CloSpan: Mining Closed Sequential Patterns in Large Datasets*

Xifeng Yan   Jiawei Han

Department of Computer Science
University of Illinois at Urbana-Champaign
{xyan, hanj}@cs.uiuc.edu

Ramin Afshar

School of Computing Science
Simon Fraser University
rafshar@cs.sfu.ca

Abstract

Previous sequential pattern mining algorithms mine the *full set* of frequent subsequences satisfying a *min_sup* threshold in a sequence database. However, since a frequent long sequence contains a combinatorial number of frequent subsequences, such mining will generate an explosive number of frequent subsequences for long patterns, which is prohibitively expensive in both time and space.

In this paper, we propose an alternative but equally powerful solution: instead of mining the complete set of frequent subsequences, we mine frequent *closed subsequences* only, i.e., those containing no super-sequence with the same support (i.e., occurrence frequency). By exploring novel global optimization techniques, an efficient algorithm, called CloSpan (Closed Sequential pattern mining) is developed, which outperforms the previous work by one order of magnitude. Moreover, CloSpan can mine really long sequences, which, to the best of our knowledge, is un-minable by previous algorithms. Finally, CloSpan produces a significantly less number of discovered sequences than the traditional (i.e., full-set) methods while preserving the same expressive power since the whole set of frequent subsequences, together with their supports, can be derived easily from our mining results.

Keywords. *Frequent pattern, sequential pattern, closed pattern, long pattern, efficiency, scalability.*

1 Introduction

Frequent sequential pattern mining is an active research theme in data mining [4, 11, 8, 14, 17, 5], with broad applications, such as discovery of motifs and tandem repeats in DNA sequences, analysis of customer shopping sequences and Web click streams, study of engineering, scientific and medical processes, and so on. Moreover, a deep understanding of efficient sequential pattern mining methods may also have strong implications on the development of efficient methods for mining frequent subtrees, lattices, subgraphs, and other structured patterns in large databases.

The sequential pattern mining algorithms developed so far have good performance in databases consisting of short frequent sequences. Unfortunately, when mining long frequent sequences, or when using very low support thresholds, the performance of such algorithms often degrades dramatically. This is not surprising: Assume the database contains only one long frequent sequence $\langle (a_1)(a_2) \ldots (a_{100}) \rangle$, it will generate $2^{100} - 1$ frequent subsequences if the minimum support is 1, although all of them except the longest one are redundant because they have the same support as that of $\langle (a_1)(a_2) \ldots (a_{100}) \rangle$.

A similar problem occurs at mining frequent itemsets [3, 9]. An interesting solution, called mining *frequent closed itemsets* [12], has been proposed to overcome this difficulty. A frequent itemset I is *closed* if there exists no superset of I with the same support in the database. There have been quite a few interesting and effective algorithms, such as CLOSET [13], MAFIA [7], CHARM [18], and CLOSET+ [16] developed for efficient mining of frequent closed itemsets. However, to the best of our knowledge, there have been no efficient methods developed for mining closed sequential patterns. This is, based on our analysis, partly because it is a pretty challenging problem.

Since mining closed subsequences shares a similar problem setting with mining closed itemsets, our first try is to use some techniques developed in closed itemset mining. Unfortunately, most of these techniques cannot work for frequent subsequence mining because subsequence testing requires ordered matching which is more difficult than simple subset testing. Nevertheless, closed itemset mining still sheds some light on this problem.

There are two approaches for mining closed or max frequent patterns: (1) greedily find the *final* closed pattern set; and (2) find a closed pattern candidate set and conduct *post-pruning* on it (for the extreme case,

do on-the-fly checking, i.e., for each newly discovered pattern, check the previous patterns to see whether this new one is closed w.r.t. discovered patterns). It looks more preferable to use the first approach because the second requires to store discovered patterns and do post-pruning; however, when the patterns become more complicated, it is difficult to guarantee that each generated pattern is closed without checking the previously discovered patterns. Thus we explore the second approach here. Our argument is that based on today's technology and our experience, it is easy to maintain a million sequences in main memory. Considering that sequences have overlapped parts, there exist compressed data structures to store them.

In this paper, we propose a fundamentally different technique from previous work. Our algorithm, called CloSpan (Closed Sequential pattern mining), develops several efficient search space pruning methods. A novel concept about the equivalence of projected databases is introduced, which can unify these optimizations in a single step. A simple condition of such equivalence is formalized. A hash-based algorithm is designed to efficiently execute the search space optimization with negligible cost. The performance of CloSpan in both synthetic datasets and real datasets shows that CloSpan not only generates a complete closed subsequence set which is substantially smaller than that generated by PrefixSpan, but also runs much faster.

CloSpan can be applied to both small and large databases: If the entire sequence database (plus some associated data structures used in CloSpan) can fit in memory, which could be common with today's hardware technology, CloSpan can be applied directly; otherwise, frequent pattern-based projection [9] can be applied first before applying CloSpan on the projected databases.

The remaining of the paper is organized as follows. Section 2 introduces the basic concepts of frequent closed sequential pattern mining problem as well as some notations used throughout the paper. A search framework is illustrated in Section 3. In Section 4, the major result about the search space pruning is presented. Section 5 formulates the algorithm of CloSpan. We report our performance study in Section 6, discuss the related work in Section 7, and conclude our work in Section 8.

2 Preliminary Concepts

Let $I = \{i_1, i_2, \ldots, i_k\}$ be a set of all items. A subset of I is called an *itemset*. A *sequence* $s = \langle t_1, t_2, \ldots, t_m \rangle$ ($t_i \subseteq I$) is an ordered list. Without loss of generality, we assume that the items in each itemset are sorted in certain order (such as alphabetic order). The *size*, $|s|$, of a sequence is the number of itemsets in the

sequence. The *length*, $l(s)$, is the total number of items in the sequence, i.e., $l(s) = \sum_{i=1}^{n} |t_i|$. A sequence $\alpha = \langle a_1, a_2, \ldots, a_m \rangle$ is a *sub-sequence* of another sequence $\beta = \langle b_1, b_2, \ldots, b_n \rangle$, denoted as $\alpha \sqsubseteq \beta$ (if $\alpha \neq \beta$, written as $\alpha \sqsubset \beta$), if and only if $\exists i_1, i_2, \ldots, i_m$, such that $1 \leqslant i_1 < i_2 < \ldots < i_m \leqslant n$ and $a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, \ldots, and\ a_m \subseteq b_{i_m}$. We also call β a *super-sequence* of α, and β *contains* α. If β *contains* α and their supports are the same, we call β *absorbs* α.

A sequence database, $D = \{s_1, s_2, \ldots, s_n\}$, is a set of sequences. Each sequence is associated with an *id*. For simplicity, say the *id* of s_i is i. $|D|$ represents the number of sequences in the database D. The *support* of a sequence α in a sequence database D is the number of sequences in D which contain α, $support(\alpha) = |\{s|s \in D\ and\ \alpha \sqsubseteq s\}|$. Given a minimum support threshold, *min_sup*, the set of **frequent sequential pattern**, FS, includes all the sequences whose support is no less than *min_sup*. The set of **closed frequent sequential pattern** is defined as follows, $CS = \{\alpha | \alpha \in FS\ and\ \nexists \beta \in FS\ such\ that\ \alpha \sqsubseteq \beta\ and\ support(\alpha) = support(\beta)\}$. Since CS includes no sequence which has a super-sequence with the same support, we have $CS \subseteq FS$. The problem of **closed sequence mining** is to find CS above a minimum support threshold. Finally, we define a database containment relation: $D \sqsubseteq D'$ means if $\exists$ an injective function $f : D \rightarrow D'$, s.t., $\forall s \in D, s \sqsubseteq f(s)$.

EXAMPLE 1. Table 1 is a sample sequence database, referred as D when the context is clear. The alphabetic order is taken as the default lexicographical order. If *min_sup* $= 2$ (taken as default in this paper), $CS = \{\langle (af)(d) \rangle : 2, \langle (af)(e) \rangle : 2, \langle (e)(a) \rangle : 3, \langle (e)(a)(b) \rangle : 2\}$ while the corresponding FS set has 16 sequences. CS has the exact same information as FS, but includes much fewer patterns. ∎

Seq ID.	Sequence
0	$\langle (af)(d)(e)(a) \rangle$
1	$\langle (e)(a)(b) \rangle$
2	$\langle (e)(abf)(bde) \rangle$

Table 1: A Sample Sequence Database D

Given a sequence $s = \langle t_1, \ldots, t_m \rangle$ and an item α, $s \diamond \alpha$ means s concatenates with α. It can be *I-Step extension* [5], $s \diamond_i \alpha = \langle t_1, \ldots, t_m \cup \{\alpha\} \rangle$ if $\forall k \in t_m, k < \alpha$; or *S-Step extension* [5], $s \diamond_s \alpha = \langle t_1, \ldots, t_m, \{\alpha\} \rangle$. For example, $\langle (ae) \rangle$ is an I-Step extension of $\langle (a) \rangle$. $\langle (a)(c) \rangle$ is an S-Step extension of $\langle (a) \rangle$. We extend the definition of item extension to sequence extension. Given two

sequences, $s = \langle t_1, \ldots, t_m \rangle$ and $p = \langle t'_1, \ldots, t'_n \rangle$, $s \diamond p$ means s concatenates with p. It can be *itemset-extension*, $s \diamond_i p = \langle t_1, \ldots, t_m \cup t'_1, \ldots, t'_n \rangle$ if $\forall k \in t_m, j \in t'_1, k < j$; or *sequence-extension*, $s \diamond_s p = \langle t_1, \ldots, t_m, t'_1, \ldots, t'_n \rangle$. If $s' = p \diamond s$, p is a *prefix* of s' and s is a *suffix* of s'. For example, $\langle (e)(a) \rangle$ is a prefix of $\langle (e)(abf)(bde) \rangle$ and $\langle (bf)(bde) \rangle$ is its suffix.

An *s-projected* database is defined as $D_s = \{p \mid s' \in D, s' = r \diamond p \text{ s.t. } r \text{ is the minimum prefix (of } s') \text{ containing } s \text{ (i.e., } s \sqsubseteq r \text{ and } \nexists r', s \sqsubseteq r' \sqsubset r)\}$. In the above definition, p can be empty. For the sample database in Table 1, $D_{\langle (af) \rangle} = \{\langle (d)(e)(a) \rangle, \langle (bde) \rangle\}$. $D_{\langle (e)(a) \rangle} = \{\$, \langle (b) \rangle, \langle (\_bf)(bde) \rangle\}$, where $\$$ means there is a sequence in D which contains $\langle (e)(a) \rangle$, but its suffix is an empty string, and $(\_bf)$ means that (bf) and the last item a in the sequence belong to the same itemset. For each suffix sequence p in D_s, the type of extension, i.e., whether s' is an itemset-extension or a sequence-extension of s, is recorded. The type of extension is useful to correctly grow s using the projected database. For simplicity, we do not explicitly represent it. Since the s-projected database will be used to mine frequent sequences which share the same prefix s, the definition of projected database can be refined in the way that it contains only *frequent* items. For example, $D_{\langle (af) \rangle} = \{\langle (d)(e) \rangle, \langle (de) \rangle\}\}$, where "$a$" in the first sequence and "b" in the second sequence are omitted because they appear only once in the projected database.

There are two kinds of projections: *physical projection* and *pseudo projection*. Physical projection requires D_s to be stored in a separate table. For pseudo projection, D_s is not physically generated: only pointers to the projected point is saved for each sequence.

3 Lexicographic Sequence Tree

In this section, we introduce the concept of *lexicographic sequence tree* [6, 2, 5] which provides a necessary background for our algorithm development.

Assume that there exists a lexicographic order in the set of all items in a database. *Set Lexicographic Order* is a linear order defined as follows. Let $t = \{i_1, i_2, \ldots, i_k\}, t' = \{j_1, j_2, \ldots, j_l\}, where\ i_1 \leqslant i_2 \leqslant \ldots \leqslant i_k\ and\ j_1 \leqslant j_2 \leqslant \ldots \leqslant j_l$. Then $t < t'$ iff either of the following is true:

1. for some $h, 0 \leqslant h \leqslant min\{k, l\}$, we have $i_r = j_r$ for $r < h$, and $i_h < j_h$, or

2. $k < l$, and $i_1 = j_1, i_2 = j_2, \ldots, i_k = j_k$.

For example, $(a, f) < (b, f)$, $(a, b) < (a, b, c)$, and $(a, b, c) < (b, c)$.

Based on this set lexicographic order, *Sequence Lexicographic Order* is given as follows: (i) if $s' = s \diamond p$,

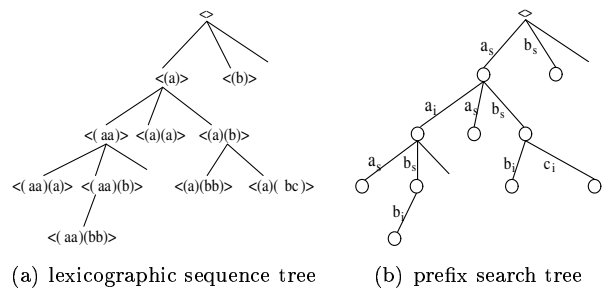(a) lexicographic sequence tree (b) prefix search tree

Figure 1: Lexicographic Sequence Tree and Prefix Search Tree

then $s < s'$; (ii) if $s = \alpha \diamond_i p$ and $s' = \alpha \diamond_s p'$, no matter what the order relation between p and p' is, $s < s'$; (iii) if $s = \alpha \diamond_i p$ and $s' = \alpha \diamond_i p'$, $p < p'$ indicates $s < s'$; and (iv) if $s = \alpha \diamond_s p$ and $s' = \alpha \diamond_s p'$, $p < p'$ indicates $s < s'$.

For example, $\langle (ab) \rangle < \langle (ab)(a) \rangle$ (i.e., a sequence is greater than its any prefix); $\langle (ab) \rangle < \langle (a)(a) \rangle$ (i.e., an itemset-extended sequence is less than sequence-extended sequence if their prefixes are the same).

A Lexicographic Sequence Tree can be constructed as follows:

1. each node in the tree corresponds to a sequence, and the root is a *null* sequence;

2. if a parent node corresponds to a sequence s, its child is either an itemset-extension of s, or a sequence-extension of s; and

3. the left sibling is less than the right sibling in sequence lexicographic order.

Figure 1(a) shows a lexicographic sequence tree. For a finite database, all the frequent sequences can be arranged in this tree. Figure 2 shows the complete search space for the sample database in Table 1 with $min\_sup = 2$. The numbers in Figure 2 represent the support of each frequent sequence. A small difference between ours and [5] is that we define the order relation among all the sequences, not only among sequences and their super-sequences. If we do pre-order transversal in the tree, an operational picture of lexicographic sequence tree is depicted in Figure 1(b). It shows that the process grows a sequence by adding one I-Step item or S-Step item. We always first perform I-Step, then S-Step, following the lexicographical order (we use a subscript "i" to denote I-Step, and "s" for S-Step).

Algorithm 1 from PrefixSpan [14] provides a general framework for depth-first search in the prefix search tree. It finds all the frequent sequences, closed or non-closed. For each discovered sequence s and its pro-
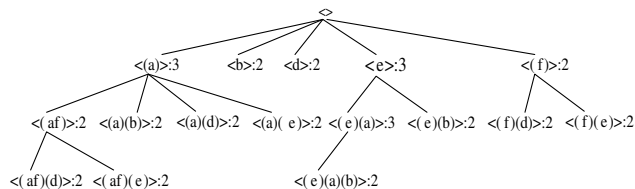
Figure 2: Lexicographic Sequence Tree for the Sample Database

jected database D_s, it performs I-Step extension (line 5) and S-Step extension (line 7) recursively until all the frequent sequences which have the prefix s are discovered. Line 3 shows the termination condition: when the number of sequences in the s-projected database is less than min_sup, it is unnecessary to extend s any more. CloSpan formulates other termination conditions to make this recursive process "return" as early as possible for closed sequence mining.

Algorithm 1 PrefixSpan(s, D_s, min_sup, F)

Input: A sequence s, a projected DB D_s, and min_sup.
Output: The frequent sequence set F.

1: insert s to F;
2: scan D_s once, find every frequent item α such that
    (a) s can be extended to $(s \diamond_i \alpha)$, or
    (b) s can be extended to $(s \diamond_s \alpha)$;
3: **if** no valid α available **then**
4:     **return**;
5: **for each** valid α **do**
6:     Call PrefixSpan($s \diamond_i \alpha$, $D_{s \diamond_i \alpha}$, min_sup, F);
7: **for each** valid α **do**
8:     Call PrefixSpan($s \diamond_s \alpha$, $D_{s \diamond_s \alpha}$, min_sup, F);
9: **return**;

---

4 Search Space Pruning and Prefix Sequence Lattice

CloSpan divides the mining process into two stages. In the first stage, a candidate set is generated. Usually this candidate set is larger than the final closed sequence set. The second stage helps eliminate non-closed sequences.

In this section, our pruning techniques for the first stage are introduced. It is possible to generate the candidate set exactly the same as CS. However, it is too expensive to do so. Our design is to make a trade-off between the size of the candidate set and the cost to compute it.

LEMMA 1. (COMMON PREFIX) *Given a subsequence s, and its projected database D_s, if $\exists \alpha$, α is a common prefix for all the sequences with the same extension type (either itemset-extension or sequence-extension) in D_s, then $\forall \beta, if\, s \diamond \beta$ is closed, α must be a prefix of β. That means $\forall \beta \sqsubset \alpha$, we need not search $s \diamond \beta$ and its descendants except the branch of $s \diamond \alpha$.*

Assume $D_s = \{\langle (d)(e)(af) \rangle, \langle (d)(e)(fg) \rangle\}$. Since all the sequences in D_s share a common prefix $\langle (d)(e) \rangle$, any sequence which begins with the prefix s but not $s \diamond \langle (d)(e) \rangle$ must not be closed. So it is unnecessary to extend $s \diamond \langle (e) \rangle$. When a common prefix α is detected in D_s, we can directly "jump" to the branch $s \diamond \alpha$ without even checking other branches below s in the search space. At the first glance, it seems that the occurrences of common prefix may not be that frequent. However, our experiments indicate that the common prefix does often take place when the threshold goes lower and the patterns turn to be longer. As observed in our experiments, more than 90% nodes in the search space can be skipped when min_sup is low. Certainly most of skipped nodes are located in the deep levels of the search space, but more frequently than people feel.

We first came up with this simple idea before we devised the salient result of CloSpan. An intermediate algorithm, CommonPrefix, has been developed which adopts the PrefixSpan framework plus the common prefix pruning technique. Although the idea is simple, the experiments indicate that it outperforms PrefixSpan by an order of magnitude for some datasets [1].

LEMMA 2. (PARTIAL ORDER) *Given a sequence s, and its projected database D_s, if among all the sequences in D_s, an item α does always occur before an item β (either in the same itemset for all sequences in D_s or in a different itemset, but not both), then $D_{s \diamond \alpha \diamond \beta} = D_{s \diamond \beta}$. Therefore, $\forall \gamma, s \diamond \beta \diamond \gamma$ is not closed. We need not search any sequence in the branch of $s \diamond \beta$.*

Let's consider the sample database in Table 1. Before projecting D into $D_{\langle (a) \rangle}$, $D_{\langle (b) \rangle}$, $D_{\langle (d) \rangle}$, $D_{\langle (e) \rangle}$, and $D_{\langle (f) \rangle}$, we find in D the first occurrence of a is always before the first occurrence of f in all the sequences. Then we need not search any sequence beginning with $\langle (f) \rangle$ because all of them will not be closed with respect to sequences beginning with $\langle (af) \rangle$. We can completely ignore the search branch of $\langle (f) \rangle$ in the prefix search space. Note that one restriction for partial order is that either the first occurrence of α and β should be in the same itemset for all the sequences in D_s or the itemset containing α is always before that containing β in all the cases. For example, $D_{\langle (af) \rangle} = \{\langle (d)(e) \rangle, \langle (de) \rangle\}$ in D, but we cannot say that

there exists a partial order between d and e because in the first sequence d and e (first occurrence) are in a different itemset, while in the second sequence they are in the same itemset. $D_{\langle(af)(e)\rangle} = \{\$, \$\}$, which is different from $D_{\langle(af)(de)\rangle} = \{\$\}$ and $D_{\langle(af)(d)(e)\rangle} = \{\$\}$. Therefore, both $\langle(af)(d)\rangle$ and $\langle(af)(e)\rangle$ are closed and should be searched separately.

Lemma 1 is a special case of Lemma 2 because partial order can do whatever common prefix can. So partial order can prune more search space than common prefix. However, the major challenge of partial order pruning is how to efficiently implement it. We once implemented two complicated algorithms to find partial orders. However, the search space that the partial order helps prune, i.e., the corresponding cost it saves, is sometimes even less than the additional effort needed to compute the partial orders. Thus the techniques developed became obsolete even before put to use. Instead, we started seeking for other pruning methods which may have efficient implementation.

Let $\mathcal{I}(D)$ represent the total number of items in D, defined as

$$\mathcal{I}(D) = \sum_{i=1}^{n} l(s_i).$$

We call $\mathcal{I}(D)$ the *size of the database*. For the sample dataset in Table 1, $\mathcal{I}(D) = 15$.

THEOREM 1. (EQUIVALENCE OF PROJECTED DATABASES) *Given two sequences, s, s', $s \sqsubseteq s'$, then*

$$(4.1) \qquad D_s = D_{s'} \Leftrightarrow \mathcal{I}(D_s) = \mathcal{I}(D_{s'})$$

Proof. *It is obvious that $D_s = D_{s'} \Rightarrow \mathcal{I}(D_s) = \mathcal{I}(D_{s'})$. Now we prove the sufficient condition. Since $s \sqsubseteq s'$, then $D_{s'} \sqsubseteq D_s$ and $\mathcal{I}(D_{s'}) \leqslant \mathcal{I}(D_s)$. The equality between $\mathcal{I}(D_{s'})$ and $\mathcal{I}(D_s)$ holds only if $\forall \gamma \in D_{s'}$, $\gamma \in D_s$, and vice vera. Therefore, $D_s = D_{s'}$.* ∎

For the sample database in Table 1, $D_{\langle(af)\rangle} = D_{\langle(f)\rangle} = \{\langle(d)(e)\rangle, \langle(de)\rangle\}$ (physical projection), and $\mathcal{I}(D_{\langle(af)\rangle}) = \mathcal{I}(D_{\langle(f)\rangle}) = 4$.

Based on Theorem 1, the following search space pruning can be achieved.

LEMMA 3. (EARLY TERMINATION BY EQUIVALENCE) *Given two sequences, $s \sqsubseteq s'$ and also $\mathcal{I}(D_s) = \mathcal{I}(D_{s'})$, then $\forall \gamma, support(s \diamond \gamma) = support(s' \diamond \gamma)$.*

Considering the previous example, we have $\mathcal{I}(D_{\langle(af)\rangle}) = \mathcal{I}(D_{\langle(f)\rangle})$ and both $\langle(af)(d)\rangle$ and $\langle(af)(e)\rangle$ are frequent. Then we can conclude that the support of $\langle(af)(d)\rangle$ and $\langle(f)(d)\rangle$, $\langle(af)(e)\rangle$ and $\langle(f)(e)\rangle$ are

the same without knowing the support of $\langle(f)(d)\rangle$ and $\langle(f)(e)\rangle$.

It is recognized that if s and all of its descendants $(s \diamond \gamma)$ have been discovered, it is unnecessary to search the branches under s' in the prefix search tree. Instead, s and s' share exactly the same descendants in the prefix search tree. So we can directly transplant the branch under s to s' with small modification of extension types. The power for such transplanting is that only two operations needed to detect such condition: first, containment ($s \sqsubseteq s'$); second, $\mathcal{I}(D_s)$ comparison ($\mathcal{I}(D_s) = \mathcal{I}(D_{s'})$). Since $\mathcal{I}(D_s)$ is just a number and can be produced as a side-product (using a counter) when we project the database, the computation cost is nearly negligible. We define *projected database closed set*, $LS = \{s \mid support(s) \geqslant min\_sup \text{ and } \nexists s', s.t. \ s \sqsubseteq s' \text{ and } \mathcal{I}(D_s) = \mathcal{I}(D_{s'})\}$. We have $CS \subseteq LS \subseteq FS$. In our algorithm, instead of mining CS directly, it first produces the complete set of LS, then it applies the non-closed sequence elimination in LS to generate the exact set of CS. Based on Lemma 3, efficient search space pruning methods are developed to detect Early Termination condition.
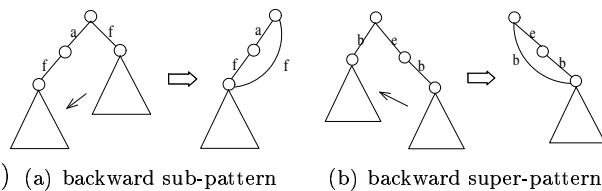

(a) backward sub-pattern   (b) backward super-pattern

Figure 3: Backward Sub-Pattern and Super-Pattern

COROLLARY 1. (BACKWARD SUB-PATTERN) *If a sequence $s < s'$ and $s \sqsupset s'$, the condition of $\mathcal{I}(D_s) = \mathcal{I}(D_{s'})$ is sufficient to stop searching any descendant of s' in the prefix search tree.*

We call s' a *backward sub-pattern* of s if $s < s'$ and $s \sqsupset s'$ (s' is discovered after s). For the sample database in Table 1, if we know $\mathcal{I}(D_{\langle(f)\rangle}) = \mathcal{I}(D_{\langle(af)\rangle})$, we can conclude that $D_{\langle(f)\rangle} = D_{\langle(af)\rangle}$. We even need not compare the sequences in $D_{\langle(f)\rangle}$ and $D_{\langle(af)\rangle}$ one by one to determine whether they are the same. This is the advantage of only comparing their size, just as proved in Theorem 1. If their size is equal, we can conclude $D_{\langle(f)\rangle} = D_{\langle(af)\rangle}$. For $\langle(f)\rangle$, it has two frequent children d and e in this case. We need not grow $\langle(f)\rangle$ anymore since all the children of $\langle(f)\rangle$ are the same as that of $\langle(af)\rangle$ and vice versa under the condition of $D_{\langle(f)\rangle} = D_{\langle(af)\rangle}$. Moreover, their supports are the same. Therefore, any sequence beginning with $\langle(f)\rangle$ **is absorbed** by the sequences

beginning with $\langle(af)\rangle$. Figure 3(a) shows that their subtrees (descendant branches) can be merged into **one** without mining the subtree under $\langle(f)\rangle$.

COROLLARY 2. (BACKWARD SUPER-PATTERN) *If a sequence $s < s'$ and $s \sqsubset s'$, if the condition of $\mathcal{I}(D_s) = \mathcal{I}(D_{s'})$ holds, it is sufficient to transplanting the descendants of s to s' instead of searching any descendant of s' in the prefix search tree.*

We call s' a *backward super-pattern* of s if $s < s'$ and $s \sqsubset s'$ (s' is discovered after s). For example, if we know $\mathcal{I}(D_{\langle(b)\rangle}) = \mathcal{I}(D_{\langle(e)(b)\rangle})$, we can conclude that $D_{\langle(e)(b)\rangle} = D_{\langle(b)\rangle}$. There is no need to grow $\langle(e)(b)\rangle$ since all the children of $\langle(b)\rangle$ are the same as that of $\langle(e)(b)\rangle$ and vice versa, with the same support. Therefore, the sequences beginning with $(e)(b)$ **can absorb** any sequence beginning with (b). Figure 3(b) shows that their subtrees can be merged into **one** without discovering the subtree under $\langle(e)(b)\rangle$.

A further analysis of Lemma 2 shows that Corollaries 1 and 2 cover the situation that Lemma 2 can prune. Let us look at the previous case of $D_{\langle(f)\rangle} = D_{\langle(af)\rangle}$ in detail. For Partial Order, we find that a always occurs before f, so we need not generate the projected database with the prefix of $\langle(f)\rangle$. For Corollary 1, we have to generate $D_{\langle(f)\rangle}$ and compare its size with $D_{\langle(af)\rangle}$. Then we find that they are equal, and thus stop searching the branch of $\langle(f)\rangle$. It seems that one step projection is wasted in this case; however, the advantage is that there is an efficient algorithm to implement "Early Termination by Equivalence". Furthermore, it can prune much larger search space than Partial Order can.
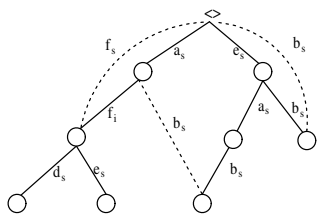

Figure 4: A Partial Prefix Sequence Lattice for D

The above discussion indicates that by subtree merging under the condition of early termination, a prefix search tree can be replaced with a **Prefix Sequence Lattice**. Fig. 4 shows a part of prefix sequence lattice for the sample database in Table 1 by merging the corresponding subtrees in Figure 2. We have $\mathcal{I}(D_{\langle(e)(b)\rangle}) = \mathcal{I}(D_{\langle(b)\rangle})$, $\mathcal{I}(D_{\langle(e)(a)(b)\rangle}) = \mathcal{I}(D_{\langle(a)(b)\rangle})$, and $\mathcal{I}(D_{\langle(af)\rangle}) = \mathcal{I}(D_{\langle(f)\rangle})$ as illustrated in Fig. 4. Prefix sequence lattice is not only a search space, but also

an internal data structure that stores the LS set. The mining result of CloSpan in the first stage is a prefix sequence lattice. The dotted links between nodes in Figure 4 are not saved.

5 CloSpan: Design and Implementation

In this section, we formulate our CloSpan based on the early termination techniques. CloSpan can be outlined as two major steps: (1) it generates the LS set, a superset of closed frequent sequences, and stores it in a prefix sequence lattice; and (2) it does post-pruning to eliminate non-closed sequences.

Algorithm 2 ClosedMining(D, *min_sup*, L)

Input: A database D_s, and *min_sup*.
Output: The complete closed sequence set L.

1: remove infrequent items and empty sequences, and sort each itemset of a sequence in D_s;
2: $S^1 \leftarrow$ all frequent 1-item sequence;
3: $S \leftarrow S^1$;
4: **for each** sequence $s \in S^1$ **do**
5:     CloSpan(s, D_s, *min_sup*, L);
6: eliminate non-closed sequences from L;

Algorithm 2, ClosedMining, illustrates the framework which includes the necessary preprocessing step. It first sorts every itemset and removes infrequent items and empty sequences. Then it calls CloSpan recursively by doing depth-first search on the prefix search tree and building the corresponding prefix sequence lattice. Finally, it eliminates non-closed sequences. Algorithm 3, CloSpan, is similar to PrefixSpan, however, it performs a major improvement using the search space pruning techniques developed above. That is, before exploring a discovered sequence and its corresponding projected database to mine its successive super-sequences, CloSpan first checks whether a discovered sequence s' exists, s.t. either $s \sqsubseteq s'$ or $s' \sqsubseteq s$, and $\mathcal{I}(D_s) = \mathcal{I}(D_{s'})$. If the condition is satisfied, based on Lemma 3, it is unnecessary to continue expansion since all its possible descendants have been discovered before. Algorithm 3 outlines the pseudo code of CloSpan.

Now one problem remains: how to do line 1-4 of Algorithm 3 efficiently. There are two approaches to check the condition of Theorem 1 since the condition has two components: (1) the containment, and (2) the size of projected database. The containment testing is involved with a large testing space. If we first check the containment, i.e., finding all the sequences which are sub-sequences or super-sequences of the current sequence, it is expensive. Although when a new sequence

171

Algorithm 3 CloSpan(s, D_s, min_sup, L)

Input: A sequence s, a projected DB D_s, and min_sup.
Output: The prefix search lattice L.

1: Check whether a discovered sequence s' exists s.t.
    either $s \sqsubseteq s'$ or $s' \sqsubseteq s$, and $\mathcal{I}(D_s) = \mathcal{I}(D_{s'})$;
2: **if** such super-pattern or sub-pattern exists **then**
3:     modify the link in L, **return**;
4: **else** insert s into L;
5: Scan D_s once, find every frequent item α such that
    (a) s can be extended to $(s \diamond_i \alpha)$, or
    (b) s can be extended to $(s \diamond_s \alpha)$;
6: **if** no valid α available **then**
7:     **return**;
8: **for each** valid α **do**
9:     Call CloSpan($s \diamond_i \alpha$, $D_{s \diamond_i \alpha}$, min_sup, L);
10: **for each** valid α **do**
11:     Call CloSpan($s \diamond_s \alpha$, $D_{s \diamond_s \alpha}$, min_sup, L);
12: **return**;

Algorithm 4 checkProjectedDBSize(s, k, H)

Input: A sequence s, its key k, and a hash table H
Output: An updated hash table H

0: $l_{sup} \leftarrow \varnothing, l_{sub} \leftarrow \varnothing$;
1: index the hash table with the key k;
2: find a list of pairs $\langle k, s' \rangle$;
3: **for each** pair $\langle k, s' \rangle$ **do**
4:     **if** $support(s) = support(s')$ **then**
5:         **if** $s' \sqsubseteq s$ **then** $l_{sup} \leftarrow l_{sup} \cup \{\langle k, s' \rangle\}$;
6:         **if** $s \sqsubseteq s'$ **then** $l_{sub} \leftarrow l_{sub} \cup \{\langle k, s' \rangle\}$;
7: **if** l_{sup} not empty **then**
    remove all pairs in l_{sup} from H;
    merge descendant subtrees (of s' in l_{sup}) in L^1;
8: **if** l_{sub} not empty **then**
    merge descendant subtrees (of s' in l_{sub}) in L;
    return;
9: insert $\langle k, s \rangle$ into H;

is extended from the current sequence, its sub-sequence and super-sequence set can be directly computed from the current set, it is still costly based on our testing. Thus, we devised an alternative approach which uses a hash index on the size of projected database. Then only the sequences whose projected database size is the same as that of the current sequence are tested. We found this approach significantly improves the performance and makes the cost of such checking nearly negligible compared to the total running time.
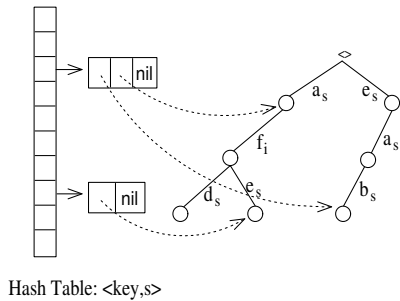

Hash Table: <key,s>

Figure 5: Hashing for Fast Condition Checking

The idea is illustrated in Figure 5. Basically, the hash table uses $\mathcal{I}(D_s)$ as a hash key and store the following pair $\langle \mathcal{I}(D_s), s \rangle$ in the hash table. When a new sequence s comes in, the hash value of $\mathcal{I}(D_s)$ is calculated to index it in the hash table. Then we check the hash table to see whether the value of $\mathcal{I}(D_s)$ already exists. If it does, for example, s' has the same projected

database size as s, we then check whether $s \sqsubseteq s'$ or $s' \sqsubseteq s$. If the former is true, we do not add $\langle \mathcal{I}(D_s), s \rangle$ into the hash table. If the latter is true, we replace $\langle \mathcal{I}(D_{s'}), s' \rangle$ with $\langle \mathcal{I}(D_s), s \rangle$. In our implementation, we do not put the whole sequence into the hash table, instead, we only record a pointer which points to the corresponding node in the prefix sequence lattice as shown in Figure 5. The sequence can be retrieved if we traverse the path from this node to the root. This procedure of hashing and checking is outlined in Algorithm 4, which corresponds to line 1-4 in Algorithm 3.

Line 7 of Algorithm 4 can discover the condition of backward super-pattern, $s' \sqsubseteq s$. It removes all satisfying $\langle \mathcal{I}(D_{s'}), s' \rangle$ pairs in the hash table and merges their corresponding descendant subtrees in L, which means deleting the duplicate subtrees produced by s' and retaining only one such tree for the new super-pattern s so that s need not grow any descendant.

The case for $s \sqsubseteq s'$ in line 8, backward sub-pattern, is a little bit different. There may exist several sequences s' such that $s \sqsubseteq s'$. For example, it is possible that

$$(5.2) \qquad D_{\langle (a)(d)(c) \rangle} = D_{\langle (b)(d)(c) \rangle} = D_{\langle (d)(c) \rangle}$$

In this case, $\langle (d)(c) \rangle$ shares the same descendants with $\langle (a)(d)(c) \rangle$ and $\langle (b)(d)(c) \rangle$. However, before the size of projected database of $\langle (d)(c) \rangle$ is known, we cannot make any valid conclusion that $\langle (a)(d)(c) \rangle$ and

---

[1] Subtrees are the trees below the node that s' points in L. s' is from the pairs in l_{sup}.

$\langle(b)(d)(c)\rangle$ have the exactly same descendant set just because $\mathcal{I}(D_{\langle(a)(d)(c)\rangle})$ and $\mathcal{I}(D_{\langle(b)(d)(c)\rangle})$ are equal. The reason is $\mathcal{I}(D_{\langle(a)(d)(c)\rangle}) = \mathcal{I}(D_{\langle(b)(d)(c)\rangle})$ does not imply $D_{\langle(a)(d)(c)\rangle} = D_{\langle(b)(d)(c)\rangle}$. When we find equation (5.2) holds, one of duplicated subtrees from $\langle(a)(d)(c)\rangle$ and $\langle(b)(d)(c)\rangle$ can be eliminated. For another instance, if it is discovered that $D_{\langle(a)(b)(c)\rangle} = D_{\langle(b)(c)\rangle} = D_{\langle(f)(b)(c)\rangle}$. Not only one of duplicated subtrees from $\langle(a)(b)(c)\rangle$ and $\langle(b)(c)\rangle$ can be eliminated, but also it is unnecessary to grow $\langle(f)(b)(c)\rangle$ as indicated by Lemma 3. Unfortunately, in our current implementation, this situation cannot be detected because $\langle(b)(c)\rangle$ has been absorbed by $\langle(a)(b)(c)\rangle$. When $\langle(f)(b)(c)\rangle$ comes, it cannot contain or be contained by $\langle(a)(b)(c)\rangle$ while the trace of $\langle(b)(c)\rangle$ also disappears. This difficulty makes our current approach cannot cover all the situations in Lemma 3. However, its implementation is succinct and efficient.

For this hash approach, the performance is related to how often the following situation happens: Two sequences do not have any containment relationship, but their projected database size is equal. The projected database size, theoretically, can range from 0 to $\mathcal{I}(D)$ (D is the original whole database). If the values of $\mathcal{I}(D_s)$ for lots of s are dense in a small range, the performance will degrade. We use another hash key which has a larger value distribution. Remember the core of equivalence, $D_s = D_{s'} \Leftrightarrow \mathcal{I}(D_s) = \mathcal{I}(D_{s'})$, is $D_s = D_{s'}$. Therefore, any necessary propositions of holding $D_s = D_{s'}$ can be used as a part of hash key in order to make the key more uniformly distributed. The following are some of these propositions. Given an s, $D = \{s_1, s_2, \ldots, s_n\}, D^s = \{s_{i_1}, s_{i_2}, \ldots, s_{i_m}\}, 1 \le i_1 < i_2 < \ldots < i_m \le n, s \sqsubseteq s_{i_j}, j = 1..m$, we assign a random number to each sequence in D as a signature ($r_i = random(s_i)$),

1. $\mathcal{I}(D_s)$.

2. $support(s)$.

3. $\sum_{j=1}^{m}(i_j)$, a sum of sequences' identifiers, denoted by $\mathcal{T}(D_s)$.

4. $\sum_{j=1}^{m}(r_{i_j})$.

5. $\mathcal{I}(D_s) + \sum_{j=1}^{m}\sum_{k=i_j+1}^{n} l(s_k)$ (pseudo projection) or $\mathcal{I}(D_s) + \sum_{j=1}^{m-1}\sum_{k=j+1}^{m} l(r_k)$ (physical projection, let $D_s = \{r_1, r_2, \ldots, r_m\}$).

We denote $\mathcal{I}(D_s) + \sum_{j=1}^{m}\sum_{k=i_j+1}^{n} l(s_k)$ by $\mathcal{L}(D_s)$. The intuitive explanation of $\mathcal{L}(D_s)$ is that $\mathcal{L}(D_s)$ sums up the distance between the start position of a projected sequence and the end position of the whole database in pseudo projection. For the sample database in Table 1, $\mathcal{I}(D_{\langle(af)\rangle}) = 2 + 2 = 4$, $\mathcal{L}(D_{\langle(af)\rangle}) = 4 + 2 = 6$ (physical projection), and $\mathcal{L}(D_{\langle(af)\rangle}) = 6 + 3 = 9$

(pseudo projection). This proposition is a little bit different from the first fours. In fact, we have if $s \sqsubseteq s'$,

$$\mathcal{L}(D_s) = \mathcal{L}(D_{s'}) \Leftrightarrow \mathcal{I}(D_s) = \mathcal{I}(D_{s'})$$

We use the fifth proposition as a replacement of $\mathcal{I}(D_s)$. There are two reasons: first, $\mathcal{L}(D_s)$ is easy to compute since the end position of the database is always known; second, it generates a better distribution of hash keys since it not only includes the size of the projected database, but also contains the information of sequence ids. We also think the propositions 3 and 4 besides 1 and 2 are good candidates to be added to prune some candidates inside the loop (line 4, Algorithm 4).

With this design, we have a pretty good key distribution. For the same key value or the same hash value of different keys, probably there exist several candidates. By comparing the above propositions, it can quickly filter out some potential invalid candidates without doing containment test. After that, a final containment test is executed to find $s \sqsubseteq s'$ or $s' \sqsubseteq s$. In order to have $\mathcal{I}(D_s) = \mathcal{I}(D_{s'})$, s and s' must share the same last item. We also use this simple property to prune some invalid candidates.

5.1 Non-Closed Sequence Elimination The remaining task is to eliminate non-closed sequences from the prefix sequence lattice. The problem is to check out for each sequence s, whether there exists a supersequence s' s.t. $support(s) = support(s')$. Obviously a naïve algorithm, which compares each sequence with other sequences in the lattice, does not work because of its $O(N^2)$ complexity. We adopt the fast subsumption checking algorithm introduced by Zaki [18], which shares the same data structure in Figure 5. It uses $support$ of a sequence as its hash function. CloSpan first finds all the sequences that have the same support of s, then it checks whether there is a super-sequence containing s. Since the value of $support$ is very dense, we need some other hash key to make the key distribution sparse. In fact, if $s \sqsubseteq s'$ and $support(s) = support(s')$, their corresponding sequences' id sum should be the same, i.e. $\mathcal{T}(D_s) = \mathcal{T}(D_{s'})$. Because the value of $\mathcal{T}(D_s)$ is distributed much sparse, [18] proposes using $\mathcal{T}(D_s)$ as its hash key instead of using $support$. However, since the equivalence of $\mathcal{T}(D_s)$ does not imply the equivalence of $support$, for the sequences that have the same $\mathcal{T}(D_s)$ value, their $supports$ have to be checked to eliminate invalid candidates. Finally, the containment is tested to see whether a sequence can be absorbed. This elimination design is similar to our design for early termination by equivalence. The advantage is that the hash key is easy to compute and is good at reducing the search space.

6 Performance Study

A comprehensive performance study has been conducted in our experiments on both synthetic and real world datasets. We use a synthetic data generator provided by IBM and a click stream dataset from KDDCup2000. The synthetic dataset generator can be retrieved from an IBM website, http://www.almaden.ibm.com/cs/quest. It can accept parameters like the number of sequences (customers), the average number of itermsets (transactions) in each sequence (customer), the average number of items (products) in each itemset (transaction), and the number of different items in the dataset. It also allows the users to define parameters of frequent patterns in the dataset and their correlation. Table 2 shows some major parameters in this generator and their meanings. More details can be referred in [4]. The performance of three algorithms are compared: PrefixSpan, CommonPrefix, and CloSpan.

abbr.	meaning
D	Number of sequences in 000s
C	Average itemsets per sequence
T	Average items per itemset
N	Number of different items in 000s
S	Average itermsets in maximal sequences
I	Average items in maximal sequences

Table 2: Parameters for IBM Quest Data Generator

All experiments are done on a 1.7GHZ Intel Pentium-4 PC with 1GB main memory, running Windows XP Professional. All three algorithms are written in C++ with STL library support and compiled by g++ in cygwin environment with -O3 optimization.

Figure 6 shows the performance and mining result for the dataset D10C10T2.5N10S6I2.5 (-seq.npats 2000 -lit.npats 5000). Figure 6(a) illustrates the running time. Overall, CloSpan outperforms CommonPrefix while CommonPrefix is much faster than PrefixSpan. PrefixSpan even cannot complete the job below the minimum support of 0.001 due to too long running time. Figure 6(b) shows the distribution of discovered frequent closed patterns in terms of their length. It is reasonable to see with the decreasing minimum support, the maximum length of frequent closed sequences grows larger. Figure 6(c) shows the number of frequent sequences which are discovered and checked in order to generate the frequent closed sequence set. This number is roughly equal to how many times the inner procedure (CloSpan) is called and how many times projected databases are generated. Surprisingly, this

number accurately predicates the total running time as the great similarity shows between Figure 6(a) and 6(c). Therefore, for the same dataset, the number of checked frequent sequences approximately determines the performance.

Figure 7 shows a dataset with larger parameters of C, T, S, and I. That means each transaction and sequence are longer, and the patterns also turn to be longer, which can be concluded by comparing Figure 7(b) ($\sigma = 0.006$) and Figure 6(b) ($\sigma = 0.003$). Therefore, it is much more difficult to mine this dataset with the same minimum support threshold as the previous one. However, CloSpan still has a similar performance improvement opposed to CommonPrefix and PrefixSpan.

We then test the performance of these three algorithms as some major parameters in the synthetic data generator are varied. The impact of different parameters is presented on the running time of each algorithm. We select the parameters shown in Table 2 as varied ones: the number of sequences in the dataset, the average number of itemsets per sequence, and the average number of items per itemset. For each experiment, only one parameter varies with the others fixed. The experimental results are shown in Figure 8. We also discovered in other experiments, the speed-up decreases when the number of different items (N) in the dataset goes down. However, it is still faster than PrefixSpan.

The gazelle dataset comes from click-stream data from gazelle.com, which no longer exists. The dataset was once used in KDDCup-2000 competition and is now available through the website: http://www.ecn. purdue.edu/KDDCUP. [10] describes the background information about this dataset. Basically the original data includes a set of page views (each page contains a specific product information) in a legwear and legcare website. Each session contains page views done by a customer over a short period. Product pages viewed in one session are considered as an itemset, and different sessions for one user is considered as a sequence. We use the combination of the productID and AssortmentID as the product code. SessionID is considered to identify items in one itemset. For linking itemsets to create a sequence we use the cookieID. The database contains 1423 different products and assortments which are viewed by 29369 different users. There are total 29369 sequences, 35722 sessions, and 87546 page views. The average number of sessions in a sequence is around 1. The average number of pageviews in a session is 2. The largest session contains 342 views, the longest sequence has 140 sessions, and the largest sequence contains 651 page views. Thus each session and each sequence is short on average, but this dataset does have very long sequences.
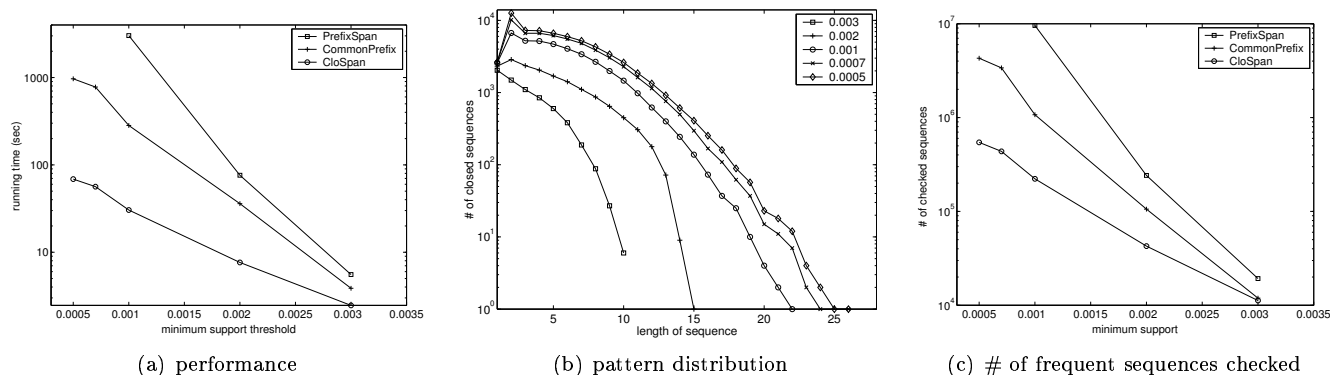
Figure 6: Varying Support for Dataset D10C10T2.5N10S6I2.5 (-seq.npats 2000 -lit.npats 5000))
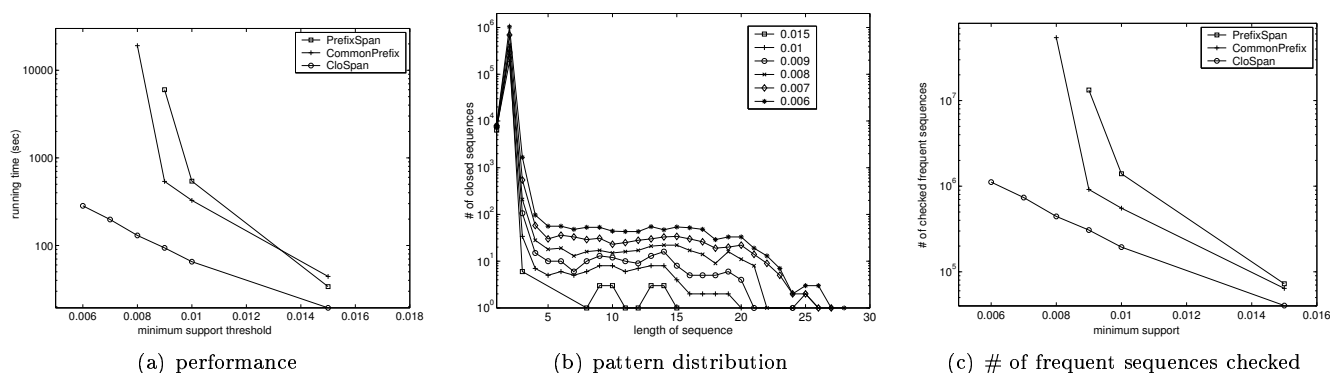

(a) performance (b) pattern distribution (c) # of frequent sequences checked

Figure 7: Varying Support for Dataset D5C20T20N10S20I20

Figure 9(a) shows the runtime with *min_sup* varying from 0.03%-0.013%. The distribution of the length of the discovered sequences is shown in Figure 9(b). The longest sequence has the length of 57, which is unminable using previous algorithms.

All the experiments show that CloSpan outperforms PrefixSpan by over one order of magnitude when the minimum support is low and the length of patterns is long. CloSpan also demonstrates a better scalability over PrefixSpan and CommonPrefix since it succeeds in completing the mining process with larger database. Therefore, CloSpan achieves our original design goal. That is, it not only efficiently works out the final result, but also outperforms the underlying mining algorithm without changing its framework.

7 Related Work

Previous studies have developed some efficient techniques for mining sequential patterns and closed frequent itemsets, which are related to this study. MaxMiner [6] is an Apriori-based [3], level-wise, breadth-first search method to find *max-itemset* (an itemset is a *max-itemset* if it is frequent but none of its supper-

pattern is). It performs *superset frequency pruning* and *subset infrequency pruning* for search space reduction. Other works for mining frequent or closed patterns including DepthProject [2], MAFIA [7], CLOSET [13], and CHARM [18] adopt space-efficient depth-first search. These studies show that depth-first search is time-efficient especially in case when the database can be put into main memory. DepthProject proposes heuristic rules for selective projection to reduce the database size. Also, an efficient counting technique is proposed in DepthProject. MAFIA uses the vertical bitmap to compress transaction id (*tid*) list, thus improves the counting efficiency. Algorithms like CLOSET and CHARM mine closed frequent itemsets in transactions. CLOSET recursively projects the database in each level and uses a compressed database representation called *FP-tree* to mine closed patterns. CHARM develops a compact vertical tid list structure called *diffset* which only records the difference in the tid list of a candidate pattern from its prefix pattern. A fast hash-based approach is also used in CHARM to prune non-closed patterns.

In sequential pattern mining, efficient algorithms like SPADE [17], PrefixSpan [14], and SPAM [5] were
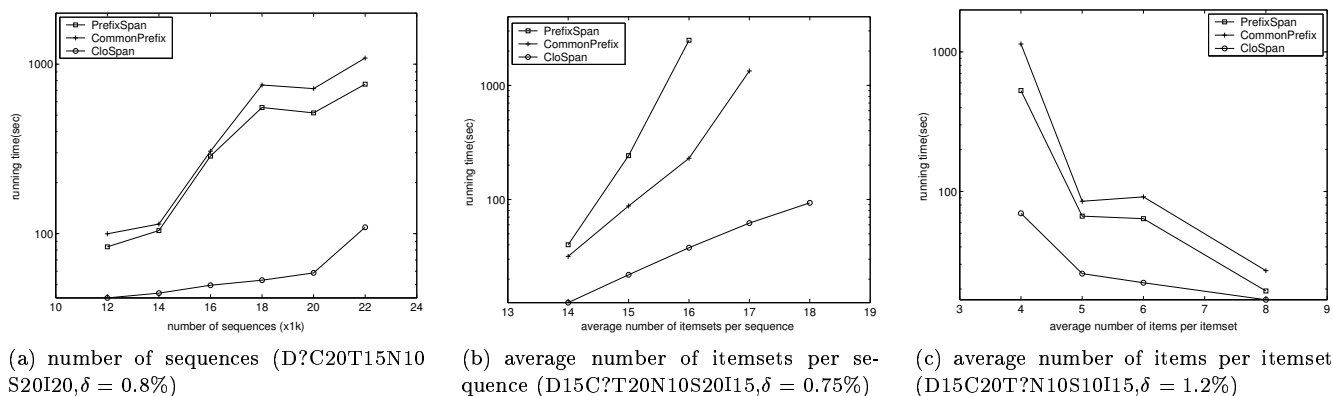
(a) number of sequences (D?C20T15N10 S20I20,$\delta = 0.8\%$)

(b) average number of itemsets per sequence (D15C?T20N10S20I15,$\delta = 0.75\%$)

(c) average number of items per itemset (D15C20T?N10S10I15,$\delta = 1.2\%$)

Figure 8: Performance vs. Varying Parameters

(a) performance

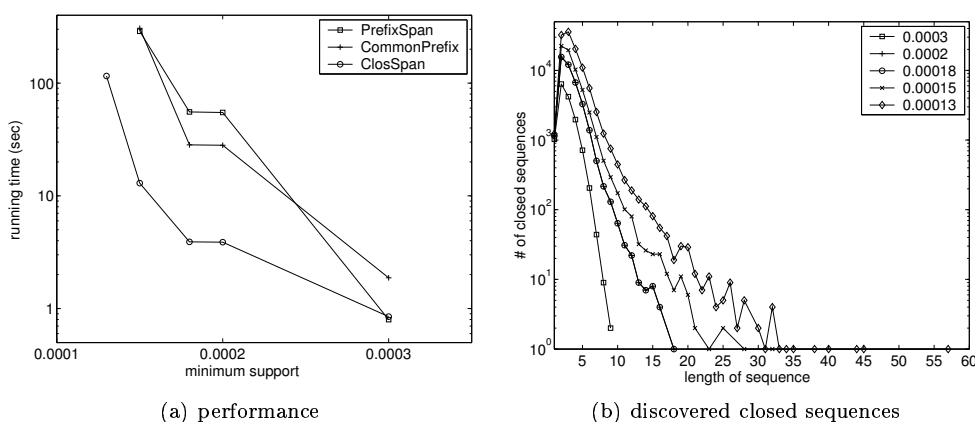(b) discovered closed sequences

Figure 9: Gazelle Click Stream

developed. All of them adopt depth-first search (SPADE has breadth-first search option). SPADE proposes a vertical id-list database format and performs frequent sequences enumeration through a simple join on id-lists. PrefixSpan proposes using projected database to accelerate the mining process. SPAM uses vertical bitmap representation of database for efficient candidate generation and support counting. [5] shows on small datasets, PrefixSpan runs faster. But on large datasets, SPAM outperforms PrefixSpan and SPADE. However, SPAM has much higher space consumption than the other two methods.

In principle, CloSpan is not directly comparable with the traditional sequential pattern mining algorithms, since CloSpan mines *closed* sequential patterns whereas the algorithms listed above mine the nonclosed ones. Direct mining of closed patterns leads to much fewer patterns, especially when the patterns are long (often lead to savings by over one order of magnitude when patterns are long or the threshold is low—considering our initial example of pattern length 100,

the contrast is obvious), but it has the same expressive power compared with the traditional sequential pattern mining algorithms. Based on the performance curves reported in these papers [5, 18] and the explosive number of subsequences generated for long sequences, it is expected that CloSpan will outperform SPADE and SPAM when the patterns to be mined are long and the database is large.

8 Conclusions

In this paper, we investigated issues for mining closed frequent sequential patterns in large data sets and addressed the possible inefficiency and redundancy of frequent sequential pattern mining problem. We introduced a new lexicographic ordering system and formulated CloSpan to mine frequent closed sequences efficiently. To the best of our knowledge, this is the first piece of work to solve closed sequential pattern mining problem. CloSpan outperforms PrefixSpan by more than one order of magnitude and is capable of mining longer frequent sequences in a large data set with low mini-

mum support without information loss. CloSpan adopts a novel pruning technology, and thus provides a new insight for scalable mining of long patterns.

Since the search space pruning does not modify the underlying frequent pattern mining algorithm, and it only defines the early termination condition of search branches, this method can be extended to other existing well-known sequential pattern mining algorithms like SPADE and SPAM. We analyze the structure of vertical bitmap in SPAM and find it is feasible to calculate the corresponding size of projected database in SPAM with a little additional cost. We speculate that it can achieve similar performance gain if our pruning algorithm is applied there.

There are many interesting research problems related to CloSpan that should be pursued further. For example, how to take full advantage of the search space pruning property proposed in this paper, how to incorporate user-specified constraints [8, 15] in the mining of closed sequential patterns, and how to extend CloSpan to mining other complicated structured patterns are interesting problem for future research.

References

[1] R. Afshar. Mining frequent max, and closed sequential patterns. *M.Sc. Thesis*, School of Computing Science, Simon Fraser University, Aug. 2002.

[2] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. Depth-first generation of large itemsets for association rules. In *IBM Technical Report RC21538*, July 1999.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases (VLDB'94)*, pages 487–499, Santiago, Chile, Sept. 1994.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. 1995 Int. Conf. Data Engineering (ICDE'95)*, pages 3–14, Taipei, Taiwan, Mar. 1995.

[5] J. Ayres, J. E. Gehrke, T. Yiu, and J. Flannick. Sequential pattern mining using bitmaps. In *Proc. 2002 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'02)*, Edmonton, Canada, July 2002.

[6] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98)*, pages 85–93, Seattle, WA, June 1998.

[7] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 443–452, Heidelberg, Germany, April 2001.

[8] M. Garofalakis, R. Rastogi, and K. Shim. SPIRIT: Sequential pattern mining with regular expression constraints. In *Proc. 1999 Int. Conf. Very Large Data Bases (VLDB'99)*, pages 223–234, Edinburgh, UK, Sept. 1999.

[9] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00)*, pages 1–12, Dallas, TX, May 2000.

[10] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2:86–98, 2000.

[11] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proc. 1995 Int. Conf. Knowledge Discovery and Data Mining (KDD'95)*, pages 210–215, Montreal, Canada, Aug. 1995.

[12] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. 7th Int. Conf. Database Theory (ICDT'99)*, pages 398–416, Jerusalem, Israel, Jan. 1999.

[13] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *Proc. 2000 ACM-SIGMOD Int. Workshop Data Mining and Knowledge Discovery (DMKD'00)*, pages 11–20, Dallas, TX, May 2000.

[14] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 215–224, Heidelberg, Germany, April 2001.

[15] J. Pei, J. Han, and W. Wang, Constraint-based sequential pattern mining in large databases In *Proc. 2002 Int. Conf. Information and Knowledge Management (CIKM'02)*, pages 18-25, McLean, VA, Nov. 2002.

[16] J. Wang, J. Han, and J. Pei. Closet+: Scalable and space-saving closed itemset mining. *Submitted for publication*, Feb. 2003.

[17] M. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 40:31–60, 2001.

[18] M. J. Zaki and C. J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *Proc. 2002 SIAM Int. Conf. Data Mining*, pages 457–473, Arlington, VA, April 2002.