to find out high utility sequential patterns. To address this problem, sequence-weighted utility (SWU), which is an upper bound of the utilities of a sequence and all its supersequences, has been proposed in [3] to prune the search space in high utility sequential pattern mining. A sequence can be safely pruned when the SWU of this sequence is smaller than the minimum utility threshold. Otherwise, this sequence is considered as a candidate sequence. Generally, algorithms applying SWU first scan the database to find out the set of candidate sequences whose SWUs are no less than the utility threshold, and then scan the database again to find out the high utility sequential patterns by calculating the utility of each candidate sequence. However, algorithms using SWU may generate a huge amount of candidates and hence prolong the execution time. In view of this, we design two tighter upper bounds of utility and prove that using these bounds will not wrongly prune any high utility sequential patterns. Second, for a sequence $t$ and a sequence $s$, where $t$ is a subsequence of $s$, $s$ may contain multiple instances of $t$ and each instance is of its own utility. Thus, a feasible way is to define the utility of $t$ in $s$ as the maximum utility of all instances of $t$ in $s$. In order to calculate the utility of a subsequence $t$ in a sequence $s$, a brute force way is to find out each instance of $t$ in $s$, calculate the utility of each instance and then obtain the maximum among the utilities of these instances. Such process will degrade the performance of calculating the utility of any sequence.

To address the above problems, we propose an efficient algorithm HUS-Span for mining high utility sequential patterns. Specifically, we propose two utility upper bounds, named prefix extension utility (PEU) and reduced sequence utility (RSU), and devise two pruning strategies accordingly. HUS-Span then uses these two pruning strategies to prune the search space. Compared with SWU, PEU and RSU are much tighter and hence generate less candidates. In addition, a data structure named utility-chain is also proposed to speed up the calculation of the PEUs, RSUs and utilities of sequences. The utility-chain of a sequence $t$ stores the compact utility information so that the PEU, RSU and utility of $t$ can be quickly obtained from the utility-chain without enumerating each instance of $t$ in each sequence containing $t$.

Although HUS-Span is able to efficiently mine high utility sequential patterns, setting a proper utility threshold is usually difficult for users without sufficient knowledge about a SDB. In view of this, the concept of top-$k$ high utility sequential pattern mining was introduced in [27]. Instead of setting the minimum utility threshold, the users only need to set the value of $k$ to ask the algorithms to identify the top-$k$ high utility sequential patterns in the SDB. Different from high utility sequential pattern mining, the minimum utility threshold is not specified in advance in top-$k$ high utility sequential pattern mining. A general solution is to use a min heap structure to store the top-$k$ high utility sequential patterns found so far and gradually replace the smallest utility sequence in the structure with a new sequence with higher utility. Once the min heap contains $k$ sequences, the sequence with the lowest utility in the min heap can be applied to prune the search space. Thus, how to design an algorithm able to quickly fill the min heap with higher utility sequences is important for top-$k$ high utility sequential pattern mining.

To address the problem of top-$k$ high utility sequential pattern mining, we propose three algorithms, namely TKHUS-Span$_{\text{GDFS}}$, TKHUS-Span$_{\text{BFS}}$ and TKHUS-Span$_{\text{Hybrid}}$ for top-$k$ high utility sequential pattern mining by applying the guided depth-first search (GDFS), best-first search (BFS) and hybrid search of BFS and GDFS into HUS-Span. Similar to [27], the search space is modeled as a lexicographic tree. TKHUS-Span$_{\text{GDFS}}$ traverses the tree by DFS and always visits the child node with highest PEU among its unvisited siblings. As mentioned in [17,28], DFS-based algorithms usually visit mores nodes than BFS-based algorithms. Therefore, TKHUS-Span$_{\text{BFS}}$ is proposed to traverse the search space by BFS according to the PEU of each node. Similar to most BFS-based algorithms, TKHUS-Span$_{\text{BFS}}$ may consume a lot of memory and is not suitable for the cases with limited memory. In view of this, TKHUS-Span$_{\text{Hybrid}}$ is proposed to traverse the search space with memory constraint. Specifically, TKHUS-Span$_{\text{Hybrid}}$ uses BFS to traverse the search space when memory is sufficient and switches to use guided DFS when memory usage almost reaches memory limitation. Hence, TKHUS-Span$_{\text{Hybrid}}$ is able to strike a balance between mining efficiency and memory usage.

To measure the performance of the proposed algorithms, several experiments on various real and synthetic datasets are conducted. The experimental results show that for high utility sequential pattern mining, HUS-Span outperforms the state-of-the-art algorithm USpan [26]. For top-$k$ high utility sequential pattern mining, TKHUS-Span$_{\text{BFS}}$ outperforms other algorithms including TUS [27]. Even so, TKHUS-Span$_{\text{BFS}}$ runs out of memory space on one of the datasets. Fortunately, due to controlling the required memory space, TKHUS-Span$_{\text{Hybrid}}$ performs the best when the memory space is insufficient.

The rest of this paper is organized as follows. Section 2 gives the problem definitions and reviews the related work. Sections 3 and 4 present the proposed algorithms for high utility sequential pattern mining and top-$k$ high utility sequential pattern mining, respectively. Section 5 shows the experimental results, and finally, Sect. 6 concludes this paper.

## 2 Preliminaries

In this section, we first give the formal definition of high utility sequential pattern mining and top-$k$ high utility sequential pattern mining in Sect. 2.1 and then review previous studies on high utility itemset and high utility sequential pattern mining in Sect. 2.2.

### 2.1 Problem definition

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of distinct items. A nonempty subset $X \subseteq I$ is called an itemset and $|X|$ represents the $size$ of $X$. For brevity, the brackets are omitted if an itemset contains only one item. Without loss of generality, all items within an itemset are assumed to be arranged lexicographically. A sequence $\langle e_1 e_2 \ldots e_m \rangle$ is an ordered list of itemsets, where $e_i \subseteq I, \forall i = 1, \ldots, m$. The length $l$ of sequence $\langle e_1 e_2 \ldots e_m \rangle$ is defined as $l \equiv \Sigma_{i=1}^{m} |e_i|$, and the size of sequence $\langle e_1 e_2 \ldots e_m \rangle$ is defined as $m$. A sequence with length $l$ is called an $l$-sequence. Sequence $t = \langle X_1 X_2 \ldots X_k \rangle$ is called a subsequence of another sequence $s = \langle Y_1 Y_2 \ldots Y_m \rangle$, denoted as $t \sqsubseteq s$, if there exists integers $1 \leq j_1 < j_2 < \cdots < j_k \leq m$ such that $X_1 \subseteq Y_{j_1}, X_2 \subseteq Y_{j_2}, \ldots, X_k \subseteq Y_{j_k}$. A sequence database $D$ is a set of tuples $(sid, s)$, where $sid$ is the ID of sequence $s$. Each item $i$ of an itemset $X_j$ in a sequence $s = \langle X_1 X_2 \ldots X_m \rangle$ is associated with a positive real number $q(i, j, s)$, which is called the *internal utility* or quantity of $i$. In the example sequence database given in Table 1a, the internal utility of item $b$ within the second itemset in $s_2$ (i.e., $q(b, 2, s_2)$) is 1.

**Table 1** A sequence database and its profit table

(a) A sequence database

| SID | Sequence | | | | | |
|---|---|---|---|---|---|---|
| $s_1$ | TID | 1 | 2 | 3 | | |
| | Itemset | $b$ | $a, b$ | $c, e$ | | |
| | Internal utility | 3 | 5, 2 | 1, 6 | | |
| $s_2$ | TID | 1 | 2 | 3 | 4 | |
| | Itemset | $c, e$ | $a, b$ | $d$ | $a, d$ | |
| | Internal utility | 5, 4 | 3, 1 | 6 | 2, 8 | |
| $s_3$ | TID | 1 | 2 | 3 | 4 | |
| | Itemset | $c, d$ | $c, d, e$ | $e$ | $a$ | |
| | Internal utility | 2, 1 | 5, 6, 1 | 1 | 2 | |
| $s_4$ | TID | 1 | 2 | 3 | 4 | 5 |
| | Itemset | $a, d$ | $a, b$ | $b, d$ | $a, b$ | $d, e$ |
| | Internal utility | 1, 5 | 3, 2 | 1, 4 | 2, 1 | 6, 1 |
| $s_5$ | TID | 1 | 2 | 3 | 4 | |
| | Itemset | $a, c, e$ | $a, c, e$ | $a, c, e$ | $d$ | |
| | Internal utility | 1, 2, 3 | 40, 1, 1 | 5, 1, 1 | 1 | |

(b) A profit table

| Item | $a$ | $b$ | $c$ | $d$ | $e$ |
|---|---|---|---|---|---|
| External utility | 2 | 10 | 4 | 3 | 1 |

**Definition 1** The *external utility* of item $i$, denoted as $p(i)$, is a positive real number which represents the unit profit or importance of $i$. The external utility of each item $i \in I$ is recorded in a profit table. Table 1b shows the corresponding profit table for the sequence database given in Table 1a.

**Definition 2** The utility of item $i$ within the $j$th itemset of $s$ is defined as $u(i, j, s) = p(i) \times q(i, j, s)$. For example, $u(b, 2, s_2) = p(b) \times q(b, 2, s_2) = 10 \times 1 = 10$.

**Definition 3** The utility of itemset $X$ contained in the $j$th itemset of $s$ is defined as

$$u(X, j, s) = \sum_{\forall i \in X} u(i, j, s). \tag{1}$$

For example, $u(\{a, c\}, 2, s_5) = u(a, 2, s_5) + u(c, 2, s_5) = 2 \times 40 + 4 \times 1 = 84$.

**Definition 4** When $t$ is a subsequence of $s$, where $1 \leq j_1 < j_2 < \cdots < j_k \leq m$ and $X_1 \subseteq Y_{j_1}, X_2 \subseteq Y_{j_2}, \ldots, X_k \subseteq Y_{j_k}$, we say that $s$ has an *instance* of $t$ at position $\langle j_1, j_2, \ldots, j_k \rangle$.

**Definition 5** When $s$ has an $instance$ of $t$ at position $\langle j_1, j_2, \ldots, j_k \rangle$, the utility of the $instance$ of $t$ at position $\langle j_1, j_2, \ldots, j_k \rangle$ is defined as

$$u(t, \langle j_1, j_2, \ldots, j_k \rangle, s) = \sum_{i=1}^{k} u(X_i, j_i, s), \tag{2}$$

where $X_i \subseteq Y_{j_i}$. For example, $u(\langle a\{c, e\}\rangle, \langle 1, 2\rangle, s_5) = u(a, 1, s_5) + u(\{c, e\}, 2, s_5) = 2 + (4+1) = 7$, and $u(\langle a\{c, e\}\rangle, \langle 2, 3\rangle, s_5) = u(a, 2, s_5) + u(\{c, e\}, 3, s_5) = 80 + (4+1) = 85$.

**Definition 6** The utility of sequence $t$ in sequence $s$, denoted as $u(t, s)$, is defined as the maximum of the utilities of all instances of $t$ in $s$. That is,

$$u(t, s) = \max\left\{u\left(t, \langle j_1, j_2, \ldots, j_k \rangle, s\right) | \forall \langle j_1, j_2, \ldots, j_k \rangle : t \sqsubseteq \left\langle Y_{j_1} Y_{j_2} \ldots Y_{j_k} \right\rangle\right\}. \tag{3}$$

For example, $u(\langle a\{c, e\}\rangle, s_5) = \max\{u(\langle a\{c, e\}\rangle, \langle 1, 2\rangle, s_5), u(\langle a\{c, e\}\rangle, \langle 1, 3\rangle, s_5), u(\langle a\{c, e\}\rangle, \langle 2, 3\rangle, s_5)\} = \max\{7, 7, 85\} = 85$.

**Definition 7** The utility of a sequence $t$ in $D$, denoted as $u(t)$, is defined as

$$u(t) = \sum_{\forall s \in D \wedge t \sqsubseteq s} u(t, s). \tag{4}$$

For example, $u(\langle a\{c, e\}\rangle) = u(\langle a\{c, e\}\rangle, s_1) + u(\langle a\{c, e\}\rangle, s_5) = 20 + 85 = 105$.

**Definition 8** A sequence $t$ is called a **high utility sequential pattern** in a sequence database $D$ if $u(t) \geq \xi$, where $\xi$ is a user-specified minimum utility threshold. When $\xi = 100$, $\langle a\{c, e\}\rangle$ is a high utility sequential pattern since $u(\langle a\{c, e\}\rangle) = 105 \geq 100$.

**Problem 1** (*High Utility Sequential Pattern Mining*) Given a sequence database $D$, a profit table and the minimum utility threshold $\xi$, high utility sequential pattern mining is to identify the complete set of high utility sequential patterns in $D$.

**Definition 9** A sequence $t$ is called a **top-$k$ high utility sequential pattern** in a sequence database $D$ if in $D$, there are less than $k$ sequences whose utilities are larger than $u(t)$.

**Problem 2** (*Top-k High Utility Sequential Pattern Mining*) Given a sequence database $D$, a profit table and the desired number of high utility sequential patterns $k$, top-$k$ high utility sequential pattern mining is to identify a set of top-$k$ high utility sequential patterns in $D$, where the set contains $k$ sequences with highest utilities in the database $D$ if there are at least $k$ sequences in $D$; otherwise, it contains all sequences in $D$.

## 2.2 Related work

### *2.2.1 High utility itemset mining*

The concept of high utility itemset mining, which is to mine all itemsets with utilities larger than or equal to a user-specified threshold in a transaction database, was first introduced in [25]. Mining high utility itemsets is more difficult than frequent itemset mining since the downward closure property does not hold for the utilities of itemsets. Fortunately, the downward closure property holds for transaction-weighted utility (TWU), which is an upper bound of the utilities of an itemset and all its super-itemsets. Thus, several algorithms have been proposed to mine high utility itemsets by applying TWU to prune the search space. Generally, these algorithms can be grouped into two classes: the level-wise candidate generation-and-test manner and the pattern-growth manner. In the level-wise candidate generation-and-test manner, Liu et al. [14] proposed a two-phase algorithm with a pruning strategy to efficiently mine high utility itemsets. Li et al. [11] proposed the isolated item discarding strategy to further reduce the number of candidates. However, these algorithms suffer from the problem of multiple database scans. To address this problem, some algorithms based on the pattern-growth manner have been proposed. Ahmed et al. [4] devised three tree structures to facilitate incremental high utility itemset mining. Inspired by FP-tree used in FP-Growth [8], Tseng et al. [21] also proposed a tree structure UP-Tree as well as four pruning strategies to mine high utility itemsets. However, all of these algorithms cannot avoid candidate generation. Recently, two algorithms [12,13] without candidate generation have been proposed to achieve mining efficiency better than the algorithms proposed in [4,21].

Since it is difficult for users to set an appropriate minimum utility threshold if they do not have sufficient knowledge on the transaction database, this limitation brings out the top-$k$ high utility itemset mining problem. To the best of our knowledge, algorithm TKU [23] is the only algorithm for top-$k$ high utility itemset mining. TKU adopted the UP-Tree to maintain the information of top-$k$ high utility itemsets. TKU consists of three phases. In the first phase, UP-tree is constructed. In the second phase, all the candidate top-$k$ high utility itemsets are found. In the third phase, the actual utilities of the candidate itemsets are calculated. By initially setting a minimum utility threshold to 0, after $k$ itemsets are found, the lowest utility among the $k$ itemsets can be used to replace the minimum utility threshold. Therefore, after $k$ itemsets are found, how to raise the minimum utility threshold as soon as possible becomes a major concern for mining efficiency. Accordingly, five strategies are proposed to effectively raise the minimum utility threshold in order to prune the search space during different phases of TKU.

### *2.2.2 High utility sequential pattern mining*

The problem of high utility sequential pattern mining was first addressed in [3]. References [10,26] argued that the definition of utility used in [3] is too specific, and hence adopted "the maximum utility of all occurrences of $t$ in $s$" as the utility of $t$ in $s$. Our work follows the definition of utility used in [10,26]. Due to the absence of downward closure property in sequence utility, SWU, which is an upper bound of the utilities of a sequence and all its supersequence, was applied to prune the search space during the mining process. Here, we give the definition of SWU as below.

**Definition 10** ([3,10,26]) The SWU of a sequence $t$, denoted as $SWU(t)$, is defined as

$$\text{SWU}(t) = \sum_{\forall s \in D \wedge t \sqsubseteq s} u(s, s). \tag{5}$$

For example, $SWU(\langle a\{c, e\}\rangle) = u(s_1, s_1) + u(s_5, s_5) = 70 + 116 = 186$.

**Theorem 1** (Sequence-Weighted Downward Closure Property [3,26]) *Given a sequence $t$ in a sequence database $D$, $SWU(t) \geq SWU(t')$ for all $t \sqsubseteq t'$.*

**Theorem 2** ([3,10]) *Given a sequence $t$ in a sequence database $D$, $SWU(t) \geq u(t)$.*

According to Theorems 1 and 2, we can see that the downward closure property holds in SWU and SWU of a sequence is no less than the utility of the sequence. By using SWU to prune the search space, Ahmed et al. [3] proposed an Apriori-like algorithm *UL* and PrefixSpan-like algorithm *US* to mine high utility sequential patterns by the following two phases. In phase I, the sequences with high SWUs are first found from the sequence database. Then, in phase II, the actual utilities of all high SWUs sequences are computed and hence all high utility sequential patterns can be identified. There are some applications related to high utility sequential pattern mining; for example, UMSP algorithm [18] was proposed to mine high utility mobile sequences, and algorithms with UWAS-tree and IUWAS-tree [2] were designed to mine high utility Web log sequences. Yin et al. gave in [26] a generic problem definition of high utility sequential pattern mining and proposed algorithm USpan to mine high utility sequential patterns. They represented the search space of high utility sequential pattern mining problem as a lexicographic tree and used USpan to mine high utility sequential patterns by traversing the tree in a DFS manner. They also used SWU and proposed a depth pruning strategy to prune the tree for a better efficiency. Although SWU adopted by most of the above algorithms (e.g., UL [3], US [3] and USpan [26]) can prune the search space, they usually suffer from the problem of massive candidate sequences, especially when minimum utility threshold is small. In view of this, we propose two tight utility upper bounds as well as two companion pruning strategies to generate less candidates, thereby achieving efficient mining of high utility sequential patterns.

As in high utility itemset mining, it is difficult for users to select a suitable minimum utility threshold, and thus, Yin et al. [27] proposed algorithm TUS to solve the problem of top-$k$ high utility sequential pattern mining. Similar to USpan, they mined top-$k$ patterns by traversing the lexicographic tree in a DFS way. They proposed a utility metric SPU, identical to the depth pruning strategy in USpan [26], to determine the visiting order of the child nodes of a parent node and to raise the minimum utility threshold as soon as possible. Moreover, they proposed a utility metric SRU to prune the search space.

To address the problem of top-$k$ high utility sequential pattern mining, we first propose algorithm TKHUS-Span$_{\text{GDFS}}$ to mine top-$k$ high utility sequential patterns by using DFS to traverse lexicographic trees with the aid of PEU and RSU. Although TUS and TKHUS-Span$_{\text{GDFS}}$ can mine top-$k$ high utility sequential patterns effectively by DFS, they may not find out the top-$k$ high utility sequential patterns in the early stage and will visit many nodes to complete the mining process [17,28]. In view of this, we propose algorithm TKHUS-Span$_{\text{BFS}}$ by adopting BFS to explore the search space as thoroughly as possible. TKHUS-Span$_{\text{BFS}}$ can raise the minimum utility threshold more effectively than TUS and TKHUS-Span$_{\text{GDFS}}$. In the situation of limited memory space of machines, we also proposed an algorithm with a hybrid search of BFS and GDFS, named TKHUS-Span$_{\text{Hybrid}}$, to balance the mining efficiency and space usage.

## 3 HUS-Span: the proposed algorithm for mining high utility sequential patterns

### 3.1 Lexicographic tree

Similar to SPAM [5] and USpan [26], the search space of the high utility sequential pattern mining problem can be represented as a lexicographic tree. A lexicographic tree is a tree structure with the root labeled with "⟨⟩", and each node $t$ other than the root consists of the following fields: $t.seq$, and $t.uchain$, where $t.seq$ is the sequence represented by $t$ and $t.uchain$ contains the utility-chain (some auxiliary information to facilitate efficient PEU, RSU and utility calculation) of $t$ for search space pruning.

**Definition 11** ([5]) Given an $l$-sequence $t$, if an $(l+1)$-sequence $t'$ is generated by appending a new itemset consisting of a single item to the end of $t$, $t'$ is called an s-extension sequence. On the other hand, $t'$ is called an i-extension sequence if it is formed by inserting an item into the last itemset of $t$. The process of generating an s-extension sequence is called *S-Extension*, and the process of generating an i-extension sequence is called *I-Extension*. For example, $\langle cd \rangle$ is an s-extension sequence of $\langle c \rangle$, and $\langle \{c, e\} \rangle$ is an i-extension sequence of $\langle c \rangle$.

Given a node $t$ in the tree, the sequence of each child node $t'$ of $t$ (i.e., $t'.seq$) is either an i-extension sequence or an s-extension sequence of $t.seq$. Figure 1 shows a part of the lexicographic tree of the sequence database in Table 1a. All the child nodes of a parent node are ordered lexicographically with i-extension sequences before s-extension sequences.

Similar to SPAM [5] and USpan [26], the basic idea of HUS-Span is to traverse the lexicographic tree by DFS. For each visited node $t$, HUS-Span calculates the utility of $t$ and reports $t$ as a high utility sequential pattern if the utility of $t$ is larger than or equal to $\xi$. It is clear that a lot of nodes in a lexicographic tree will not be of enough utility. For better efficiency, some existing algorithms [3,26] adopted SWU to prune a sequence $t$ if $SWU(t)$ is less than $\xi$. However, SWU is usually much larger than the real utilities of $t$ and all supersequences of $t$, thereby severely reducing the pruning effect of SWU-based algorithms. This problem becomes worse, especially when a sequence database consists of a large amount of long sequences. Such problem motivates us to design two novel utility upper bounds, called PEU and RSU, which are tighter than SWU. With the aid of PEU and RSU, we then propose two companion pruning strategies and develop algorithm HUS-Span to efficiently mine high utility sequential patterns using the proposed pruning strategies. In Sect. 3.2, we design a novel data structure, utility-chain, to facilitate efficient calculation of PEUs, RSUs and utilities. The details of PEU and RSU as well as the companion pruning strategies are given in Sect. 3.3. Finally, the details of HUS-Span are described in Sect. 3.4.

### 3.2 Utility-chain structure

Considering two sequences $t = \langle X_1 X_2 \dots X_k \rangle$ and $s = \langle Y_1 Y_2 \dots Y_m \rangle$, where $t \sqsubseteq s$, it is possible that $s$ contains multiple instances of $t$ at different positions. Take sequence $\langle ac \rangle$ for example. Sequence $s_5 = \langle \{a, c, e\}\{a, c, e\}\{a, c, e\}d \rangle$ contains three instances of $\langle ac \rangle$ at positions $\langle 1, 2 \rangle$, $\langle 1, 3 \rangle$, and $\langle 2, 3 \rangle$. Hence, the utility of $\langle ac \rangle$ in $s_5$ (i.e., $u(\langle ac \rangle, s_5)$) should be obtained by calculating the maximum of the utilities of these three instances. Obviously, it is time-consuming to obtain the utility of each instance of $t$ in $s$ since we have to first find the positions of all instances of $t$ in $s$, then compute the utility of each instance, and finally obtain the maximum of the utilities of all instances. This motivates us to design the utility-chain

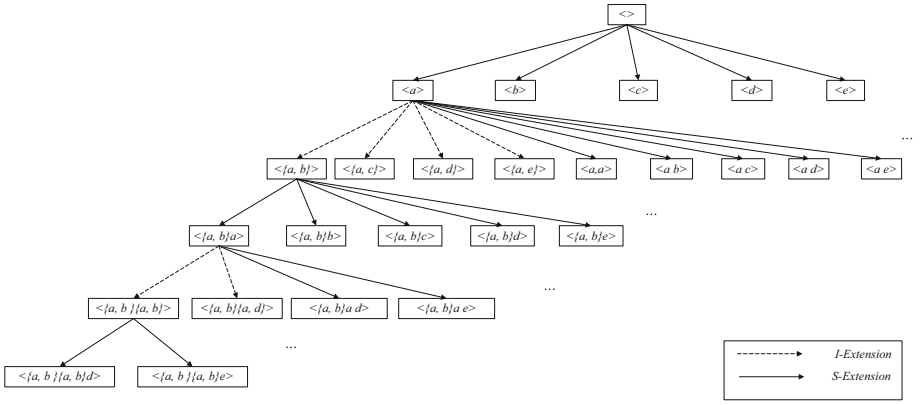


**Fig. 1** Partial lexicographic tree for the example in Table 1a

structure to speed up the calculation of the utilities of all instances of $t$ in $s$ when $s$ contains multiple instances of $t$.

**Definition 12** If $s$ has an instance of $t$ at position $\langle j_1, j_2, \ldots, j_k\rangle$, the last item of $X_k$, say item $i$, is called the *extension item* of $t$, and $j_k$ is called an *extension position* of $t$ in $s$. For example, item $e$ is the extension item of sequence $\langle a\{c, e\}\rangle$, while 2 and 3 are extension positions of $\langle a\{c, e\}\rangle$ in sequence $s_5 = \langle\{a, c, e\}\{a, c, e\}\{a, c, e\}d\rangle$.

Suppose that $t$ is an $l$-sequence. To perform the extension process of $t$ in $s$, we can record each extension position of $t$ in $s$ since it is the qualified position for $t$ to append an item to form $(l + 1)$-sequences. Furthermore, to calculate the utility of each $(l + 1)$-sequence extended from $t$ in $s$, an efficient way is, for each extension position of $t$ in $s$, say $p$, to record the maximum utility of the instances of $t$ at extension position $p$ in $s$.

**Definition 13** The maximum utility of $t$ at extension position $p$ in $s$, denoted as $u(t, p, s)$, is defined as

$$\max\{u(t, \langle j_1, j_2, \ldots, p\rangle, s) | \forall 1 \leq j_1 < j_2 < \cdots < p \leq m : t \sqsubseteq \langle X_{j_1} X_{j_2} \ldots X_p\rangle\}. \quad (6)$$

For example, $u(\langle a\{c, e\}\rangle, 3, s_5) = \max\{u(\langle a\{c, e\}\rangle, \langle 1, 3\rangle, s_5), u(\langle a\{c, e\}\rangle, \langle 2, 3\rangle, s_5)\} = \max\{7, 85\} = 85$.

It is obvious that the utility of $t$ in $s$ (i.e., $u(t, s)$) can be efficiently obtained from the maximum utility of $t$ at each extension position $p$ in $s$ (i.e., $u(t, p, s)$).

**Definition 14** Suppose that $s$ contains an instance of $t$ at position $\langle j_1, j_2, \ldots, j_k\rangle$. The *remaining sequence* of $s$ with respect to extension position $j_k$, denoted as $s/_{(t, j_k)}$, is defined as the subsequence of $s$ from the item after the extension item of such instance to the end of $s$.

**Definition 15** The utility of the remaining sequence $s/_{(t, j_k)}$, denoted as $ru(s/_{(t, j_k)})$, is the sum of the utilities of all items in $s/_{(t, j_k)}$.

For example, $s_5 = \langle\{a, c, e\}\{a, c, e\}\{a, c, e\}d\rangle$ contains three instances of $\langle c\rangle$. The remaining sequences of these three instances are $s_5/_{(\langle c\rangle, 1)} = \langle e\{a, c, e\}\{a, c, e\}d\rangle$, $s_5/_{(\langle c\rangle, 2)} =$

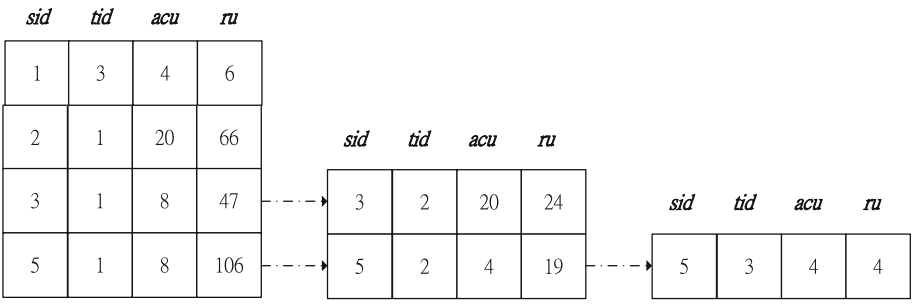


**Fig. 2** Utility-chain of sequence $\langle c \rangle$

$\langle e\{a, c, e\}d \rangle$, and $s_5/_{(\langle c \rangle,3)} = \langle ed \rangle$. In addition, the utilities of these three remaining sequences are $ru(s_5/_{(\langle c \rangle,1)}) = 106$, $ru(s_5/_{(\langle c \rangle,2)}) = 19$ and $ru(s_5/_{(\langle c \rangle,3)}) = 4$, respectively.

Suppose all the extension positions of $t$ in $s$ are $p_1, p_2, \ldots, p_q$, where $p_1 < p_2 < \cdots < p_q$. The utility-list of $t$ in $s$ is a list of $q$ elements, where the $i$th element in the utility-list contains the following fields:

- Field $sid$ is the sequence ID of $s$.
- Field $tid$ is the $i$th extension position $p_i$.
- Field $acu$ is the maximum utility of $t$ at extension position $p_i$ in $s$ (i.e., $u(t, p_i, s)$).
- Field $ru$ is the utility of the remaining sequence $s/_{(t,p_i)}$ (i.e., $ru(s/_{(t,p_i)})$)
- Field $link$ is a pointer pointing to either the ($i$+1)th element when $i < q$ or $null$ when $i = q$.

The utility-chain of a sequence $t$ is a set of utility-lists for all sequences containing $t$.

### *3.2.1 Generating utility-chains of* 1*-sequences*

To construct the utility-chains of all 1-sequences, HUS-Span first scans the sequence database $D$ to accumulate SWU of each 1-sequence and constructs the utility-chains for the 1-sequences with SWUs larger than or equal to $\xi$. Figure 2 shows the utility-chain of sequence $\langle c \rangle$ in the sequence database given in Table 1a. The sequences containing $\langle c \rangle$ are $s_1$, $s_2$, $s_3$ and $s_5$. In sequence $s_1 = \langle b\{ab\}\{ce\} \rangle$, only one instance of $\langle c \rangle$ at extension position 3 exists. Thus, $u(\langle c \rangle, 3, s_1) = 4$, and the remaining utility $ru(s_1/_{(\langle c \rangle,3)}) = u(e, 3, s_1) = 6$. So, the utility-list of sequence $\langle c \rangle$ for $s_1$ contains only one element $(1, 3, 4, 6, null)$. On the other hand, $s_3 = \langle \{c, d\}\{c, d, e\}ea \rangle$ contains two instances of $\langle c \rangle$ at extension positions 1 and 2. The utilities as well as the remaining utilities of these two instances are $u(\langle c \rangle, 1, s_3) = 8$ and $ru(s_3/_{(\langle c \rangle,1)}) = 47$, and $u(\langle c \rangle, 2, s_3) = 20$ and $ru(s_3/_{(\langle c \rangle,2)}) = 24$, respectively. We can see in Fig. 2 that the utility-list of $\langle c \rangle$ for $s_3$ consists of two elements $e1 = (3, 1, 8, 47, e2)$, and $e2 = (3, 2, 20, 24, null)$, where the link of $e1$ pointing to $e2$ indicates that the next extension position after the position 1 is 2.

### *3.2.2 Generating utility-chains of* $l$*-sequences* ($l \geq 2$)

We use the following examples to describe, given an $l$-sequence $t$, how to generate all $t$'s i- or s-extension $(l + 1)$-sequences as well as their utility-chains based on the utility-chains of $t$. In a sequence $s$, items able to form i-extension sequences from $t$ are those items which exist within the $p_i th$ itemset of s and are lexicographically larger than the extension item of $t$, for

all extension positions $p_i$ of $t$ in $s$. Consider sequence $s_3 = \langle\{c, d\}\{c, d, e\}ea\rangle$ consisting of the 1-sequence $\langle c\rangle$. Since there are two instances of $\langle c\rangle$ in $s_3$, the utility-list of $\langle c\rangle$ for $s_3$ consists of two elements. The value of $tid$ field in the first element is set to 1 to indicate that the first extension position of $\langle c\rangle$ in $s_3$ is 1. Similarly, the value of $tid$ of the second element is set to 2. In the first itemset of $s_3$, $d$ is the item after the first occurrence of the extension item of $\langle c\rangle$, and in the second itemset, $d$ and $e$ are items after the second occurrence of the extension item. Thus, we know that items in $s$ able to form i-extension sequences from $\langle c\rangle$ are $d$ and $e$.

Next, we take item $d$ as an example to show how to construct the utility-chain of $\langle\{c, d\}\rangle$. Since item $d$ is within the first and second itemsets of $s_3$, there are two extension positions of $\langle\{c, d\}\rangle$ in $s_3$, and the utility-chain of $\langle\{c, d\}\rangle$ consists of two elements. The first element is of field $tid = 1$ since the first extension position of $\langle\{c, d\}\rangle$ in $s_3$ is 1. Similarly, the second element is of field $tid = 2$. The utility of $u(\langle\{c, d\}\rangle, 1, s_3)$ can be calculated by the utility of the extension item in the first itemset of $s_3$ (i.e., $u(d, 1, s_3)$) plus the $acu$ of the first element of the utility-list of $\langle c\rangle$ in $s_3$. Thus, we have $u(\langle\{c, d\}\rangle, 1, s_3) = u(d, 1, s_3) + u(\langle c\rangle, 1, s_3) = 3 + 8 = 11$, and the value of $u(\langle\{c, d\}\rangle, 1, s_3)$ is stored in the field $acu$ of the first element of the utility-list of $\langle\{c, d\}\rangle$ in $s_3$. The value of $u(\langle\{c, d\}\rangle, 2, s_3)$ is obtained by $u(d, 2, s_3) + u(\langle c\rangle, 2, s_3) = 18 + 20 = 38$, and hence the value of $acu$ of the second element of the utility-list of $\langle\{c, d\}\rangle$ in $s_3$ is 38. Since $\langle\{c, d\}\rangle$ occurs only in $s_3$, the utility-chain of $\langle\{c, d\}\rangle$ consists of only one utility-list and the utility-chain of $\langle\{c, d\}\rangle$ is shown in Fig. 3.

On the other hand, items able to form s-extension sequences from $t$ in $s$ are those items which exist after the $p_1th$ itemset of $s$, where $p_1$ is the first extension position of $t$ in $s$. Take $\langle\{c, d\}\rangle$ as an example. Since the value of $tid$ of the first element of the utility-list of $\langle\{c, d\}\rangle$ for $s_3$ is 1, the items able to form s-extension sequences from $\langle\{c, d\}\rangle$ in $s_3$ are all the items occurring at least once among the itemsets from the second itemset to the last itemset of $s_3$. In this example, they are $a$, $c$, $d$ and $e$. Here, we take item $e$ to show how to construct the utility-list of $\langle\{c, d\}e\rangle$ for $s_3$. With a scan from the first item of the second itemset to the last item of the last itemset, two extension positions of $\langle\{c, d\}e\rangle$ within the second and the third itemsets are found. Therefore, a utility-chain with two elements $e1$ and $e2$ is formed, and the values of the fields $tid$ of $e1$ and $e2$ are set to 2 and 3, respectively. The value of field $acu$ of the first element (i.e., $u(\langle\{c, d\}e\rangle, 2, s_3)$) can be calculated by $u(\langle\{c, d\}\rangle, 1, s_3) + u(e, 2, s_3) = 11 + 1 = 12$. We now consider the second element. From the utility-list of $\langle\{c, d\}\rangle$ for $s_3$, we know that there are two extension positions 1 and 2 for $\langle\{c, d\}\rangle$. Sequence $\langle\{c, d\}e\rangle$ can be formed by appending item $e$ to the extension item of $\langle\{c, d\}\rangle$ occurring in the first and the second itemsets. Thus, $u(\langle\{c, d\}e\rangle, 3, s_3) = \max\{u(\langle\{c, d\}\rangle, 1, s_3), u(\langle\{c, d\}\rangle, 2, s_3)\} + u(e, 3, s_3) = \max\{11, 38\} + 1 = 38 + 1 = 39$. Finally, the utility-chain of $\langle\{c, d\}e\rangle$ is shown in Fig. 4.

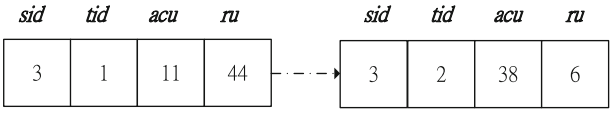


**Fig. 3** Utility-chain of $\langle\{c, d\}\rangle$

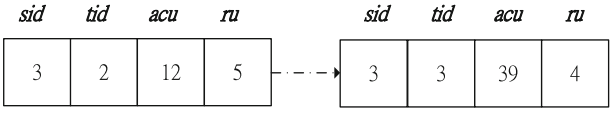


**Fig. 4** Utility-chain of sequence $\langle\{c, d\}e\rangle$

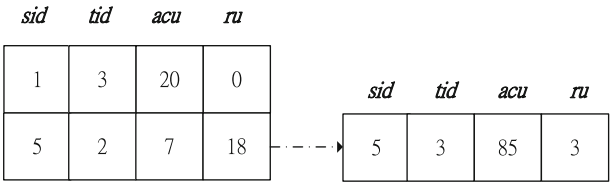


**Fig. 5** Utility-chain of sequence $\langle a\{c, e\}\rangle$

### 3.3 The proposed pruning strategies

#### *3.3.1 The prefix extension utility strategy*

**Definition 16** The PEU of sequence $t$ in sequence $s$ at position $p$, denoted as $PEU(t, p, s)$, is defined as:

$$PEU(t, p, s) = \begin{cases} u(t, p, s) + ru(s/_{(t,p)}) \text{ if } ru(s/_{(t,p)}) > 0, \\ 0 \qquad\qquad\qquad\quad \text{otherwise.} \end{cases} \tag{7}$$

The PEU of sequence $t$ in sequence $s$, denoted as $PEU(t, s)$, is defined as:

$$PEU(t, s) = \max\{PEU(t, p, s) | \forall \text{extension position } p \text{ of } t \text{ in } s\}. \tag{8}$$

**Definition 17** The PEU of sequence $t$ in $D$, denoted as $PEU(t)$, is defined as:

$$PEU(t) = \sum_{\forall s \in D \wedge t \sqsubseteq s} PEU(t, s). \tag{9}$$

*Example 1* Consider the sequence $t = \langle a\{c, e\}\rangle$. Since $t$ is a subsequence of $s_1 = \langle b\{a, b\}\{c, e\}\rangle$ and $s_5 = \langle\{a, c, e\}\{a, c, e\}\{a, c, e\}d\rangle$, the utility-chain of $t$ is shown in Fig. 5. The value of $ru$ of the first element in the utility-list for $s_1$ is 0, meaning that it is not possible to form any i- or s-extension sequence of $t$ in $s_1$. Thus, $PEU(t, s_1)$ is 0. On the other hand, the utility-list for $s_5$ consists of two elements, and their remaining utilities (i.e., $ru$) are larger than 0. It means that in $s_5$, there are some items able to be inserted or appended to the end of $t$ to form i- or s-extension sequences. According to the definition of PEU, we have $PEU(t, s_5) = \max\{7 + 18, 85 + 3\} = \max\{25, 88\} = 88$, meaning that the upper bound of the utility of any descendant sequence of $t$ in $s_5$ is 88. Finally, the upper bound of the utility of any descendant sequence of $t$ is $PEU(t) = PEU(t, s_1) + PEU(t, s_5) = 88$.

**Theorem 3** *Given a sequence $t$, for each sequence $t'$ where $t$ is a prefix of $t'$ (i.e., $t'$ is a descendant of $t$), $u(t') \leq PEU(t)$.*

*Proof* Let $S_t \subseteq D$ be the set of sequences containing $t$. For each $s \in S_t$, let $EP_{t,s}$ be the set of each extension position of $t$ in $s$ whose utility of the remaining sequence is larger than 0 (i.e., $EP_{t,s} = \{p | ru(s/_{(t,p)}) > 0\}$. Since $t$ is a prefix of $t'$, we assume that $t' = t \cdot t''$ where $|t''| > 0$. Consider an instance of $t'$ in $s$ and let $p$ be the extension position of the instance of $t$ in $s$. The utility of this instance in $s$ consists of two parts: (1) the utility of an instance of $t$ at extension position $p$ in $s$ and (2) the utility of an instance of $t''$ where the first item of such instance is after $p$. The first part can be obtained from field $acu$ of the corresponding element in $t$'s utility-chain. As to the second part, it is clear that the instance of $t''$ must be a subsequence of the remaining sequence $s/_{(t,p)}$. Thus, the upper bound of the utility of the

instance of $t''$ is the utility of the remaining sequence $s/_{(t,p)}$, which is able to be obtained from field $ru$ of the corresponding element in $t$'s utility-chain. Thus, the upper bound of an instance of $t'$, where the extension position of the corresponding instance of $t$ in $s$ is $p$, is $u(t, p, s)+ru(s/_{(t,p)})$. Since $PEU(t, s)$ is defined as $\max_{\forall p \in EP_{t,s}}\{(u(t, p, s)+ru(s/_{(t,p)})\}$, we have $u(t', s) \leq PEU(t, s)$ and $u(t') \leq PEU(t)$. □

According to Theorem 3, $PEU(t)$ is an upper bound of the utilities of any descendants of $t$. Therefore, each descendant node of $t$ can be safely pruned without affecting the mining result when $PEU(t) < \xi$.

### *3.3.2 The reduced sequence utility strategy*

**Definition 18** Consider a sequence $t$ and let $\alpha$ be the sequence able to generate $t$ by one $I$- or $S$-$Extension$. The RSU of $t$ in sequence $s$, denoted as $RSU(t, s)$, is defined as:

$$RSU(t, s) = \begin{cases} PEU(\alpha, s) : t \sqsubseteq s \ \wedge \ \alpha \sqsubseteq s \\ 0 \qquad\qquad\quad : \text{otherwise} \end{cases}, \tag{10}$$

where $p$ is an extension position of $\alpha$ in $s$. The RSU of a sequence $t$, denoted as $RSU(t)$, is defined as:

$$RSU(t) = \sum_{\forall s \in D} RSU(t, s). \tag{11}$$

*Example 2* Consider the sequence $\langle\{c, d\}e\rangle$. In the database given in Table 1a, only sequence $s_3 = \langle\{c, d\}\{c, d, e\}ea\rangle$ contains $\langle\{c, d\}e\rangle$. After scanning $s_3$ from the item after the extension position of $\langle\{c, d\}e\rangle$ in $s_3$ (i.e., the first item of the third itemset of $s_3$) to the last item of the last itemset, we can find that only items $a$, and $e$ are able to form $s$-extension sequences of $\langle\{c, d\}e\rangle$ (i.e., $\langle\{c, d\}ea\rangle$ and $\langle\{c, d\}ee\rangle$). According to Definition 18, we have $RSU(\langle\{c, d\}ee\rangle, s_3) = \max\{12 + 5, 39 + 4\} = \max\{17, 43\} = 43$. $\langle\{c, d\}ee\rangle$ occurs only in $s_3$, and thus, we have $RSU(\langle\{c, d\}ee\rangle) = RSU(\langle\{c, d\}ee\rangle, s_3) = 43$.

**Theorem 4** *Given a sequence $t$, for each sequence $t'$ where either $t$ is a prefix of $t'$ or $t' = t$, $u(t') \leq RSU(t)$.*

*Proof* Let $\alpha$ be the sequence able to generate $t$ by one $I$- or $S$-$Extension$. In the lexicographic tree, it is clear that $\alpha$ is the parent node of $t$ and $t'$ is a descendant of $t$. Let $S_t \subseteq D$ be the set of sequences containing $t$. Given a sequence $s \in S_t$, according to Definition 18, we have $RSU(t, s) = PEU(\alpha, s)$. In addition, based on the proof of Theorem 3, we know that for each sequence $t'$ where $\alpha$ is a prefix of $t'$ (i.e., either $t$ is a prefix of $t'$ or $t' = t$), $u(t', s) \leq PEU(\alpha, s)$. Thus, we have $u(t', s) \leq PEU(\alpha, s) = RSU(t, s)$ and $u(t') \leq RSU(t)$. □

According to Theorem 4, $RSU(t)$ is an upper bound of the utilities of $t$ and each descendant of $t$, and thus, the sub-tree rooted by $t$ can be safely pruned without affecting the mining result when $RSU(t) < \xi$.

## 3.4 HUS-span algorithm

As shown in Algorithm 1, HUS-Span traverses a lexicographic tree by DFS. When visiting a node $t$, HUS-Span first checks whether each descendant of $t$ is unable to be a high utility sequential pattern by the PEU pruning strategy (line 1). If so, HUS-Span backtracks to the

parent of $t$. Otherwise, HUS-Span scans $t$-projected DB (line 3) and puts i- and s-extension items into $ilist$ or $slist$, respectively (lines 4 to 5). Then, HUS-Span removes some items from $ilist$ and $slist$ according to the RSU pruning strategy (line 6). For each item $i$ in $ilist$, HUS-Span performs the I-Extension operation to generate each child node $t'$ (line 8), constructs the utility-chain $t'.uc$ of $t'$ and outputs $t'$ as a high utility sequential pattern if the utility of $t'$ is larger than or equal to the minimum utility threshold $\xi$ (lines 10 to 11). Finally, HUS-Span recursively expands node $t'$ (line 12) in a similar manner. Similarly, HUS-Span performs the above procedure to handle each s-extension item $i$ in $slist$ (lines 13–18).

**Algorithm 1**: HUS-Span($t$, $t.uc$)

**Input**: A sequence database $D$, the minimum utility threshold $\xi$, and a sequence $t$ with utility-chain $t.uc$
**Output**: The set of high utility sequential patterns

```
1  if PEU(t) < ξ then
2      return
3  scan t-projected DB once to;
4      1. put i-extension items into ilist;
5      2. put s-extension items into slist;
6  delete low RSU items from ilist and slist;
7  foreach item i ∈ ilist do
8      t' ← I − Extension(t, i);
9      construct t'.uc;
10     if u(t') ≥ ξ then
11         output t';
12     HUS-Span(t', t'.uc);
13 foreach item i ∈ slist do
14     t' ← S − Extension(t, i);
15     construct t'.uc;
16     if u(t') ≥ ξ then
17         output t';
18     HUS-Span(t', t'.uc);
```

### 3.5 Implementation details

Similar to USpan, each sequence in a sequence database is represented as a utility matrix stored in memory. The utility matrix of a sequence $s$ is indexed by item and TID, where the $(i, j)$ entry represents the utility $u(i, j, s)$ of item $i$ within the $j$th itemset of $s$. In addition, for each sequence, we also use a remaining utility matrix to record the utility of remaining sequence after each item in the sequence, where the $(i, j)$ entry represents the utility of the remaining sequence after item $i$ of the $j$th itemset. Table 2a, b represent the utility matrix and remaining utility matrix of $s_1$, respectively.

For sequences $\alpha$ and $t$, where $t$ is an i- or s-extension sequence of $\alpha$, it is intuitive that $PEU(t)$ and the utility $u(t)$ of $t$ can be obtained by first (1) building the utility-chain of $t$ based on the utility-chain of $\alpha$ and then (2) scanning the utility-chain of $t$. To reduce the execution time, when obtaining the $ru$ of each element, say element $j$, in the utility-chain of $t$, we can directly retrieve $ru$ from the $(i, p_j)$ entry of the remaining utility matrix without computation, where $i$ is an extension item of $t$ and $p_j$ is the $tid$ of element $j$. In addition, $PEU(t)$ and utility $u(t)$ are stored together with sequence of $t$ for better efficiency.

**Table 2** The sequence representation of $s_1$

| TID | 1 | 2 | 3 |
|---|---|---|---|
| (a) The utility matrix of $s_1$ | | | |
| $a$ | 0 | 10 | 0 |
| $b$ | 30 | 20 | 0 |
| $c$ | 0 | 0 | 4 |
| $e$ | 0 | 0 | 6 |
| (b) The remaining utility matrix of $s_1$ | | | |
| $a$ | 70 | 30 | 10 |
| $b$ | 40 | 10 | 10 |
| $c$ | 40 | 10 | 6 |
| $e$ | 40 | 10 | 0 |

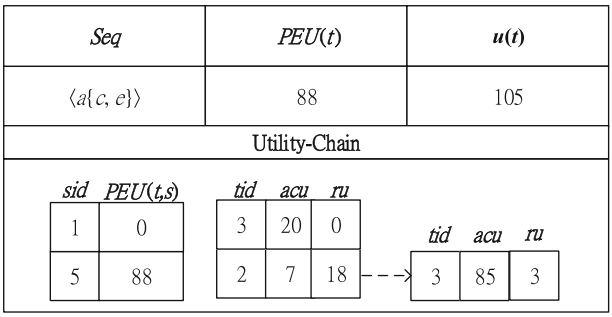


**Fig. 6** Utility-chain implementation of sequence $t = \langle a\{c, e\}\rangle$

For a sequence $t$ whose $PEU(t) \geq \xi$, HUS-Span needs to scan $t$-projected DB to find the i- and s-extension sequences whose $RSUs$ are larger than or equal to $\xi$ and computes the utility of each resultant extension sequence. Assume that $s \in D$, and let $t'$ be an extension sequence of $t$, where $t' \sqsubseteq s$. To compute $RSU(t', s)$, during the $t$-projected sequence scan with respect to $s$, an intuitive way is to scan all elements in the utility-list of $t$ in $s$. According to Definition 18, we know that $RSU(t', s) = PEU(t, s)$. Thus, we insert $PEU(t, s)$ into the utility-list of $t$ in $s$ to avoid scanning all the elements in the utility-list of $t$ in $s$. In this way, the computation of $RSU(t', s)$ is accelerated during $t$-projected DB scan. Furthermore, instead of keeping $sid$ of $s$ in each element in the utility-list of $t$ in $s$, the field $sid$ of $s$ is only kept once in the utility-list of $t$ in $s$ for saving space. Figure 6 shows the implementation of the utility-chain of sequence $t = \langle a\{c, e\}\rangle$ (Fig. 5) with optimization.

## 4 TKHUS-Span: the proposed algorithm for mining top-$k$ high utility sequential patterns

To mine top-$k$ high utility sequential patterns in a sequence database $D$, we use a min heap structure $topkL$ with size $k$ to maintain the $k$ highest utility sequences found during the mining process. Each element in $topkL$ consists of two fields: sequence and its utility. Let $topkL.lowest.s$ be the sequence with the lowest utility among all sequences in $topkL$ and $topkL.lowest.u$ be the utility of $topkL.lowest.s$. The basic idea of the proposed top-$k$

high utility sequential pattern mining algorithm, TKHUS-Span, is as follows. The minimum utility threshold $\xi$ is initialized to -1. Similar to HUS-Span, TKHUS-Span traverses the nodes of a lexicographic tree. If $topkL$ is not full, when finding a sequence $t$, TKHUS-Span inserts sequence $t$ as well as its utility $u(t)$ into $topkL$; otherwise, if $u(t) > topkL.lowest.u$, TKHUS-Span will remove the sequence $topkL.lowest.s$ and its utility $topkL.lowest.u$ from $topkL$ and insert $t$ as well as $u(t)$ into $topkL$, and thus, $\xi$ is updated to $topkL.lowest.u$ after $topkL$ is updated. After all nodes in the lexicographic tree have been either visited or pruned by TKHUS-Span, the sequences in $topkL$ are the top-$k$ high utility sequential patterns.

In high utility sequential pattern mining, the order to traverse a lexicographic tree does not affect the mining efficiency since all nodes should be visited or pruned. Thus, DFS is usually applied to traverse the lexicographic tree. However, in top-$k$ high utility sequential pattern mining, the traversal order becomes important since a good traversal order may lead TKHUS-Span to identify top-$k$ high utility sequential patterns earlier, thereby making TKHUS-Span able to prune more nodes with insufficient utilities. Therefore, we propose three search strategies, namely guided depth-first search (guided DFS), best-first search (BFS) and hybrid search of BFS and GDFS, in the following subsections.

### 4.1 TKHUS-Span with guided depth-first search strategy

Algorithm 2 illustrates the proposed algorithm TKHUS-Span$_{\text{GDFS}}$ for mining top-$k$ high utility sequential patterns by the guided DFS strategy. In the beginning of mining process,

**Algorithm 2**: TKHUS-Span$_{\text{GDFS}}$

**Input**: A sequence database $D$
**Output**: Top-$k$ high utility sequential patterns in $D$
construct a min heap $topkL$ of size $k$;
set minimum utility threshold $\xi$ to -1;
scan $D$ once to:;
    put s-extension items into $slist$;
//process 1-sequences
**foreach** *item* $i \in slist$ **do**
    CalUtility($\langle i \rangle$);
sort each sequence in $slist$ according to its PEU in descending order;
**foreach** *sequence* $t' \in slist$ **do**
    GDFS($t', t'.uc, topkL$);
output all sequences in $topkL$;

**Function** `CalUtility(`$t'$`)`

**Input**: A sequence $t'$
construct $t'.uc$;
**if** $IsFull(topkL)$ **then**
    **if** $u(t') > topkL.lowest.u$ **then**
        remove $topkL.lowest.s$ and $topkL.lowest.u$ from $topkL$;
        insert $(t', u(t'))$ into $topkL$;
        set $\xi$ to $topkL.lowest.u$;
**else**
    insert $(t', u(t'))$ into $topkL$;
    set $\xi$ to $topkL.lowest.u$ when $IsFull(topkL) = true$;

**Function** GDFS$(t, t.uc, topkL)$

**Input**: A sequence $t$, the utility-chain $t.uc$ of $t$, and a min heap $topkL$
**if** *IsFull(topkL) and* $PEU(t) \leq \xi$ **then**
  **return**
scan $t$-projected DB once to:;
  1. put i-extension items into $ilist$;
  2. put s-extension items into $slist$;
**foreach** *item* $i \in ilist$ **do**
  sequence $t' \leftarrow I - Extension(t, i)$;
  **if** *IsFull(topkL) and* $RSU(t') \leq \xi$ **then**
    Remove $i$ from $ilist$;
  **else**
    CalUtility($t'$);
**foreach** *item* $i \in slist$ **do**
  sequence $t' \leftarrow S - Extension(t, i)$;
  **if** *IsFull(topkL) and* $RSU(t') \leq \xi$ **then**
    Remove $i$ from $slist$;
  **else**
    CalUtility($t'$);
put all sequences in $ilist$ and $slist$ into $clist$;
sort each sequence in $clist$ according to its PEU in descending order;
**foreach** *sequence* $t' \in clist$ **do**
  GDFS($t', t'.uc, topkL$);

since no sequence exists in $topkL$, a database scan is performed to construct the utility-chain $t.uc$ of each 1-sequence $t$ by calling Function $CalUtility$. During the construction of the utility-chain of each sequence, the utility $u(t)$ and $PEU(t)$ of each sequence are also computed. After the first database scan, these 1-sequences as well as their utilities are inserted into $topkL$. Once there are $k$ sequences in $topkL$, TKHUS-Span$_{\text{GDFS}}$ updates $\xi$ to $topkL.lowest.u$ and employs the PEU and RSU pruning strategies to prune some nodes during traversing the lexicographic tree.

When determining the visiting order of all the child nodes of a parent node, TKHUS-Span$_{\text{GDFS}}$ first visits the child with the highest PEU among its unvisited siblings since we argue that a node with high PEU is of high likelihood to be of high utility. Our strategy is called guided DFS since we use PEU as the hint to guide the traversal order of DFS. After visiting a parent node $t$, TKHUS-Span$_{\text{GDFS}}$ calls function GDFS and takes the child node with the highest PEU among all $t$'s unvisited children as input to recursively traverse the sub-tree rooted by such child node in DFS. Once function GDFS terminates, TKHUS-Span$_{\text{GDFS}}$ outputs the sequences in $topkL$ as the top-$k$ high utility sequential patterns.

### 4.2 TKHUS-Span with best-first search strategy

Although being able to mine top-$k$ high utility sequential patterns, TKHUS-Span$_{\text{GDFS}}$ may fill $topkL$ with sequential patterns whose utilities are not high enough from the whole database's perspective in the early stage of mining process. This may make TKHUS-Span$_{\text{GDFS}}$ not able to prune some nodes which should be pruned indeed. In view of this, we propose algorithm TKHUS-Span$_{\text{BFS}}$, which uses a best-first search (BFS) strategy to traverse the

lexicographic tree in order to rapidly find high utility sequential patterns from the whole database's perspective. The idea of TKHUS-Span$_{\text{BFS}}$ is as follows. When determining the next node to be visited, different from the TKHUS-Span$_{\text{GDFS}}$ which always chooses the child node with the highest PEU to visit, TKHUS-Span$_{\text{BFS}}$ expands the node with highest PEU among all unvisited nodes found so far. To do so, TKHUS-Span$_{\text{BFS}}$ uses a max heap structure $BFSqueue$ to store the unvisited nodes found so far.

**Algorithm 5**: TKHUS-Span$_{\text{BFS}}$

```
   Input: A sequence database D
   Output: Top-k high utility sequential patterns in D
 1 construct a min heap topkL of size k;
 2 construct a max heap BFSqueue;
 3 set minimum utility threshold ξ to -1;
 4 scan D once to;
 5   put s-extension items into slist;
   //process 1-sequences
 6 foreach item i ∈ slist do
 7   CalUtility(⟨i⟩);
 8   insert (⟨i⟩, PEU(⟨i⟩), ⟨i⟩.uc) into BFSqueue;
 9 while IsEmpty(BFSqueue)=false do
10   if IsFull(topkL) and BFSqueue.highest.PEU ≤ ξ then
11     break;
12   BFS(BFSqueue, topkL);
13 output all sequences in topkL;
```

In $BFSqueue$, each node consists of three fields: sequence $t$, its PEU $PEU(t)$ and its utility-chain $t.uc$. Let $BFSqueue.higest.s$ be the sequence of the highest PEU among all nodes in $BFSqueue$ and $BFSqueue.highest.PEU$ be the PEU of $BFSqueue.highest.s$. Algorithm 3 shows the procedure of TKHUS-Span$_{\text{BFS}}$. The procedure of visiting the root is the same as that of TKHUS-Span$_{\text{GDFS}}$. After visiting the root, TKHUS-Span$_{\text{BFS}}$ puts all child nodes of the root into $BFSqueue$ and visits the nodes with the highest PEU in $BFSqueue$ (i.e., $BFSqueue.highest.s$) by calling function BFS. The main purpose of function BFS is to pick the node $BFSqueue.highest.s$ in $BFSqueue$ to visit, and to put the child nodes of $BFSqueue.highest.s$ with their PEU and utility-chain into $BFSqueue$. Once (1) $BFSqueue$ gets empty or (2) $BFSqueue.highest.PEU \leq topkL.lowest.u$ when $topkL$ is full, TKHUS-Span$_{\text{BFS}}$ terminates immediately and outputs the sequences in $topkL$ as the top-$k$ high utility sequential patterns.

### 4.3 TKHUS-Span with hybrid search strategy

Although TKHUS-Span$_{\text{BFS}}$ can explore the lexicographic tree more efficiently than TKHUS-Span$_{\text{GDFS}}$, it suffers from the drawback of high memory usage in $BFSqueue$. To limit the memory usage, the maximal size of $BFSqueue$ is set. In the beginning, TKHUS-Span$_{\text{Hybrid}}$ traverses the lexicographic tree by BFS. When $BFSqueue$ gets full, TKHUS-Span$_{\text{Hybrid}}$ starts to visit the nodes in $BFSqueue$ by the guided DFS until the number of nodes in $BFSqueue$ is much smaller than the maximal size of $BFSqueue$. When the number of nodes in $BFSqueue$ gets small enough, TKHUS-Span$_{\text{Hybrid}}$ starts to traverse the lexicographic tree by BFS until $BFSqueue$ gets full again. Since taking BFS and GDFS

**Function** BFS ($BFSqueue, topkL$)

**Input**: A min heap $topkL$, and a max heap $BFSqueue$.

$t \leftarrow BFSqueue.highest.s$;
$t.uc \leftarrow BFSqueue.highest.uc$;
scan $t$-projected DB once to:;
    1. put i-extension items into $ilist$;
    2. put s-extension items into $slist$;
delete $BFSqueue.highest.s$, $BFSqueue.highest.PEU$ and $BFSqueue.highest.uc$ from $BFSqueue$;
**foreach** *item* $i \in ilist$ **do**
    sequence $t' \leftarrow I - Extension(t, i)$;
    **if** *IsFull(topkL) and* $RSU(t') \leq \xi$ **then**
        Remove $i$ from $ilist$;
    **else**
        CalUtility($t'$);
        insert ($t'$, $PEU(t')$, $t'.uc$) into $BFSqueue$;
**foreach** *item* $i \in slist$ **do**
    sequence $t' \leftarrow S - Extension(t, i)$;
    **if** *IsFull(topkL) and* $RSU(t') \leq \xi$ **then**
        Remove $i$ from $slist$;
    **else**
        CalUtility($t'$);
        insert ($t'$, $PEU(t')$, $t'.uc$) into $BFSqueue$;

in turn to traverse the lexicographic tree, TKHUS-Span$_{\text{Hybrid}}$ is able to take the relative advantages of TKHUS-Span$_{\text{BFS}}$ and TKHUS-Span$_{\text{GDFS}}$ to strike a balance between mining efficiency and memory usage.

## 5 Experimental evaluation

In this section, the performance of the proposed algorithms is evaluated. The experiments were performed on a computer with a 3.4 GHz Intel Core i7 CPU, 12 GB memory and running on Windows 7 × 64 operating system. All proposed algorithms were implemented in C++ using Mingw-w64 compiler. USpan and TUS are also implemented for comparison purposes. Two synthetic and six real datasets are used for experiments. The synthetic datasets D50C10T5N0.5S4I2.5 and D100C8T4N10S6I2.5 are generated by the IBM data generator [19]. Real datasets Foodmart and Tafeng are grocery shopping datasets, where Foodmart is acquired from Microsoft SQL Server 2008 database [15] and Tafeng is from AIIA Lab [20]. Real datasets Msnbc and Gene are obtained from the UC Irvine Machine Learning Repository [6], where Msnbc is the Web click stream data from the IIS log for msnbc.com on September, 28, 1999, and Gene is the primate splice-junction gene sequences data collected from Genbank 64.1. Real datasets Music and Movies are the customer ratings of the 100 most popular music and movies products, respectively, in Amazon.com [1]. The parameters settings of the synthetic datasets and the statistics of the real datasets are summarized in Table 3a, while the meaning of the parameters and statistics are expressed in Table 3b.

Datasets Foodmart, Tafeng, Music and Movies are of the utility of each item (i.e., unit profit times quantity in Foodmart and Tafeng, and customer rating in Music and Movies)

**Table 3** Characteristics of synthetic and real datasets

| Dataset | *D* | *C* | *T* | N | Type (domain) |
|---|---|---|---|---|---|
| (a) Statistics and parameters of synthetic and real datasets | | | | | |
| D10C8T4N0.1S4I2.5 | 10K | 8 | 4 | 100 | Synthetic |
| D100C8T4N10S6I2.5 | 100K | 8 | 4 | 10000 | Synthetic |
| Foodmart | 8842 | 6.59 | 4.63 | 1559 | Real (shopping) |
| Tafeng | 32266 | 3.71 | 6.84 | 23812 | Real (shopping) |
| Msnbc | 989818 | 4.74713 | 1 | 17 | Real (web log) |
| Gene | 3190 | 60 | 1 | 8 | Real (bioinformatics) |
| Music | 32778 | 1.13387 | 4.0131 | 100 | Real (product review) |
| Movies | 11528 | 1.04754 | 7.1291 | 100 | Real (product review) |
| Statistics/parameter | Description | | | | |
| (b) Statistics and parameter description | | | | | |
| *D* | Number of sequences | | | | |
| *C* | Average number of itemsets per sequence | | | | |
| *T* | Average number of items per itemset | | | | |
| *N* | Number of distinct items | | | | |

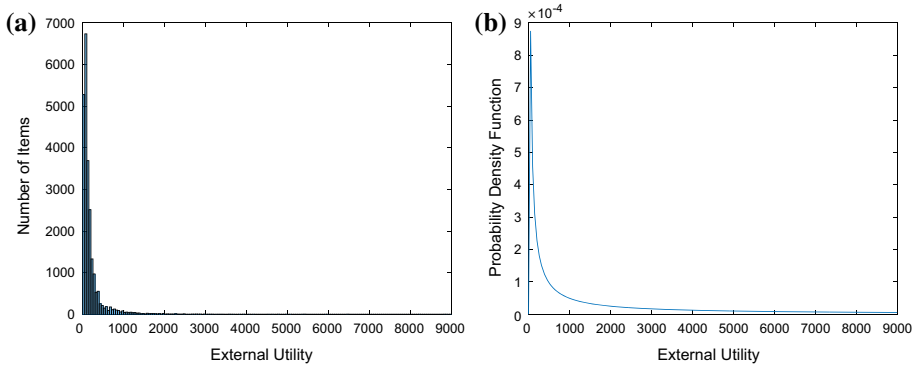


**Fig. 7** External utility distribution. **a** External utility distribution on Tafeng. **b** Log-normal distribution with $(\mu, \sigma) = (8, 8)$

in the original datasets. For other datasets, similar to [26], the internal utility of each item within each itemset in each sequence is randomly generated ranging from 1 to 10. Most studies on high utility pattern mining [26,27] use log-normal distributions to generate external utility for each item on those datasets without utility information according to the observation that most items are of low external utility in most real datasets. Figure 7a shows the distribution of the external utility in dataset Tafeng. As shown in Fig. 7b, the PDF of log-normal distribution with parameter $(\mu, \sigma) = (8, 8)$ is quite close to the distribution of the external utility in dataset Tafeng. Thus, we use the log-normal distribution with parameter $(\mu, \sigma) = (8, 8)$ to generate external utility for the datasets without external utility.

**Table 4** Preprocessing time

| Dataset | Time (s) | Dataset | Time (s) |
|---|---|---|---|
| D10C8T4N0.1S4I2.5 | 0.702 | D100C8T4N10S6I2.5 | 118.556 |
| Foodmart | 9.019 | Tafeng | 127.623 |
| Msnbc | 31.766 | Gene | 0.466 |
| Music | 72.923 | Movie | 2.074 |

## 5.1 Experimental results of high utility sequential pattern mining

Similar to [26], the *preprocessing time* is defined as the time period of loading each sequence from disk and transforming it into the matrices. As given in Table 4, USpan and HUS-Span are of the same preprocessing time for each dataset. In the following subsections, we use *execution time* as the performance metric, where execution time is defined as the time period elapsing from the time that all sequences have been loaded into memory to the time that all high utility sequential patterns have been found.

### *5.1.1 Effect of utility threshold*

This subsection shows the experimental results of HUS-Span and USpan for high utility sequential pattern mining on each dataset. The minimum utility threshold ($\xi$) is determined as a ratio ($\gamma$) times the utility of the whole database, i.e., $\xi = \gamma \times \sum_{\forall s \in D} u(s)$. Figure 8 shows the execution time on each dataset under varied utility threshold, and Table 5 shows the numbers of candidates generated by USpan and HUS-Span. We can observe from Fig. 8 that HUS-Span generally outperforms USpan on each dataset, and the execution time of both algorithms increases when the utility threshold becomes smaller. Figure 8a, b shows the execution time on the synthetic dense dataset D10C8T4N0.1S4I2.5 and the synthetic sparse dataset D100C8T4N10S6I2.5, while Fig. 8e, f shows the execution time on real dense datasets, Msnbc and Gene, under different utility thresholds. We can observe from Figure 8a, b, e, f that HUS-Span is of the best performance. On datasets D10C8T4N0.1S4I2.5, Msnbc, and Gene, HUS-Span runs faster than USpan when the utility threshold becomes smaller due to the reason that HUS-Span is able to generate less candidates than USpan does as given in Table 5. Furthermore, although the numbers of candidate sequences generated by USpan and HUS-Span on the sparse dataset D100C8T4N10S6I2.5 are close, HUS-Span still runs faster than USpan since with the aid of PEU, HUS-Span can greatly reduce the number of projected DB scans than USpan.

Figure 8c, d show the execution time of HUS-Span and USpan on real shopping datasets, Foodmart and Tafeng, respectively. As shown in Fig. 8c, HUS-Span runs about 1.7 times faster than USpan on dataset Foodmart. On the other hand, on the sparsest dataset Tafeng, whose number of distinct items is much larger than that of Foodmart, HUS-Span still runs faster than USpan though the difference of the performance between HUS-Span and USpan is not significant as shown in Fig. 8d. When the number of distinct items and sequences gets larger, HUS-Span and USpan take more time to generate a huge number of nodes in the lexicographic tree. Fortunately, since our pruning strategies are more effective than those used in USpan, as shown in Table 5, HUS-Span is still able to prune more nodes than USpan does. Thus, HUS-Span still runs faster than USpan on dataset Tafeng.

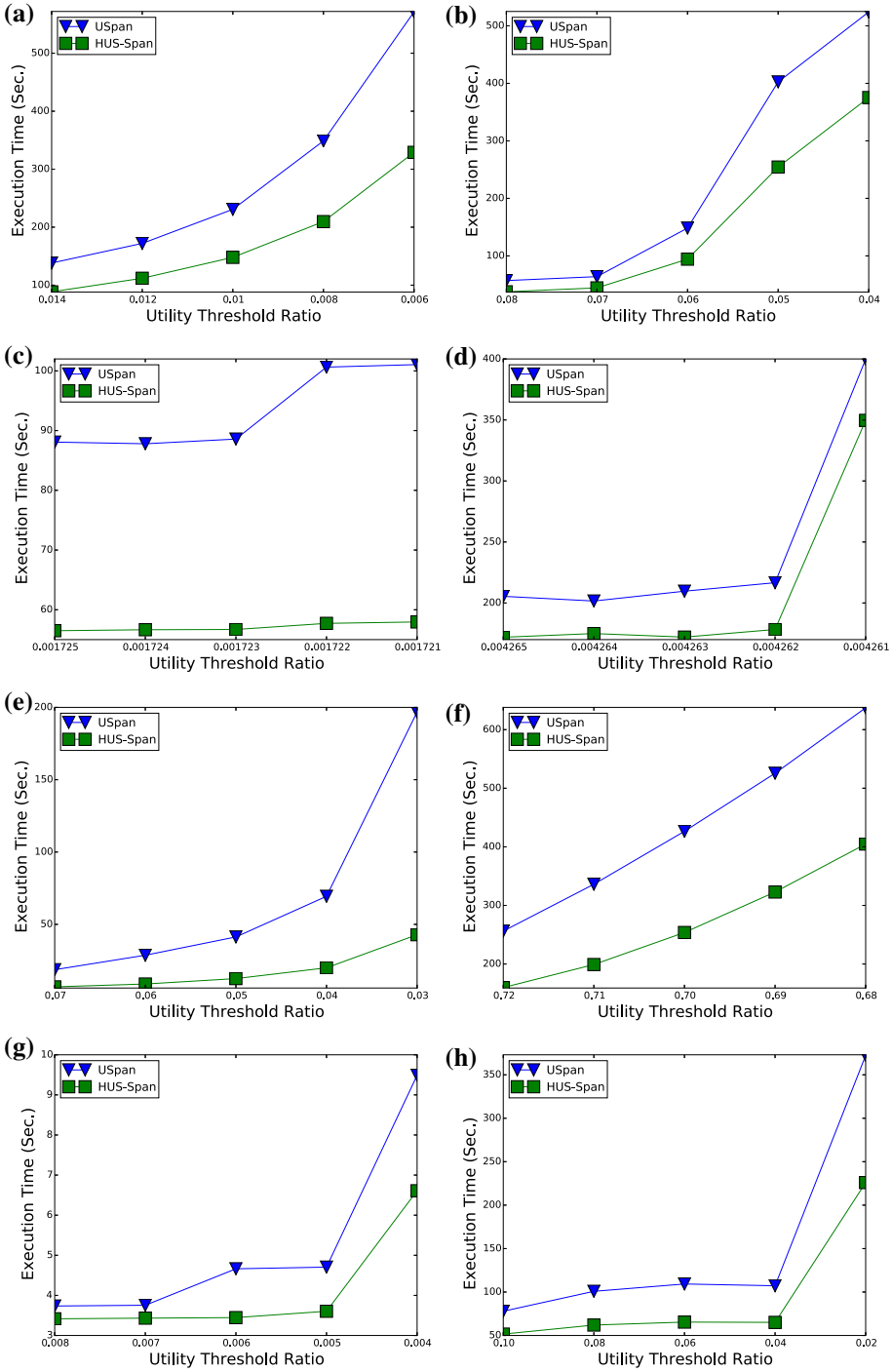


**Fig. 8** Effect of utility threshold. **a** Execution time on D10C8T4N0.1S4I2.5. **b** Execution time on D100C8T4N10S6I2.5. **c** Execution time on foodmart. **d** Execution time on Tafeng. **e** Execution time on Msnbc. **f** Execution time on gene. **g** Execution time on music. **h** Execution time on movies

Figure 8g, h shows the performance on real product reviews datasets, Music and Movies, respectively. We can observe from Fig. 8g and Table 5 that on dataset Music, the speedup of HUS-Span over USpan becomes more significant when utility threshold ratio gets smaller. The reason is that with the aid of PEU and RSU, HUS-Span can generate less candidates

**Table 5** Number of candidates

| $\gamma$ | 0.014 | 0.012 | 0.01 | 0.008 | 0.006 |
|---|---|---|---|---|---|
| D10C8T4N0.1S4I2.5 | | | | | |
| USpan | 1,268,636 | 1,931,159 | 3,184,118 | 5,895,354 | 13,146,809 |
| HUS-Span | 285,404 | 422,656 | 676,013 | 1,208,351 | 2,554,345 |
| $\gamma$ | 0.08 | 0.07 | 0.06 | 0.05 | 0.04 |
| D100C8T4N10S6I2.5 | | | | | |
| USpan | 17,955 | 135,915 | 1,524,253 | 7,942,976 | 13,555,491 |
| HUS-Span | 16,995 | 135,915 | 1,523,803 | 7,939,825 | 13,555,487 |
| $\gamma$ | 0.001725 | 0.001724 | 0.001723 | 0.001722 | 0.001721 |
| Foodmart | | | | | |
| USpan | 1,540,820 | 1,543,929 | 1,547,228 | 2,625,574 | 2,667,625 |
| HUS-Span | 568,910 | 570,554 | 572,150 | 613,906 | 646,331 |
| $\gamma$ | 0.004265 | 0.004264 | 0.004263 | 0.004262 | 0.004261 |
| Tafeng | | | | | |
| USpan | 53,774 | 53,850 | 128,481 | 5,455,894 | 147,189,639 |
| HUS-Span | 18,111 | 18,130 | 62,097 | 5,389,344 | 147,122,966 |
| $\gamma$ | 0.07 | 0.06 | 0.05 | 0.04 | 0.03 |
| Msnbc | | | | | |
| USpan | 558 | 902 | 1713 | 4217 | 18,338 |
| HUS-Span | 354 | 552 | 1002 | 2230 | 8181 |
| $\gamma$ | 0.72 | 0.71 | 0.7 | 0.69 | 0.68 |
| Gene | | | | | |
| USpan | 69907 | 92,801 | 121,220 | 152,510 | 188,722 |
| HUS-Span | 34,301 | 43,018 | 55,516 | 71,489 | 90,507 |
| $\gamma$ | 0.008 | 0.007 | 0.006 | 0.005 | 0.004 |
| Music | | | | | |
| USpan | 10,138 | 10,510 | 37,791 | 42,258 | 296,084 |
| HUS-Span | 9186 | 9527 | 10,757 | 14,835 | 134,142 |
| $\gamma$ | 0.1 | 0.08 | 0.06 | 0.04 | 0.02 |
| Movies | | | | | |
| USpan | 250,955 | 296,671 | 308,481 | 309,720 | 8,067,671 |
| HUS-Span | 250,955 | 296,671 | 308,481 | 309,720 | 4,092,608 |

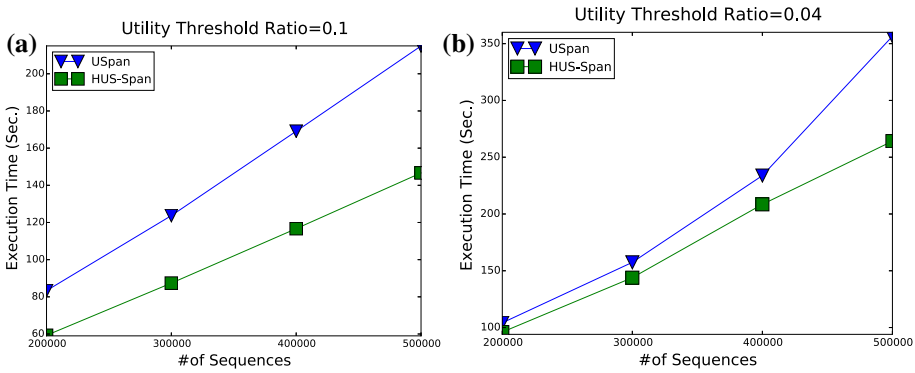


**Fig. 9** Effect of dataset size. **a** Execution time on DxC8T4N0.1S4I2.5. **b** Execution time on DxC8T4N10S6I2.5

than USpan does. On dataset Movies, although the numbers of candidates generated by HUS-Span and USpan are the same when utility threshold ratio is larger than or equal to 0.04 (as shown in Table 5), HUS-Span still outperforms USpan as shown in Fig. 8h. The reason is that HUS-Span can reduce more projected DB scans with the aid of PEU.

### *5.1.2 Effect of database size*

We also measure the performance of HUS-Span and USpan under different dataset sizes, where the size of the synthetic datasets ranges from 200000 to 500000 in increments of 100000. The experiments are performed on the synthetic dense dataset D*x*C8T4N0.1S4I2.5 and the sparse dataset D*x*C8T4N10S6I2.5, and the results are shown in Fig. 9a, b, respectively. We observe from Fig. 9a, b that the speedup of HUS-Span over USpan in the dense dataset D*x*C8T4N0.1S4I2.5 is more significant than that in the sparse dataset D*x*C8T4N10S6I2.5. Moreover, HUS-Span gets better than USpan when the number of sequences increases, showing that HUS-Span is more scalable than USpan, especially on dense datasets.

In summary, we can observe from the above experiments that HUS-Span outperforms USpan in all cases on the synthetic and real datasets. The reasons are twofold. First, our utility upper bounds, PEU and RSU, are tighter than those adopted by USpan, making HUS-Span able to prune more nodes and reduce more projected DB scans than USpan does. Second, with the aid of the proposed utility-chain structure, HUS-Span is able to rapidly calculate the PEU, RSU and utility of each sequence, thereby achieving shorter execution time.

## 5.2 Experimental results of top-*k* high utility sequential pattern mining

### *5.2.1 Effect of the maximal size of BFS queue*

To evaluate the effect of the maximal size of $BFSqueue$ on execution time and memory consumption, we conduct an experiment on dataset Foodmart by fixing $k$ to 100, where the maximal size of $BFSqueue$ ranges from $2^{16}$ to $2^{20}$ with a grow ratio of 2. Since only TKHUS-Span$_{\text{Hybrid}}$ is affected by the maximal size of $BFSqueue$, only the experimental result of TKHUS-Span$_{\text{Hybrid}}$ is shown in Fig. 10. We can observe from Fig. 10b that the memory consumption almost linearly increases with the increase in the maximal size of $BFSqueue$. Such result agrees with our intuition. In addition, as shown in Fig. 10a, the

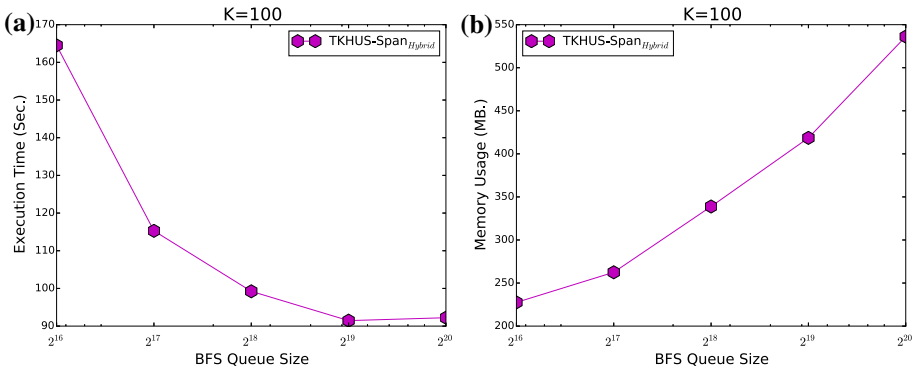


**Fig. 10** Effect of the maximal size of BFS queue. **a** Execution time. **b** Memory consumption

execution time decreases as the maximal size of $BFSqueue$ increases since larger available memory makes TKHUS-Span$_{\text{Hybrid}}$ able to find high utility sequential patterns more quickly. However, when the maximal size of $BFSqueue$ is large enough (i.e., available memory is large enough), increasing the maximal size of $BFSqueue$ does not result in significant reduction in execution time. In the following experiments, we set the maximal size of $BFSqueue$ to $2^{18}$ to strike a balance between mining efficiency and memory consumption for each dataset except dataset Gene. On dataset Gene, we set the maximal size of $BFSqueue$ to $2^{13}$ due to the characteristic that TKHUS-Span$_{\text{Hybrid}}$ runs out of memory space when the maximal size of $BFSqueue$ grows to $2^{14}$ for $k > 3$.

### *5.2.2 Effect of k*

In this section, we evaluate the effect of $k$ on the performance of these algorithms. To evaluate the effects of varied $k$, we increase $k$ from 1 to 10,000 with a grow ratio of 10 for each dataset. However, due to the characteristics of these datasets, the DFS-based algorithms spend considerable time (larger than 10000 seconds) on some datasets when $k \geq 100$, yet short on others. So, we use different scales for each dataset to compare the performance of these algorithms. The execution time and the numbers of candidates generated by these algorithms on each dataset are shown in Fig. 11 and Table 6, respectively.

Figure 11a, b shows the execution time on the synthetic datasets D10C8T4 N0.1S4I2.5 and D100C8T4N10S6I2.5, respectively. In Fig. 11a, on the dense dataset D10C8T4N0.1S4I2.5, we can see that TKHUS-Span$_{\text{BFS}}$ and TKHUS-Span$_{\text{Hybrid}}$ perform the best and the difference between TUS and TKHUS-Span$_{\text{BFS}}$ becomes significant when $k \geq 10000$. This is because the BFS-based algorithms traverse the lexicographic tree in a more global manner than DFS-based algorithms do, thus being able to find top-$k$ high utility sequential patterns more quickly. This is also validated by the numbers of generated candidates as given in Table 6. The number of candidates generated by TUS is about 180 times more than that generated by the BFS-based algorithms when $k = 10000$. On the sparse dataset D100C8T4N10S6I2.5, we can see that TKHUS-Span$_{\text{Hybrid}}$ has the best performance, followed by TKHUS-Span$_{\text{BFS}}$, TKHUS-Span$_{\text{GDFS}}$ and TUS. Similarly, as shown in Fig. 11b and Table 6, when $k$ gets larger, the improvements of the BFS-based algorithms over the DFS-based algorithms become more significant in terms of execution time and the numbers of generated candidates.

Figure 11c, d shows the execution time on the real grocery shopping datasets Foodmart and Tafeng, respectively. As shown in Fig. 11c, TKHUS-Span$_{\text{BFS}}$ has the best performance

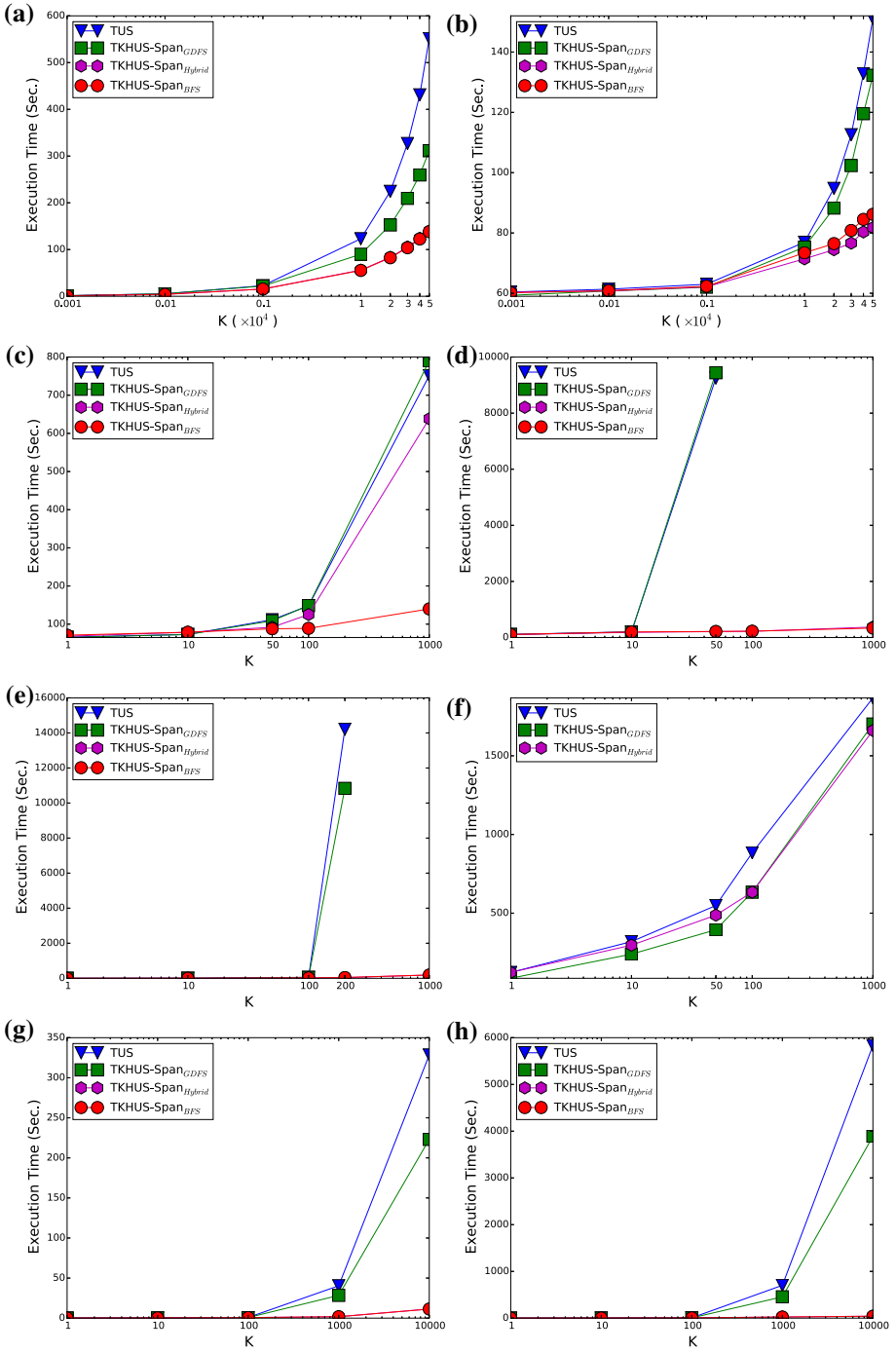


**Fig. 11** Effect of $k$. **a** Execution time on D10C8T4N0.1S4I2.5. **b** Execution time on D100C8T4N10S6I2.5. **c** Execution time on foodmart. **d** Execution time on Tafeng. **e** Execution time on Msnbc. **f** Execution time on gene. **g** Execution time on music. **h** Execution time on movies

**Table 6** Number of visited nodes

| K | 10000 | 20000 | 30000 | 40000 | 50000 |
|---|---|---|---|---|---|
| D10C8T4N0.1S4I2.5 | | | | | |
| TUS | 6,596,345 | 16,897,917 | 28,006,638 | 39,652,138 | 52,407,544 |
| TKHUS-Span$_{GDFS}$ | 1,575,403 | 5,291,370 | 9,231,872 | 12,797,686 | 16,844,738 |
| TKHUS-Span$_{Hybrid}$ | 36,732 | 71,236 | 103,408 | 134,534 | 165,467 |
| TKHUS-Span$_{BFS}$ | 36,732 | 71,236 | 103,408 | 134,534 | 165,467 |
| D100C8T4N10S6I2.5 | | | | | |
| TUS | 527,411 | 1,790,952 | 3,595,141 | 5,628,303 | 7,669,982 |
| TKHUS-Span$_{GDFS}$ | 245,647 | 989,746 | 2,096,549 | 3,442,181 | 4,771,458 |
| TKHUS-Span$_{Hybrid}$ | 7061 | 18,446 | 25,997 | 44,682 | 46,916 |
| TKHUS-Span$_{BFS}$ | 7061 | 18,446 | 25,997 | 44,682 | 46,916 |
| K | 1 | 10 | 50 | 100 | 1000 |
| Foodmart | | | | | |
| TUS | 60,082 | 112,242 | 382,266 | 739,450 | 7,561,228 |
| TKHUS-Span$_{GDFS}$ | 60,011 | 111,355 | 382,963 | 744,691 | 7,588,483 |
| TKHUS-Span$_{Hybrid}$ | 46,224 | 66,459 | 167,421 | 392,338 | 6,025,016 |
| TKHUS-Span$_{BFS}$ | 46,224 | 47,842 | 53,989 | 59,839 | 155,850 |
| Tafeng | | | | | |
| TUS | 505 | 8927 | 5,124,466 | – | – |
| TKHUS-Span$_{GDFS}$ | 504 | 8657 | 5,121,166 | – | – |
| TKHUS-Span$_{Hybrid}$ | 504 | 5655 | 6769 | 7925 | 572,250 |
| TKHUS-Span$_{BFS}$ | 504 | 5655 | 6769 | 7925 | 22,452 |
| Msnbc | | | | | |
| TUS | 7 | 32 | 2294 | 800,672 | – |
| TKHUS-Span$_{GDFS}$ | 5 | 29 | 2041 | 760,280 | – |
| TKHUS-Span$_{Hybrid}$ | 5 | 19 | 546 | 1442 | 38,707 |
| TKHUS-Span$_{BFS}$ | 5 | 19 | 546 | 1442 | 38,707 |
| Gene | | | | | |
| TUS | 11,632 | 37,812 | 62,327 | 97,111 | 321,860 |
| TKHUS-Span$_{GDFS}$ | 10,540 | 33,571 | 54,542 | 84,377 | 271,602 |
| TKHUS-Span$_{Hybrid}$ | 10,540 | 33,196 | 52,249 | 73,943 | 217,082 |
| TKHUS-Span$_{BFS}$ | 10,540 | – | – | – | – |
| Music | | | | | |
| TUS | 71 | 329 | 1388 | 1,258,852 | 10,631,332 |
| TKHUS-Span$_{GDFS}$ | 69 | 284 | 1042 | 798,800 | 6,710,901 |
| TKHUS-Span$_{Hybrid}$ | 69 | 254 | 861 | 2324 | 28,845 |
| TKHUS-Span$_{BFS}$ | 69 | 254 | 861 | 2324 | 28,845 |

**Table 6** continued

| K | 1 | 10 | 100 | 1000 | 10000 |
|---|---|---|---|---|---|
| Movies | | | | | |
| TUS | 988 | 4104 | 14,361 | 9,716,140 | 84,187,158 |
| TKHUS-Span$_{GDFS}$ | 834 | 3235 | 10,342 | 5,898,046 | 51,746,841 |
| TKHUS-Span$_{Hybrid}$ | 834 | 3214 | 9416 | 41,717 | 67,359 |
| TKHUS-Span$_{BFS}$ | 834 | 3214 | 9416 | 41,717 | 67,359 |

under varied $k$, followed by TKHUS-Span$_{Hybrid}$, TKHUS-Span$_{GDFS}$ and TUS. Similar to other experiments, the improvements of TKHUS-Span$_{BFS}$ over the DFS-based algorithms become more significant when $k$ increases. On the sparse dataset Tafeng, we can also see from Fig. 11d that the execution time of all the algorithms is close when $k$ is smaller than 50. When $k$ is 50, the BFS-based algorithms outperform the DFS-based algorithms about 45 times in terms of execution time. Moreover, the execution time of the BFS-based algorithms does not significantly increase when $k$ increases. The reason is that the BFS-based algorithms are able to find top-$k$ high utility sequential patterns in the early stage, and thus, usually outperform DFS-based algorithms.

Figure 11e shows the execution time of each algorithm on real dense dataset Msnbc, and we can observe that the execution time of each algorithm is close when $k$ is smaller than 100. Similar to the experiment on dataset Tafeng, the DFS-based algorithms is much slower than the BFS-based algorithms when $k$ is larger than 100. In our experiment, the DFS-based algorithms run even almost 356 times slower than the BFS-based algorithms when $k$ is 200. Figure 11f shows the execution time on real dataset Gene. We only show the results of TKHUS-Span$_{Hybrid}$, TKHUS-Span$_{GDFS}$ and TUS since TKHUS-Span$_{BFS}$ runs out of memory space when $k$ is larger than 3. In Fig. 11f, we can see that although TKHUS-Span$_{Hybrid}$ is of the best performance, the execution time of these three algorithms is close. The reason is that the number of distinct items in the dataset Gene is 8 and almost all sequences are formed by only 4 of these 8 items. In addition, each itemset in the dataset Gene consists of only one item. This phenomenon makes each node in the lexicographic tree is of close PEU, thereby reducing the pruning effect of PEU.

We can observe from Fig. 11g, h that all algorithms are of close performance on real datasets Music and movies when $k$ is smaller than or equal to 100. However, the DFS-based algorithms get much slower than the BFS-based algorithms when $k$ gets larger than 100. When $k$ is 10,000, the DFS-based algorithms run even about 29 and 142 times slower than the BFS-based algorithms do on datasets Music, and Movies, respectively. The reason is that when $k$ becomes larger, the BFS-based algorithms can visit much more promising nodes in the lexicographic tree in the earlier stage than the DFS-based algorithms, thus generating much fewer candidates than the DFS-based algorithms (as shown in Table 6).

### *5.2.3 Effect of database size*

We also measure the performance of these algorithms under different dataset sizes, where the size of the synthetic datasets ranges from 200,000 to 500,000 in increments of 100,000. The experiments are performed on the synthetic dense dataset D$x$C8T4N0.1S4 I2.5 under $k = 100$ and the sparse dataset D$x$C8T4N10S6I2.5 under $k = 50000$, and the experimental results are shown in Fig. 12a, b, respectively. In Fig. 12a, b, we can observe that when the

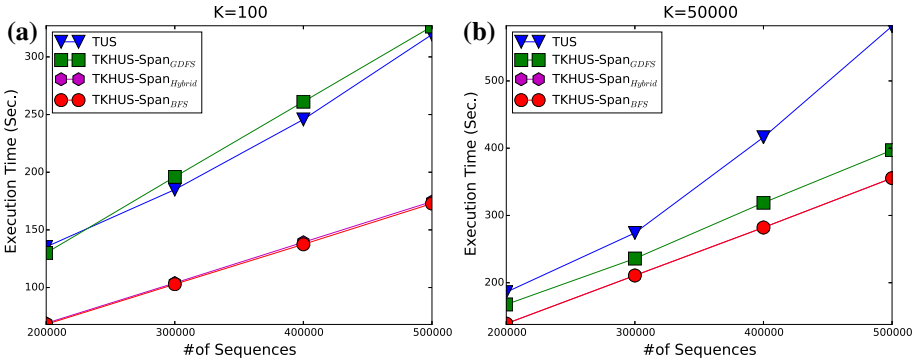


**Fig. 12** Effect of database size. **a** Execution time on DxC8T4N0.1S4I2.5. **b** Execution time on DxC8T4N10S6I2.5

number of sequences increases, the execution time of all algorithms almost increases linearly, and all the BFS-based algorithms outperform the DFS-based algorithms.

In summary, we can see from the above experiments that TKHUS-Span$_{\text{BFS}}$ and TKHUS-Span$_{\text{Hybrid}}$ outperform TKHUS-Span$_{\text{GDFS}}$ and TUS almost in all cases on all datasets when $k$ is large enough. The reasons are twofold. First, the BFS-based algorithms are able to find the top-$k$ high utility sequential patterns in the early stage by traversing the lexicographic tree using BFS. Second, PEU and RSU are able to prune more nodes than SPU (the pruning strategy used in TUS) does. Although outperforming the other algorithms on most datasets, TKHUS-Span$_{\text{BFS}}$ may run out of memory space. To avoid this problem, TKHUS-Span$_{\text{Hybrid}}$ can be applied to strike a balance between execution time and memory usage.

## 6 Conclusions

In this paper, we proposed a novel algorithm HUS-Span to efficiently mine high utility sequential patterns. Specifically, we developed two tighter utility upper bounds, PEU and RSU, as well as two companion pruning strategies to prune the search space. We also developed the utility-chain so that the utility, PEU and RSU of each sequence can be efficiently calculated. Extensive experimental results on both the synthetic and real datasets showed that HUS-Span outperforms the state-of-the-art algorithm USpan in terms of execution time.

In addition, with the aid of PEU, RSU and utility-chain, we also proposed three algorithms, TKHUS-Span$_{\text{GDFS}}$, TKHUS-Span$_{\text{BFS}}$ and TKHUS-Span$_{\text{Hybrid}}$ to mine top-$k$ high utility sequential patterns by traversing the lexicographic tree by guided DFS, BFS and hybrid search of BFS and GDFS. It is obvious that TKHUS-Span$_{\text{BFS}}$ has better possibility to find out the top-$k$ high utility sequential patterns more quickly. However, TKHUS-Span$_{\text{BFS}}$ suffers from the problem of running out of memory space in some cases. In view of this, TKHUS-Span$_{\text{Hybrid}}$, a hybrid search algorithm, which combines the relative advantages of GDFS and BFS, is a good trade-off in all cases. Extensive experimental results on both the synthetic and real datasets showed that TKHUS-Span$_{\text{BFS}}$ generally outperforms all algorithms when $k$ is large except the dataset where TKHUS-Span$_{\text{BFS}}$ runs out of memory space. Therefore, in the situation with limited memory space, TKHUS-Span$_{\text{Hybrid}}$ can be applied to achieve a better performance by avoiding memory shortage.

**Acknowledgments** The authors were supported by the Ministry of Science and Technology, Taiwan, under Project No. MOST 104-2221-E-032-037-MY2, MOST 103-2221-E-009-126-MY2, MOST 104-2918-I-009-003, MOST 104-2218-E-009-009 and MOST 104-2218-E-009-029.

## References

1. Amazon reviews (2013). http://snap.stanford.edu/data/web-Amazon.html
2. Ahmed CF, Tanbeer SK, Byeong-Soo J (2011) A framework for mining high utility web access sequences. IETE Tech Rev 28(1):3–16
3. Ahmed CF, Tanbeer SK, Jeong B-S (2010) A novel approach for mining high-utility sequential patterns in sequence databases. ETRI J 32(5):676–686
4. Ahmed CF, Tanbeer SK, Jeong B-S, Lee Y-K (2009) Efficient tree structures for high utility pattern mining in incremental databases. IEEE Trans Knowl Data Eng 21(12):1708–1721
5. Ayres J, Gehrke J, Yiu T, Flannick J (2002) Sequential pattern mining using a bitmap representation. In: Proceedings of the 8th ACM international conference on knowledge discovery and data mining, pp 429–435
6. Bache K, Lichman M (2013) UCI machine learning repository. http://archive.ics.uci.edu/ml
7. Garofalakis MN, Rastogi R, Shim K (1999) Spirit: sequential pattern mining with regular expression constraints. In: Proceedings of the 25th international conference on very large data bases, pp 223–234
8. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proceedings of the 2000 ACM SIGMOD international conference on management of data, pp 1–12
9. Kim C, Lim J-H, Ng RT, Shim K (2007) Squire: sequential pattern mining with quantities. J Syst Softw 80(10):1726–1745
10. Lan G-C, Hong T-P, Tseng VS, Wang S-L (2014) Applying the maximum utility measure in high utility sequential pattern mining. Expert Syst Appl 41(11):5071–5081
11. Li Y-C, Yeh J-S, Chang C-C (2008) Isolated items discarding strategy for discovering high utility itemsets. Data Knowl Eng 64(1):198–217
12. Liu J, Wang K, Fung B (2012) Direct discovery of high utility itemsets without candidate generation. In: the 12rd IEEE international conference on data mining, pp 984–989
13. Liu M, Qu J (2012) Mining high utility itemsets without candidate generation. In: Proceedings of the 21st ACM international conference on Information and knowledge management, pp 55–64
14. Liu Y, Liao W, Choudhary A (2005) A fast high utility itemsets mining algorithm. In: Proceedings of the 1st ACM international workshop on utility-based data mining, pp 90–99
15. Microsoft sql server 2008 analysis services unleashed (2008). http://www.informit.com/store/microsoft-sql-server-2008-analysis-services-unleashed-9780672330018
16. Pei J, Han J, Mortazavi-Asl B, Wang J, Pinto H, Chen Q, Dayal U, Hsu M-C (2004) Mining sequential patterns by pattern-growth: the prefixspan approach. IEEE Trans Knowl Data Eng 16(11):1424–1440
17. Russell SJ, Norvig P (2003) Artificial intelligence: a modern approach, 2nd edn. Pearson Education, London
18. Shie B-E, Hsiao H-F, Tseng VS, Yu PS (2011) Mining high utility mobile sequential patterns in mobile commerce environments. In: Proceedings of the 16th international conference on database systems for advanced applications, pp 224–238
19. Srikant R, Agrawal R (1996) Mining sequential patterns: Generalizations and performance improvements. In: Proceedings of the 5th international conference on extending database technology: advances in database technology, pp 3–17
20. Ta-feng datasets (2001). http://aiia.iis.sinica.edu.tw/index.php?option=com_frontpage&Itemid=1
21. Tseng VS, Wu C-W, Shie B-E, Yu PS (2010) Up-growth: an efficient algorithm for high utility itemset mining. In: Proceedings of the 16th ACM SIGKDD international conference on knowledge discovery and data mining, pp 253–262
22. Wang J, Han J (2004) Bide: Efficient mining of frequent closed sequences. In: Proceedings of the 20th IEEE international conference on data engineering, pp 79–90
23. Wu CW, Shie B-E, Tseng VS, Yu PS (2012) Mining top-k high utility itemsets. In: Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining, pp 78–86
24. Yan X, Han J, Afshar R (2003) Clospan: Mining closed sequential patterns in large datasets. In: Proceedings of the 3rd SIAM international conference on data mining, pp 166–177
25. Yao H, Hamilton HJ, Butz CJ (2004) A foundational approach to mining itemset utilities from databases. In: Proceedings of the 7th SIAM international conference on data mining, pp 482–486

26. Yin J, Zheng Z, Cao L (2012) Uspan: an efficient algorithm for mining high utility sequential patterns. In: Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining, pp 660–668
27. Yin J, Zheng Z, Cao L, Song Y, Wei W (2013) Efficiently mining top-k high utility sequential patterns. In: the 13rd IEEE international conference on data mining, pp 1259–1264
28. Zhang W, Korf RE (1993) Depth-first vs. best-first search: New results. In: the 11th AAAI national conference on artificial intelligence, pp 769–775

**Jun-Zhe Wang** received the B.S. and M.S. degrees in Information Management from National United University and National Chi Nan University, Taiwan, in 2007 and 2009, respectively. He is currently pursuing the Ph.D. degree in Computer Science, National Chiao Tung University, Taiwan. His research interests include data mining, mobile data management and spatial query processing.

**Jiun-Long Huang** received his B.S. and M.S. degrees in Computer Science and Information Engineering Department in National Chiao Tung University in 1997 and 1999, respectively, and his Ph.D. degree in Electrical Engineering Department in National Taiwan University in 2003. He joined National Chiao Tung University in 2005 and currently is an associate professor in Computer Science Department in National Chiao Tung University. His research interests include mobile computing, wireless networks, data mining and cloud computing.

**Yi-Cheng Chen** received the B.S. degree in Computer Science and Engineering from Yuan Ze University, Taiwan, in 2000, and the M.S. degree in Computer Science and Engineering from National Taiwan University of Science and Technology, Taiwan, in 2002, and the Ph.D. degree in Computer Science, National Chiao Tung University, Taiwan, in 2012. He has been an assistant professor in the Department of Computer Science and Information Engineering at, Tamkang University, Taiwan, since 2013. His research interests include data mining, cloud computing, social network analysis, bioinformatics and multimedia.