Interactive sequence discovery by incremental mining

Ming-Yen Lin, Suh-Yin Lee *

Department of Computer Science and Information Engineering, National Chiao Tung University, Taiwan 30050, Taiwan, ROC

Received 30 December 2001; received in revised form 10 March 2003; accepted 4 September 2003

Abstract

Sequential pattern mining has become a challenging task in data mining due to its complexity. Essentially, the mining algorithms discover all the frequent patterns meeting the user specified minimum support threshold. However, it is very unlikely that the user could obtain the satisfactory patterns in just one query. Usually the user must try various support thresholds to mine the database for the final desirable set of patterns. Consequently, the time-consuming mining process has to be repeated several times. However, current approaches are inadequate for such interactive mining due to the long processing time required for each query. In order to reduce the response time for each query during the interactive process, we propose a knowledge base assisted mining algorithm for interactive sequence discovery. The proposed approach utilizes the knowledge acquired from each mining process, accumulates the counting information to facilitate efficient counting of patterns, and speeds up the whole interactive mining process. Furthermore, the knowledge base makes possible the direct generation of new candidate sets and the concurrent support counting of variable sized candidates. Even for some queries, due to the pattern information already kept in the knowledge base, database access is not required at all. The conducted experiments show that our approach outperforms GSP, a state-of-the-art sequential pattern mining algorithm, by several order of magnitudes for interactive sequence discovery.

© 2003 Elsevier Inc. All rights reserved.

Keywords: Data mining; Sequential pattern; Interactive discovery; Knowledge base; Incremental mining

*Corresponding author.
E-mail address: sylee@csie.nctu.edu.tw (S.-Y. Lee).

0020-0255/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
1. Introduction

An important issue in data mining is the discovery of sequential patterns, which finds out temporal associations among items in the sequence database [2,4,6,8,9,15,17,18]. A classic application of the problem is the market basket analysis whose database contains purchase records, where each record is an ordered sequence of itemsets (sets of items) bought by a customer. The mining is to discover the itemsets in future purchase after certain itemsets were bought. For example, a discovery might find out a sequential pattern \((1, 3, 4) \Rightarrow (2, 5)\) [support = 30\%], which means that 30\% of customers who purchase items 1, 3 and 4 at the same time would buy items 2 and 5 at some later time. The technique can be applied to various domains such as discovering the relationships between the symptoms and certain diseases in medical applications.

In order to find the interesting patterns, a user specifies a minimum support threshold (abbreviated \textit{minsup}) for the mining. The result of the mining lists all patterns, named sequential patterns or frequent sequences, whose supports are greater than or equal to the \textit{minsup}. The support of a pattern is the percentage of sequences (in the database) containing the pattern. In general, we would generate potential sequential patterns (called candidates), count the occurrence of each candidate, and then determine the sequential patterns among these candidates.

The mining process is very difficult and time-consuming due to several factors. First, the formation of a pattern is not limited to single items but itemsets. Second, neither the number of itemsets in a pattern nor the number of items in an itemset is known a priori. Third, patterns could be formed by any permutation, of any combination of possible items in the database. Most approaches focused on minimizing the search space of candidates [2,16], or on minimizing the required disk I/O due to the multiple database scanning [15,18]. Each time a user specifies a \textit{minsup}, all these approaches discover the resultant patterns by executing their mining algorithms with respect to this \textit{minsup}.

However, a user may specify a \textit{minsup} value that results in too many or too few patterns. When the specified \textit{minsup} is too large, either no patterns or only few patterns might satisfy the threshold. On the contrary, the user might have difficulty in distinguishing the interesting patterns from a large number of patterns due to a very small \textit{minsup}. Usually, the user must try various \textit{minsup}s until the result is satisfactory. Nevertheless, most approaches for mining sequential patterns are not designed to deal with repeated mining under such circumstance. For such interactive sequence discovery, these approaches consider no prior results so that the mining process must start over again for every newly specified \textit{minsup}. However, keeping knowledge obtained from the time-consuming process is beneficial to further queries [7]. For example, the result of
mining with \( \text{minsup} = 0.1 \) could be used to extract the sequential patterns for \( \text{minsup} = 0.3 \) without re-examining the sequence database.

Therefore, we propose a novel approach, named knowledge base assisted incremental sequential pattern mining (KISP), to improve the efficiency of sequential pattern discovery with changing supports. Instead of re-mining from scratch for each discovery, KISP utilizes the knowledge obtained from prior mineings, and generates a knowledge base for further queries about sequential patterns of various minsup. When the sequential patterns cannot be directly derived from the knowledge base, KISP incorporates the knowledge base into a fast sequence discovery. The candidates existing in the knowledge base are spared in the support counting process. In addition, the knowledge base could be used to support OLAP since the knowledge, sufficient for users' interests, of current database is accumulated by KISP. The conducted experiments on synthetic data also show that the proposed algorithm effectively improves the performance of interactive sequence discovery.

The rest of the paper is organized as follows. We formulate the problem of interactive sequential pattern mining in Section 2 and review some related algorithms in Section 3. Section 4 presents the proposed approach for the interactive discovery problem. The experimental evaluation is described in Section 5. Section 6 concludes our study.

2. The problem of interactive sequence discovery

Let \( \Psi = \{x_1, x_2, \ldots, x_k\} \) be a set of literals, called items. A set of items is referred to as an itemset. An itemset \( I \) with \( m \) items is denoted by \( I = (\beta_1, \beta_2, \ldots, \beta_m) \), such that \( I \subseteq \Psi \). A sequence \( x \), denoted by \( \langle a_1 a_2 \ldots a_n \rangle \), is an ordered set of \( n \) elements where each element \( a_j \) is an itemset. The size of the sequence \( x \), denoted by \( |x| \), is the total number of items in all the elements in \( x \). Sequence \( x \) is a \( k \)-sequence if \( |x| = k \). For example, \( \langle(1)(3)(5)\rangle \), \( \langle(2)(3,4)\rangle \), and \( \langle(1)(2)(1)\rangle \) are all 3-sequences. A sequence \( \omega = \langle a_1 a_2 \ldots a_n \rangle \) is a subsequence of another sequence \( \sigma = \langle b_1 b_2 \ldots b_w \rangle \) if there exist \( 1 \leq i_1 < i_2 < \cdots < i_n \leq w \) such that \( a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, \ldots, a_n \subseteq b_{i_n} \). Sequence \( \sigma \) contains sequence \( \omega \) if \( \omega \) is a subsequence of \( \sigma \). For instance, \( \langle(2)(5)\rangle \) is a subsequence of \( \langle(4)(2)(1)(3,5)\rangle \) since \( (2) \subseteq (2) \) and \( (5) \subseteq (3,5) \).

Each customer record in the database \( DB \) is referred to as a data sequence, which is a sequence of purchased itemsets ordered by transaction time. The support of sequence \( x \), denoted by \( x\cdotsup \), is the number of data sequences containing \( x \) divided by the total number of data sequences in database \( DB \). The \( \text{minsup} \) is the user specified minimum support threshold. A sequence \( x \) is a frequent sequence if \( x\cdotsup \geq \text{minsup} \). The sequence \( x \) is also called a sequential pattern. Given the \( \text{minsup} \) and the database \( DB \), the problem of sequential
pattern mining is to discover the set of all sequential patterns, denoted by $S[\text{minsup}]$.

The interactive sequence discovery process is described as follows. Given the database $DB$, the user queries with several $\text{minsup}$ values interactively, and finds out the desired set of sequential patterns with respect to the final $\text{minsup}$. The objective of interactive discovery is to respond to each query quickly and to reduce the overall mining time for the whole process accordingly.

3. Related work

3.1. Algorithms for sequential pattern mining

The problem of sequential pattern mining is first described and solved in [2] with the AprioriAll algorithm. In subsequent work, the same authors proposed the GSP algorithm [16] that outperforms AprioriAll. The GSP algorithm makes multiple passes over the database and finds frequent $k$-sequences at $k$th database scanning. Initially, each item is a candidate $1$-sequence for the first pass. Frequent $1$-sequences are determined after checking all the data sequences in the database. In succeeding passes, frequent $(k - 1)$-sequences are self-joined to generate candidate $k$-sequences, and then any candidate $k$-sequence having a non-frequent sub-sequence is deleted. Again, the supports of candidate $k$-sequences are counted by examining all data sequences, and then those candidates having minimum supports become frequent sequences. This process terminates when there is no candidate sequence any more. Owing to the generate-and-test nature, the number of candidates often dominates the overall mining time. However, the total number of candidates increases exponentially as the $\text{minsup}$ decreases, even with effective pruning techniques. The Prefix Sequential Pattern (PSP) algorithm [9] is similar to GSP, except that the placement of candidates, in a hash-tree [2,3] in GSP, is improved by a prefix tree arrangement.

The Sequential Pattern Discovery using Equivalence classes (SPADE) algorithm finds out sequential patterns using vertical database layout and join-operations [18]. Vertical database layout transforms data sequences into item-oriented lists. For example, the transformation of a sequence $(1,3)(5)$ with sequence id $C310$ would generate an entry $(C310, 1)$ in the list of item ‘1’, an entry $(C310, 1)$ in the list of item ‘3’, and an entry $(C310, 2)$ in the list of item ‘5’. The lists are joined to form a sequence lattice, in which SPADE searches and discovers the patterns [18].

Recently, the Frequent pattern-projected Sequential Pattern Mining (FreeSpan) algorithm was proposed to mine sequential patterns by a database projection technique [4]. FreeSpan first finds the frequent items after scanning the database once. The sequence database is then projected, according to the
frequent items, into several smaller databases. Finally, all sequential patterns are found by recursively growing subsequence fragments in each projected database. Based on the similar projection technique, Prefix-projected Sequential pattern mining (PrefixSpan) algorithm [14] efficiently mines the complete set of patterns employing a divide-and-conquer strategy with the PatternGrowth methodology.

However, all these algorithms re-execute the mining procedure every time a new minsup is specified during the interactive process. Therefore, the response time would be longer for subsequent queries with smaller minsup values with all these algorithms.

3.2. Algorithms for interactive pattern discovery

The problem of interactive association discovery, also called online association generation, was addressed in [1]. The method in [1] preprocesses the data in the transactional database, and stores frequent itemsets in an adjacency lattice. Each vertex in the adjacency lattice is labeled with the support of the corresponding itemset. A directed edge in the lattice links from a ‘parent’ itemset to one of its ‘child’ itemsets. An itemset Y is a ‘child’ of itemset X if Y can be obtained from X by dropping a single item from X. Online repeated queries about association rules are answered by graph theoretic searching on the lattice.

Similarly, a knowledge cache storing the discovered frequent itemsets and the non-frequent itemsets is used for interactive discovery of association rules [11]. It is indicated that their benefit replacement algorithm using B+-tree to store cache buckets is the best caching algorithm [11].

Although on-line association discovery is close to our problem, the aim of these approaches [1,5,11,12] is to interactively find frequent itemsets rather than frequent sequences. One related work of interactive sequence mining is described below.

The SPADE algorithm [18] was extended into the ISM (Incremental Sequence Mining) algorithm for incremental sequence mining and interactive sequence mining [13]. All queries are performed on a pre-processed in-memory data structure, the Increment Sequence Lattice (ISL). Therefore, a ‘small enough’ minsup must be selected in advance to mine all patterns by executing SPADE and save the results in the ISL. Nevertheless, if a query involves a support smaller than the pre-selected minsup, another (more) lengthy mining process must be performed to generate another new ISL sufficient for the new query. Moreover, the ISM might encounter memory problem if the number of the potentially frequent patterns is too large [13].

Without any assumption on the possible values of minsup, our algorithm aims to reduce the response time for interactive queries.
The proposed algorithm for interactive discovery of sequential patterns

The proposed KISP algorithm is described in Section 4.1. The algorithm speeds up the mining process by eliminating the counting efforts required for those candidates already existing in the knowledge base. Two optimizations are proposed for further improvements. In Section 4.2, the generation of the remaining 'new' candidates is optimized by direct computation. Enabled by candidate reduction and assisted by the information in the knowledge base, the optimization by current support counting is depicted in Section 4.3. The manipulation of the knowledge base is presented in Section 4.4. Section 4.5 discusses the mining efficiency and space utilization with a large knowledge base.

4.1. The knowledge base assisted incremental sequential pattern (KISP) mining algorithm

Fig. 1 outlines the proposed Basic KISP algorithm for interactive discovery of sequential patterns. We adopt the GSP algorithm as the basis for constructing the knowledge base assisted mining algorithm. KISP uses similar

Algorithm KISP \( (DB, KB, \text{minsup}) \)

Input: \( DB \) = the database of data sequences; \( \text{minsup} \) = user specified minimum support

\( KB \) = knowledge base having the supports of all the candidates in prior minings

Output: \( S[\text{minsup}] \) = sequential patterns with respect to \( \text{minsup} \); \( KB = (\text{new}) \) knowledge base

// Let \( x.sup \) be the support of a candidate \( x \), \( X_k[\text{minsup}] \) be the set of candidate \( k \)-sequence in \( DB \) with respect to \( \text{minsup} \), and \( KB.sup \) be the smallest \( \text{minsup} \) used in the construction of \( KB \)

1) if \( KB = \emptyset \) then \( KB = \{ x \text{ and } x.sup, \forall x \in X_1 \} \);
2) \( S[\text{minsup}] = \{ x \text{ where } KB \wedge x.sup \geq \text{minsup} \} \); // obtain valid sequential patterns from knowledge base
3) if \( \text{minsup} < KB.sup \) then // mine new patterns and accumulate new knowledge
    4) \( k = 2 \);
    5) generate \( X_k[\text{minsup}] \) from the frequent \((k-1)\)-sequences in \( S[\text{minsup}] \);
    6) \( X_k' = X_k[\text{minsup}] - \{ x \text{ where } x \in KB \} \); // eliminate those candidate \( k \)-sequences in \( KB \)
    7) while \( X_k' \neq \emptyset \) do // there exists candidate \( k \)-sequences, obtains their supports
        8) for each data sequence \( ds \) in database \( DB \) do
            9) for each candidate \( x \) in \( X_k' \) do
                10) increase the support of \( x \text{ if } x \text{ is contained in } ds \);
            endfor
        endfor
        11) \( KB = KB \cup \{ x \text{ and } x.sup, \forall x \in X_k' \} \); // collect new candidates and their supports
    12) \( S[\text{minsup}] = S[\text{minsup}] \cup \{ x \text{ with } x.sup \geq \text{minsup} \land x \in X_k' \} \); // collect new patterns from \( X_k' \)
    13) \( k = k+1 \);
    14) generate \( X_k[\text{minsup}] \) from the frequent \((k-1)\)-sequences in \( S[\text{minsup}] \);
    15) \( X_k' = X_k[\text{minsup}] - \{ x \text{ where } x \in KB \} \); // the reduced set eliminates candidate \( k \)-sequences in \( KB \)
    endwhile
16) \( KB.sup = \text{minsup} \); // update the smallest \( \text{minsup} \) of \( KB \)
17) \( KB = KB \cup \{ x \text{ and } x.sup, \forall x \in X_k' \} \); // collect new candidates and their supports
18) \( S[\text{minsup}] = S[\text{minsup}] \cup \{ x \text{ with } x.sup \geq \text{minsup} \land x \in X_k' \} \); // collect new patterns from \( X_k' \)
19) \( k = k+1 \);
20) endif

Fig. 1. Proposed Algorithm Basic KISP.
procedures of candidate generation and support counting as used in GSP. Nevertheless, KISP speeds up support counting by reducing considerable amounts of candidates and makes a significant performance improvement for interactive discovery.

During the interactive process, the knowledge base (denoted by KB) is empty only in the very first mining. Once KISP is executed, the information about the supports of counted candidates would be inserted into KB. The KB.sup is the minsup used when KB is constructed or expanded.

KISP works similar to GSP for the very first mining. Initially every item in the database is a candidate 1-sequence. The fundamental KB is built, only once, by a simple scan over the database to count the supports of candidate 1-sequences (line 1). After that, the supports of all candidate 1-sequences are included in KB and S[minsup] contains the frequent 1-sequences (line 2). At the end of this mining, KB would collect the supports of all the candidates in each pass (line 13), and KB.sup is the designated minsup (line 19).

Note that in KB we also keep the supports of all candidates regardless of their values for two reasons. First, several currently non-frequent candidates might turn out to be frequent when a smaller minsup is specified in subsequent queries. We can immediately obtain these patterns from KB without any database access. Second, to find out the true patterns, the mining process generally counts a large number of candidates although they are eventually rejected. We can get rid of the ‘useless counting’ for the ‘commonly non-frequent’ candidates if their supports were kept. For example, those candidates ever counted with the support value of zero would not be inserted into the candidate hash-tree afterward. Consequently, a faster counting is enabled due to the smaller hash-tree of the reduced set of candidates.

For subsequent queries, KB is not empty and contains the supports of all the generated candidates while mining with KB.sup as the support threshold. If the user-specified minsup is greater than KB.sup, we simply search in KB for patterns whose supports satisfy the new minsup, and return all patterns in S[minsup] (line 2). In this case, the employment of KB eliminates the need of re-mining completely in comparison with GSP. Tremendous gains in performance can be resulted from direct retrieval of valid patterns without re-counting the huge database. In fact, KISP would output all the valid patterns in constant time independent of the database size when KB.sup is less than the user specified minsup. On the contrary, other re-mining based algorithms such as GSP need to re-scan the database.

In case the minsup is less than KB.sup, we have to mine the database for new patterns that are not in KB. The fundamental difference between KISP and GSP is that KISP spares the counting of the candidates already existing in KB (line 6 and line 17). Take the number of candidates in pass 2 for example. Assume that in query Qi, there are 100 frequent 1-sequences so that

\[(100 \times 100) + (100 \times 99) / 2 = 14950\]

candidate 2-sequences are generated and
counted in pass 2. Assume that the number of frequent \(1\)-sequences is 110 for query \(Q_{i+1}\). In pass 2 of \(Q_{i+1}\), GSP must count in total \((110 \times 110) + (110 \times 109)/2 = 18,095\) candidates, while KISP counts only \((18,095 - 14,950) = 3,145\) candidates. In each pass of a query, we first generate the candidates and then remove those existing in \(KB\). Next, we expand \(KB\) with the support of every new candidate for reuse in future mining processes (line 13). The sequential patterns are collected (line 14). Finally, \(KB.sup\) is replaced by the new \(minsup\) since the counting base is changed (line 19). The ‘new pattern’ mining part (line 3 through line 20), which is also the part of new information acquisition step, of the procedure is activated again only when \(minsup < KB.sup\) occurs in subsequent queries.

In fact, the optimized KISP directly generates the new candidates requiring counting with the assistance of \(KB\), as presented in Section 4.2. In the following context, KISP stands for the optimized KISP.

4.2. New-candidate generation by direct computation

The first optimization in KISP is the direct generation of new candidates. As described in Section 4.1, KISP removes the candidates existing in \(KB\) before counting. The remaining candidates are referred to as new-candidates. Instead of generating all candidates and then removing the counted ones, we use Theorem 1 to generate the new-candidates in pass \(k\) (denoted by \(X'_k\)) directly. In Theorem 1, \(S_k[\text{minsup}]\) denotes the set of frequent \(k\)-sequences, \(X_k[\text{minsup}]\) denotes the set of candidate \(k\)-sequences with respect to \(minsup\), and “\(\otimes\)” represents the join operation. We use \(N_k[\text{minsup}]\) to designate the new frequent \(k\)-sequences (due to \(minsup\)) by contrast to the frequent \(k\)-sequences in \(KB\). Hence, \(S_k[\text{minsup}] = S_k[KB.sup] \cup N_k[\text{minsup}]\).

**Theorem 1.** \(X'_k = (S_{k-1}[KB.sup] \otimes N_{k-1}[\text{minsup}]) \cup (N_{k-1}[\text{minsup}] \otimes S_{k-1}[KB.sup])\). That is, \(X'_k\) is the union of the two sets; one obtained from joining the frequent \((k-1)\)-sequences in \(KB\) with the new frequent \((k-1)\)-sequences, the other obtained from self-joining the new frequent \((k-1)\)-sequences.

**Proof.** By definition, \(X_k[\text{minsup}] = S_{k-1}[\text{minsup}] \otimes S_{k-1}[\text{minsup}]\).

1. \(X_k[\text{minsup}] = (S_{k-1}[KB.sup] \cup N_{k-1}[\text{minsup}]) \otimes (S_{k-1}[KB.sup] \cup N_{k-1}[\text{minsup}])\).
2. \(X_k[\text{minsup}] = (S_{k-1}[KB.sup] \otimes S_{k-1}[KB.sup]) \cup (S_{k-1}[KB.sup] \otimes N_{k-1}[\text{minsup}]) \cup (N_{k-1}[\text{minsup}] \otimes S_{k-1}[KB.sup]) \cup (N_{k-1}[\text{minsup}] \otimes N_{k-1}[\text{minsup}])\).
3. \(X'_k = X_k[KB.sup] \cup (S_{k-1}[KB.sup] \otimes N_{k-1}[\text{minsup}]) \cup (N_{k-1}[\text{minsup}] \otimes N_{k-1}[\text{minsup}])\) due to \(X_k[KB.sup] = S_{k-1}[KB.sup] \otimes S_{k-1}[KB.sup]\) and that \(N_{k-1}[\text{minsup}] \otimes S_{k-1}[KB.sup]\) is the same set as \(S_{k-1}[KB.sup] \otimes N_{k-1}[\text{minsup}]\).
4. Since \(X'_k = X_k[\text{minsup}] - X_k[KB.sup], X'_k = (S_{k-1}[KB.sup] \otimes N_{k-1}[\text{minsup}]) \cup (N_{k-1}[\text{minsup}] \otimes N_{k-1}[\text{minsup}])\).
4.3. Concurrent support counting

The second optimization in KISP is the technique of concurrent support counting, which achieves database-pass reduction while preserving the completeness of pattern discovery. In KISP, the reduced candidate set $X'_k$ is more likely to occupy just a small part of the memory at pass $k$. We maximize memory utilization to reduce the number of database passes by inserting as many candidates as possible into the same hash-tree. We continuously generate the candidates of longer size until the memory space is nearly full. With the information about $S_{k-1}[KB.sup]$ and the $N_{k-1}[minsup]$, KISP can estimate the number of new-candidates, which indicates the space required. Therefore, we can place variable-sized candidates in the same hash-tree and concurrently count the supports against the data sequences in the same database pass. This technique reduces the total number of database scanning. The estimation procedure is described in the following.

Considering the number of candidates generated in each pass, the number of candidates in $X'_2$ is greater than that in other $X'_k$ because none in the candidate superset of size two can be pruned. Every frequent 1-sequence must join with other frequent 1-sequence since the subsequence of any frequent 1-sequence is an empty sequence. For candidates of $X'_k$ where $k > 2$, some frequent $(k - 1)$-sequences are not joined if their subsequences do not match. Assume the number of patterns in $S_1[KB.sup]$ is $p$ and the number of patterns in $N_1[minsup]$ is $q$. The number of new-candidates in pass 2 is $\frac{3(p + q)^2 - (p + q) - (3p^2 - p) - (3q^2 - q)}{2} = 3pq + (3q^2 - q)/2$. This formula can be applied to roughly estimate the maximum number of candidates in other passes. Whenever there is room for the next set of candidates (of longer size), KISP continuously generates and inserts the candidates into the same hash-tree. Thus, KISP can generate as many candidates as possible in the same pass.

Note that a similar technique named pass bundling is described for association mining in [10]. However, pass bundling statically sets a limit to determine whether the generation should be continued or not, while KISP dynamically estimates and computes the available memory for maximum utilization. The next section will describe the structure and the manipulation of the knowledge base, which is the key to facilitate the above stated improvements.

4.4. Manipulation of the knowledge base

We store the knowledge base in disk so that KISP is independent of the main memory size. Fig. 2 shows the structure of the knowledge base (KB). KB provides fast access to the pattern information, carries quick estimation of required candidate storage, and expands incrementally.

A knowledge base is composed of a minimal KB.sup, and one or more KB heads. The minimal KB.sup is the smallest KB.sup among all the KB.sup in the
We create a KB head to store the newly acquired information only when the user-specified \textit{minsup} is less than the minimal KB sup. A KB head comprises (1) a KB sup indicating the counting base while adding this head (2) the number of pattern-support heads (ps_heads) indicating the total number of pattern-support heads in this KB head (3) the pattern-support heads summarizing the pattern-support tables, and (4) the position of next KB head linking the next KB head so that the knowledge base can ‘grow’ incrementally.

We group all the same sized patterns in the same table so that the pattern information of desired size can be directly found through the position of pattern-support in the corresponding ps_head. The ps_head also contains a summary of the size of the patterns, the total number of counted candidates (of that size), and the total number of non-zero patterns for estimating the number of new-candidates. During the interactive process, KISP can obtain effectively the full pattern information of certain size by accessing the pattern-support table (of that size) in every KB head. The position of pattern-support, in the ps_head within a KB head, indicates the location of the pattern-support table.

Note that we keep only the patterns with non-zero support value to minimize the total size of each pattern-support table. The supports of patterns (of the same size) are stored in support-descending order in the structure. The descending ordered patterns eases the searching of valid patterns on answering an online query. An option to eliminate support sorting is writing the supports in the order of hash-tree traversal. Even when the pattern supports are directly stored without sorting, searching within the knowledge base is still more efficient than re-mining.

\[\begin{array}{|c|c|}
\hline
\text{KB sup} & \text{number of ps_heads} \\
\hline
\text{size of pattern} & \text{position of pattern-support} \\
\text{number of counted candidates} & \text{number of non-zero patterns} \\
\hline
\text{size of pattern} & \text{position of pattern-support} \\
\text{number of counted candidates} & \text{number of non-zero patterns} \\
\hline
\end{array}\]

\[\begin{array}{c}
\text{position of next KB head}
\end{array}\]

\text{: KB head \quad \text{ps_head (pattern-support head)}}

Fig. 2. Structure of the knowledge base.
Given a very low $KB.sup$ value, one might concern that the space used by $KB$ could be so large that $KISP$ might not sustain the high level of performance. Although $KB$ may increase as a result of accumulating more pattern information, $KISP$ still could efficiently answer the interactive query request with new $minsup$. We analyze the overall performance affected when $KB$ is getting very large below.

$KISP$ retrieves two kinds of data from $KB$, the $KB$ heads and the stored patterns with associated supports (i.e. pattern-support tables). Relatively small space is required by a $KB$ head for recording merely pattern summaries. Accessing these linked $KB$ heads is so easy and there is no influence. The performance could be affected only by the reading of the pattern-support tables. However, the reading is confined to qualified patterns only, instead of every pattern, in the tables. $KISP$ may sustain the good performance by skipping a large number of unqualified patterns in $KB$, even if the $KB$ is large.

The pattern-support tables are utilized to assist $KISP$ in either directly answering a query (when $KB.sup \leq minsup$) or generating the ‘new candidates’ by Theorem 1 in Section 4.2 (when $KB.sup > minsup$). In both circumstances, not every pattern needs to be scanned. Given a support-descending ordered table, when the first pattern whose support is smaller than $minsup$ is encountered, we stop reading the rest of the patterns in that table. Such an operation is also used in retrieving $S_{k-1}[KB.sup]$ for new-candidate generation. Thus, by sparing the reading of many unqualified patterns, $KISP$ may effectively retrieve the desired patterns and outperform the re-mining based approaches. In fact, $KISP$ would output all the valid patterns in constant time independent of the database size when $KB.sup \leq minsup$. Note that when patterns are stored in the hash-tree traversal order initially, we may re-arrange the tables in support-descending order, periodically or after several $KB$ heads are generated. Therefore, the overall performance affected due to a large $KB$ is quite limited.

We now examine the space utilization of $KB$, which comprises $KB$ heads and the pattern-support tables. When the requested new query with $KB.sup > minsup$ invokes new pattern generation in the interactive mining, one and only one $KB$ head will be added to $KB$. Otherwise, $KB$ stays intact because $KISP$ simply responds by retrieving patterns from $KB$. The total number of $KB$ heads hence is the total number of ‘new-pattern’ generation triggered. As described in Section 4.4, a $KB$ head contains $KB.sup$, the position of next $KB$ head, the number of $ps_heads$, and the $ps_heads$. A major portion of $KB$ is the $ps_heads$, i.e. the pattern-support tables. The others need only negligible space. The size of a pattern-support table is proportional to the number of stored patterns where a pattern takes typically 19 to 22 bytes according to our experiments (details in Table 6, Section 5.1). The size of $KB$, as a consequence, might be larger than that of the original database. Appropriate compression on
the pattern-support tables, being collections of the same sized patterns, could be employed to reduce the storage consumption for better storage utilization. Nevertheless, how compression would affect the performance needs further investigations.

5. Experimental results

In order to assess the performance of the KISP algorithm, we conducted comprehensive experiments. All experiments were performed with an 866 MHz Pentium-III PC having 1024 MB memory, running the Windows NT. In these experiments, the databases are synthetic datasets generated by the well-known method in [2]. We refer the readers to [2] for the details of the procedure. Table 1 shows the parameters and Table 2 lists the datasets used in the experiments.

5.1. Comparisons of KISP and GSP

Extensive experiments were performed to compare the execution time of KISP and GSP. The effect of using knowledge base without concurrent support counting optimization is studied first. The interactive discovery comprises five consecutive queries, with minsup values varying from 2.5% down to 0.5%.

Table 1
Parameters used in the experiments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>DB</td>
</tr>
<tr>
<td>$</td>
<td>C</td>
</tr>
<tr>
<td>$</td>
<td>T</td>
</tr>
<tr>
<td>$</td>
<td>S</td>
</tr>
<tr>
<td>$</td>
<td>I</td>
</tr>
</tbody>
</table>

Table 2
Datasets used in the experiments

| Name    | $|DB|$ | $|C|$ | $|T|$ | $|S|$ | $|I|$ | Size (MB) |
|---------|-----|-----|----|----|----|----------|
| Origin  | 100 K | 10 | 2.5 | 4 | 1.25 | 18.8     |
| Slen    | 100 K | 20 | 2.5 | 4 | 1.25 | 28.4     |
| Tlen    | 100 K | 10 | 5  | 4 | 1.25 | 28.0     |
| SPLen   | 100 K | 10 | 2.5 | 8 | 1.25 | 20.0     |
| LPLen   | 100 K | 10 | 2.5 | 4 | 2.5  | 18.5     |

Note: $|DB|$ is increased to 1000 K in the scale-up experiments.
Fig. 3 compares the relative performance of KISP and GSP on the Origin dataset with respect to various minsup. The total number of candidates and the total number of database scanning required for each query in GSP are also shown in the bottom of Fig. 3. The total execution time with KISP and GSP are 6652 and 8028 s, respectively. As to individual mining, KISP is faster than GSP for the last two queries with smaller minsup since considerable amount of candidates were eliminated. Fig. 3 also depicts the ratios of the number of candidates in GSP to those in KISP. Since the mining time reduced from the size-1 patterns in KB is very little in comparison with the pattern-outputting time increased, the overhead of KISP accounted for this phenomenon in the first three queries with larger minsup. In the first three queries, KISP runs slower than GSP due to the extra time spent for writing pattern information to KB being relatively larger than the time saved for the reduction in candidates. For instance, the mining stopped after pass two for the second query. Not much time was saved by the assistance of KB since the size-1 patterns occupied 77% of the reduced candidates.

A series of queries were applied on the datasets SPLen and LPLen to evaluate the impact of different sequence space for sampling. Similar results were obtained as shown in Fig. 4. The total execution time ratios of KISP to GSP are 89% and 93% for the datasets SPLen and LPLen, respectively. Due to the rush increase of qualified frequent 1-sequences which incurred the mass production of new candidates in the third query, the performance drops for minsup = 1% in Fig. 4. The reduction of total execution time is not apparent.

![Fig. 3. Relative execution time and number of candidates on dataset Origin.](image-url)
because KB manifests much effect on candidate reduction only for the last two queries.

Next, the distributions of customer sequences were changed. The \textit{Slen} dataset increases the average sequence size of customers (from 10 to 20), and the \textit{Tlen} dataset increases the average transaction size of customers (from 2.5 to 5). In general, both changes would allow the databases to have more (and longer) sequential patterns with respect to the above \textit{minsup} values. As indicated in Fig. 5, \textit{KISP} runs faster than \textit{GSP} for each individual mining except for the very first mining. \textit{KISP} benefits from the accumulated information so that the individual discovery could be accelerated. Take \textit{minsup} = 0.75\% for example, the execution time ratio of \textit{GSP} to \textit{KISP} is 2.9 times for dataset \textit{Tlen}. The time saved by \textit{KISP} resulted from the reduced number of candidates. In contrast, \textit{GSP} generated three times the number of candidates. Additionally, the total execution time ratios of \textit{KISP} to \textit{GSP} are 54\% for dataset \textit{Slen}, and 48\% for dataset \textit{Tlen}. To illustrate the accumulating power of KB, the number of candidates in each pass generated by \textit{GSP} and by \textit{KISP} for the \textit{Slen} dataset are enumerated in Table 3.

\textit{KISP} exhibits excellent mining capability for query intensive applications, as demonstrated in Fig. 6. The average execution time (also the time required for posterior queries) decreases as the number of queries increased. That is, users might have shorter response time in each query by decreasing \textit{minsup} value gradually to reach the desirable \textit{minsup} value, which generates the desired patterns. Similar results were obtained for the same series of queries applying on datasets \textit{Slen} and \textit{Tlen}.
All the preceding experiments were performed without optimization by concurrent support counting so that the number of database passes is the same in GSP and in KISP. Table 4 illustrates the number of database scanning reduced by concurrent support counting, and the reduced execution time for all the datasets with respect to \( \text{minsup} = 0.5\% \) and \( \text{KB.sup} = 0.75\% \). The first pass for support counting of candidate 1-sequences is not required for all minings in KISP in comparison with GSP. In general, the number of size-2 candidates is so many that the concurrent optimization is effective from the second pass of database scanning (which counts candidates of size-3 and above). However, most scans were combined in pass two so that the total number of passes and the total execution time were reduced.

When users need to find the appropriate set of patterns by reducing the number of sequential patterns found in a query, the next specified \( \text{minsup} \)
would be greater than the counting base of $KB$ ($KB.sup$). In the next experiment, all $KB.sup$s of the $KB$s were 0.5%, and 100 $minsup$s ranging from 0.5% to 2.5% were randomly selected. As shown in Table 5, the mining results are all available in very short time for all datasets. For most queries, the execution time of $KISP$ is several orders of magnitude faster than $GSP$, which always re-mines from scratch.

However, one drawback of $KISP$ is that the size of $KB$ might be larger than the size of the original database, due to the space increased for storing supports. The size of $KB$ is proportional to the number of patterns existing in $KB$. Table 6 shows that, in worst case, $KB$ might need as much as five times the space of the sequence database for low $KB.sup$.

Table 4
Effects of concurrent support counting

<table>
<thead>
<tr>
<th>$minsup = 0.5%$</th>
<th>Origin</th>
<th>Slen</th>
<th>Tlen</th>
<th>SPLen</th>
<th>LPLen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduced execution time (s)</td>
<td>29</td>
<td>94</td>
<td>157</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>Reduced number of passes</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5
Execution time of $KISP$ when $KB.sup \leq minsup$

<table>
<thead>
<tr>
<th>Execution time (s)</th>
<th>Origin</th>
<th>Slen</th>
<th>Tlen</th>
<th>SPLen</th>
<th>LPLen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>0</td>
<td>4</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Maximum</td>
<td>22</td>
<td>29</td>
<td>13</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>Average</td>
<td>4.3</td>
<td>11.8</td>
<td>10.8</td>
<td>5.1</td>
<td>4.4</td>
</tr>
</tbody>
</table>

Fig. 6. Average execution time vs. number of queries.
To assess the scalability of the proposed algorithm, several experiments were conducted. Since the basic construct of KISP is similar to that of GSP, similar scalable results could be expected. In the scale-up experiments, the total number of customers was increased from 100 to 1000 K and other parameters were the same as the Origin dataset. Again, KISP were faster than GSP for all the datasets. The execution time was normalized with respect to the time for 100,000 customers here. Fig. 7 shows that the execution time ratio of KISP increases linearly as the database size increases, which demonstrates good scalability of KISP.

6. Conclusions

In this paper, we propose an efficient knowledge base assisted mining algorithm for interactive discovery of sequential patterns. For online queries, manual tuning on mining parameters such as the minimum support is inevitable since no one can predict the best parameter and the corresponding outcome. A result driven discovery requires many times of repeated mining
in an interactive process. A fast mining algorithm that always re-mines from scratch is not good enough for interactive query in practice. Knowledge obtained from each mining should be utilized to accelerate the entire process.

The proposed KISP algorithm constructs a knowledge base for minimizing the total response time for online queries. Neither database access nor counting is required if the query result is a subset of patterns in the knowledge base. In case some resultant patterns are new to the knowledge base, we speed up the mining process by the assistance of the knowledge base. The proposed approach directly generates only the new candidates which are not counted before, concurrently counts variable sized candidates in the same database scanning, and incrementally expand the knowledge base by every counting effort for future queries. The knowledge base keeps the patterns grouped by the size to provide fast access to pattern information. The experiments performed shows that the proposed approach is faster than GSP by several orders of magnitude, with good linear scalability.

References


