

Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 02-023

SLPMiner: An Algorithm for Finding Frequent Sequential Patterns
Using Length-Decreasing Support Constraint

Masakazu Seno and George Karypis

June 05, 2002

SLPMiner: An Algorithm for Finding Frequent Sequential Patterns Using Length-Decreasing Support Constraint*

Masakazu Seno and George Karypis
Department of Computer Science and Engineering, Army HPC Research Center
University of Minnesota
4-192 EE/CS Building, 200 Union Street SE, Minneapolis, MN 55455
Fax: (612) 625-0572
{seno, karypis}@cs.umn.edu

Abstract

Over the years, a variety of algorithms for finding frequent sequential patterns in very large sequential databases have been developed. The key feature in most of these algorithms is that they use a constant support constraint to control the inherently exponential complexity of the problem. In general, patterns that contain only a few items will tend to be interesting if they have a high support, whereas long patterns can still be interesting even if their support is relatively small. Ideally, we desire to have an algorithm that finds all the frequent patterns whose support decreases as a function of their length. In this paper we present an algorithm called SLPMiner, that finds all sequential patterns that satisfy a length-decreasing support constraint. SLPMiner combines an efficient database-projection-based approach for sequential pattern discovery with three effective database pruning methods that dramatically reduce the search space. Our experimental evaluation shows that SLPMiner, by effectively exploiting the length-decreasing support constraint, is up to two orders of magnitude faster, and its runtime increases gradually as the average length of the sequences (and the discovered frequent patterns) increases.

1 Introduction

Data mining research during the last eight years has led to the development of a variety of algorithms for finding frequent sequential patterns in very large sequential databases [10, 12, 8]. These patterns can be used to find sequential association rules or extract prevalent patterns that exist in the sequences, and have been effectively used in many different domains and applications.

The key feature in most of these algorithms is that they control the inherently exponential complexity of the problem by finding only the patterns that occur in a sufficiently large fraction of the sequences, called the *support*. A limitation of this paradigm for generating frequent patterns is that it uses a constant support value, irrespective of the length of the discovered patterns. In general, patterns that contain only a few items will tend to be interesting if they have a high support, whereas long patterns can still be interesting even if their support is relatively small. Unfortunately, if constant-support-based frequent pattern discovery algorithms are used to find some of the longer but infrequent

*This work was supported by NSF CCR-9972519, EIA-9986042, ACI-9982274, ACI-0133464, by Army Research Office contract DA/DAAG55-98-1-0441, by the DOE ASCI program, and by Army High Performance Computing Research Center contract number DAAH04-95-C-0008. Access to computing facilities was provided by the Minnesota Supercomputing Institute.

patterns, they will end up generating an exponentially large number of short patterns. Ideally, we desire to have an algorithm that finds all the frequent patterns whose support decreases as a function of their length. Developing such an algorithm is particularly challenging because the downward closure property of the constant support constraint cannot be used to prune short infrequent patterns.

Recently [9], we introduced the problem of finding frequent itemsets whose support satisfies a non-increasing function of their length. An itemset is frequent only if its support is greater than or equal to the minimum support value determined by the length of the itemset. We identified a property that an itemset must have in order to support longer itemsets given a length-decreasing support constraint. This property, called the *smallest valid extension* or *SVE* for short, enabled us to prune large portions of the input database that are irrelevant for finding frequent itemsets that satisfy a length-decreasing support constraint.

In this paper, we extend the problem of finding patterns that satisfy a length-decreasing support constraint to the much more challenging problem of finding sequential patterns. We developed an algorithm called SLPMiner that finds all frequent sequential patterns that satisfy a length-decreasing support constraint. SLPMiner follows the database-projection-based approach for frequent pattern generation, that was shown to lead to efficient algorithms, and serves as a platform to evaluate three new pruning methods based on the SVE property. These pruning methods exploit different aspects of the sequential pattern discovery process and prune either entire sequences, items within certain sequences, or entire projected databases. Our experimental evaluation shows that SLPMiner achieves up to two orders of magnitude of speedup by effectively exploiting the SVE property, and that its runtime increases gradually as the average length of the sequences (and the discovered patterns) increases.

The rest of this paper is organized as follows. Section 2 provides some background information and related research work. Section 3 describes the basic pattern discovery algorithm of SLPMiner and how the length-decreasing support constraint can be exploited to prune the search space of frequent patterns. The experimental evaluation of SLPMiner is shown in Section 4, followed by a conclusion in Section 5.

2 Background

2.1 Sequence Model and Notation

The basic sequence model that we will use was introduced by Srikant *et al* [10] and is defined as follows. Let $I = \{i_1, i_2, \dots, i_n\}$ be the set of all items. An *itemset* is a subset of items. A *sequence* $s = \langle t_1, t_2, \dots, t_l \rangle$ is an ordered list of itemsets, where $t_j \subseteq I$ for $1 \leq j \leq l$. A sequential database D is a set of sequences. The length of a sequence s is defined to be the number of items in s and denoted as $|s|$. Similarly, given an itemset t , let $|t|$ denote the number of items in t . Given a sequential database D , $|D|$ denotes the number of sequences in D .

Sequence $s = \langle t_1, t_2, \dots, t_l \rangle$ is called a *sub-sequence* of sequence $s' = \langle t'_1, t'_2, \dots, t'_m \rangle$ ($l \leq m$) if there exist l integers i_1, i_2, \dots, i_l such that $1 \leq i_1 < i_2 < \dots < i_l \leq m$ and $t_j \subseteq t'_{i_j}$ ($j = 1, 2, \dots, l$). If s is a sub-sequence of s' , then we write $s \subseteq s'$ and say sequence s' *supports* s . The *support* of a sequence s in a sequential database D , denoted as $\sigma_D(s)$, is defined to be $|D_s|/|D|$, where $D_s = \{s_i | s \subseteq s_i \wedge s_i \in D\}$. From the definition, it always holds that $0 \leq \sigma_D(s) \leq 1$. We use the term *sequential pattern* to refer to a sequence when we want to emphasize that the sequence is supported by many sequences in a sequential database.

We assume that the items in I can be arranged in a lexicographic order, and we will use consecutive integers starting from one to represent the items according to that ordering. Finally, we will use the traditional method for writing out sequences, in which each itemset is represented as a list of items ordered according to their lexicographical order and enclosed within matched parentheses $()$, and the sequence of these itemsets is written one-after-the-other enclosed within matched angled parentheses $\langle \rangle$.

To illustrate the above definitions consider the set of items $I = \{1, 2, 3\}$. This set can generate seven possible itemsets and each of them is represented as $(1), (2), (3), (1, 2), (1, 3), (2, 3), (1, 2, 3)$. Let $t_1 = (1, 2)$, $t_2 = (1, 2, 3)$, and

$t_3 = (3)$, be three itemsets of sizes two, three, and one, respectively. Sequence $s = \langle t_1, t_2, t_3 \rangle = \langle (1, 2), (1, 2, 3), (3) \rangle$ has three itemsets and has length $|s| = 2 + 3 + 1 = 6$. Sequence $s' = \langle (1, 3), (1, 2, 3), (1, 2, 3), (2), (2, 3) \rangle$ supports s , or in other words s is a sub-sequence of s' .

2.2 Sequential Pattern Mining with Constant Support

The problem of finding frequent sequential patterns given a constant minimum support constraint [10] is formally defined as follows:

Definition 1 (Sequential Pattern Mining with Constant Support) *Given a sequential database D and a minimum support σ ($0 \leq \sigma \leq 1$), find all sequences each of which is supported by at least $\lceil \sigma|D| \rceil$ sequences in D . ■*

Such sub-sequences are called *frequent sequential patterns*. Note that if every sequence consists of exactly one itemset, the problem of finding frequent sequential patterns degenerates to the problem of finding frequent itemsets in an itemset database [3].

Efficient algorithms for finding frequent itemsets or sequences in very large itemset or sequence databases have been one of the key success stories of data mining research. One of the early computationally efficient algorithm was Apriori [3], which finds frequent itemsets of length l based on previously generated $(l - 1)$ -length frequent itemsets. The GSP [11] algorithm extended the Apriori-like level-wise mining method to find frequent patterns in sequential databases. Recently, a set of database-projection-based methods has been developed that significantly reduce the complexity of finding frequent patterns [1, 6, 5, 8]. The key idea behind these methods is to find the patterns by growing them one item at a time, and simultaneously partitioning (i.e., projecting) the original database into pattern-specific sub-databases (which in general overlap). The process of pattern-growth and database-projection is repeated recursively until all frequent patterns are discovered. Prototypical examples of such algorithms are the tree-projection algorithm [1] that constructs a lexicographical tree and projects a large database into a set of reduced, item-based sub-databases based on the frequent patterns mined so far. The original algorithm was developed for finding non-sequential patterns, but it has been extended by our group to find sequential patterns as well [5]. Another similar algorithm is the FP-growth algorithm [6] that combines projection with the use of the FP-tree data structure to compactly store in memory the itemsets of the original database. The basic ideas in this algorithm were recently used to develop a similar algorithm for finding sequential patterns [8]. Still another approach was introduced in SPADE [13]. SPADE explores the lattice of frequent sequential patterns and uses a specially preprocessed database called the *vertical id-list* database format, which transforms each sequence into a set of (item, time stamp) pairs.

The key feature in these algorithms is that they control the inherently exponential complexity of the problem by using the downward closure property [10]. This property states that in order for a pattern of length l to be frequent, all of its sub-sequences must be frequent as well. As a result, once we find that a sequence of length l is infrequent, we know that any longer sequences that include this particular sequence cannot be frequent, and thus eliminate such sequences from further consideration.

2.3 Finding Patterns with Length-Decreasing Support

A limitation of the above paradigm for generating frequent patterns is that it uses a constant value of support, irrespective of the length of the discovered patterns. In general, patterns that contain only a few items will tend to be interesting if they have a high support, whereas long patterns can still be interesting even if their support is relatively small. Unfortunately, if constant-support-based frequent pattern discovery algorithms are used to find some of the longer but infrequent patterns, they will end up generating an exponentially large number of short patterns. In the context of the frequent itemset mining, maximal frequent itemset discovery algorithms [4, 7, 2, 13] can potentially be used to find some of these longer itemsets, but these algorithms can still generate a very large number of short infrequent itemsets if these itemsets are maximal. As for the frequent sequential pattern mining, even the problem of

finding maximal patterns has not been addressed mainly due to the high complexity of the problem.

Recently, we introduced the idea of length-decreasing support constraint [9] that helps us to find long itemsets with low support as well as short itemsets with high support. A length-decreasing support constraint is given as a function of the itemset length $f(l)$ such that $f(l_a) \geq f(l_b)$ for any l_a, l_b satisfying $l_a < l_b$. The idea of introducing this kind of support constraint is that by using a support function that decreases as the length of the itemset increases, we may be able to find long itemsets that may be of interest without generating an exponentially large number of shorter itemsets. We can naturally extend this constraint to the sequence model by using the length of a sequence instead of the length of an itemset as follows:

Definition 2 (Length-Decreasing Support Constraint) *Given a sequential database D and a function $f(l)$ that satisfies $1 \geq f(l) \geq f(l+1) \geq 0$ for any positive integer l , a sequence s is frequent if and only if $\sigma_D(s) \geq f(|s|)$.* ■

Given a length-decreasing support constraint $f(l)$, we can define the inverse function of $f(l)$ as follows:

Definition 3 (The Inverse Function of Length-Decreasing Support Constraint) *Given a length-decreasing support constraint $f(l)$, its inverse function is defined as $f^{-1}(\sigma) = \min(\{l | f(l) \leq \sigma\})$ for $0 \leq \sigma \leq 1$.* ■

Figure 1 shows a typical length-decreasing support constraint. In this example, the support constraint decreases linearly to the minimum value and then stays the same for sequential patterns of longer length. Formally, the problem of finding this type of patterns is stated as follows:

Definition 4 (Sequential Pattern Mining with Length-Decreasing Support) *Given a sequential database D and a length-decreasing support constraint $f(l)$, find all sequential patterns s such that $\sigma_D(s) \geq f(|s|)$.* ■

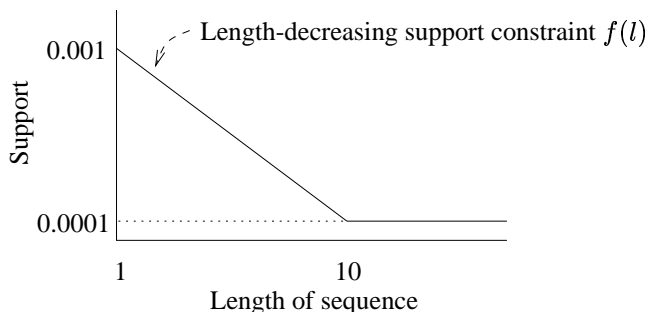


Figure 1. A typical length-decreasing support constraint

A simple way of finding such sequential patterns is to use any of the traditional constant-support frequent sequential pattern discovery algorithms, in which the support is set to $\min_{l \geq 1} f(l)$, and then discard the sequential patterns that do not satisfy the length-decreasing support constraint. This approach, however, does not reduce the number of infrequent sequential patterns being discovered, and as our experiments will show, requires a large amount of time.

Finding the complete set of frequent sequential patterns that satisfy a length-decreasing support constraint is particularly challenging since we cannot rely solely on the downward closure property of the constant support pattern mining. Notice that, under a length-decreasing support constraint, a sequence can be frequent even if its sub-sequences are infrequent since the minimum support value decreases as the length of a sequence increases. We must use $\min_{l \geq 1} f(l)$ as the minimum support value to apply the downward closure property, which will result in finding an exponentially large number of uninteresting infrequent short patterns.

A key property regarding sequences whose support decreases as a function of their length is the following. Given a sequential database D and a particular sequence $s \in D$, if the sequence s is currently infrequent ($\sigma_D(s) < f(|s|)$),

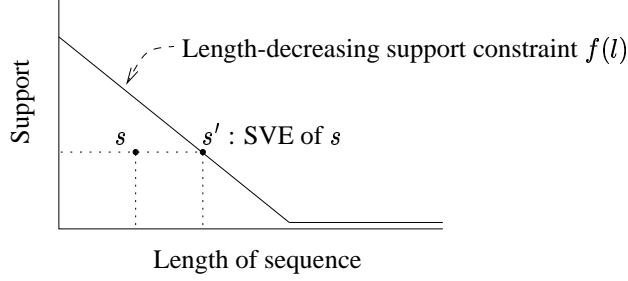


Figure 2. Smallest valid extension (SVE)

then $f^{-1}(\sigma_D(s))$ is the minimum length that a sequence $s' \supset s$ must have before it can potentially become frequent. Figure 2 illustrates this relation graphically. The length of s' is nothing more than the point at which a line parallel to the x -axis at $y = \sigma_D(s)$ intersects the support curve; here, we essentially assume that the best case in which s' exists and it is supported by the same set of sequences as its sub-sequence s . This property is called the *smallest valid extension* property or *SVE* property for short and was initially introduced for the problem of finding itemsets that satisfy a length-decreasing support constraint [9].

3 SLPMiner Algorithm

We developed an algorithm called SLPMiner that finds all the frequent sequential patterns that satisfy a given length-decreasing support constraint. SLPMiner serves as a platform to develop and evaluate various pruning methods for reducing the complexity of finding this type of patterns. Our design goals for SLPMiner were to make it both efficient and at the same time sufficiently generic so that any conclusions drawn from our experiments can carry through other database-projection-based sequential pattern mining algorithms [5, 8].

This section consists of two main parts. First, we explain how SLPMiner finds frequent sequential patterns in the case in which the support of the desired sequential patterns remains constant. Second, we explain various pruning methods that we have developed and incorporated in SLPMiner. These pruning methods substantially reduce the complexity of SLPMiner when it is used to find frequent sequential patterns that satisfy a length-decreasing support constraint.

3.1 Sequential Database-Projection-based Algorithm

SLPMiner finds frequent sequential patterns using a database-projection-based approach that was derived from the sequential version [5] of the tree-projection algorithm of Agarwal *et al* [1] for finding frequent itemsets. Note that the algorithm in [5] shares the same overall structure with the PrefixSpan [8] algorithm that was independently developed at the same time frame.

Key to this algorithm is the use of a tree to both organize the process of sequential pattern discovery and to represent the patterns that have been discovered thus far. Each node in the tree represents a frequent sequential pattern. The relation between the sequential pattern represented at a particular node at level k and that of its parent at level $k - 1$, is that they share the same $k - 1$ prefix. That is, the child's pattern is obtained from that of the parent by adding one item at the end. For example, if a node represents a pattern $\langle(1), (2, 3)\rangle$, its parent node represents $\langle(1), (2)\rangle$. The root node of the tree represents the null sequence with no itemset. From the above definition it is easy to see that given a particular node corresponding to pattern p , all the patterns represented in the nodes of the subtree rooted at that node will have p as a prefix. For this reason, we will refer to this tree as the *prefix tree*.

SLPMiner finds the frequent sequential patterns by growing this tree as follows. It starts from the root node and expands it to create the children nodes that correspond to the frequent items. Then it recursively visits each child node in a depth-first order and expands it into children nodes that represent frequent sequential patterns. SLPMiner

grows each pattern in two different ways, namely, *itemset extension* and *sequence extension*. Itemset extension grows a pattern by adding an item to the last itemset of the pattern, where the added item must be lexicographically larger than any item in the last itemset of the original pattern. For example, $\langle(1), (2)\rangle$ is extended to $\langle(1), (2, 3)\rangle$ by itemset extension, but cannot be extended to $\langle(1), (2, 1)\rangle$ or $\langle(1), (2, 2)\rangle$. Sequence extension grows a pattern by adding an item as a new itemset next to the last itemset of the pattern. For example, $\langle(1), (2)\rangle$ is extended to $\langle(1), (2), (2)\rangle$ by sequence extension.

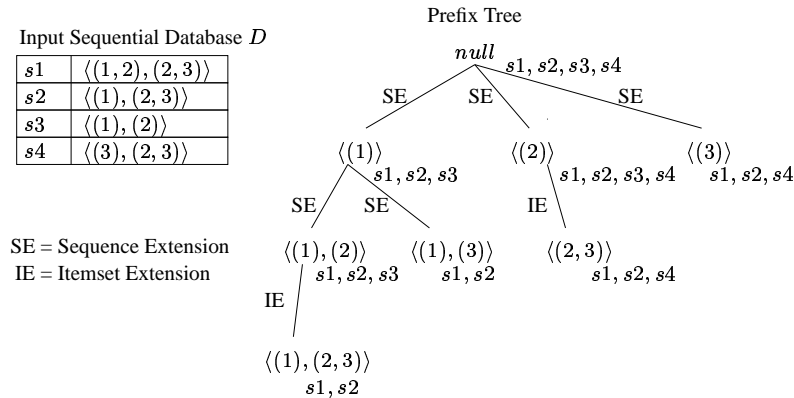


Figure 3. The prefix tree of a sequential database

Figure 3 shows a sequential database D and its prefix tree that contains all the frequent sequential patterns given minimum support 0.5. Since D contains a total of four sequences, a pattern is frequent if and only if at least two sequences in D support the pattern. The root of the tree represents the null sequence. At each node of the tree in the figure, its pattern and its supporting sequences in D are depicted together with symbol SE or IE on each edge representing itemset extension or sequence extension respectively.

The key computational step in SLPMiner is that of counting the frequency of the various itemset and sequence extensions at each node of the tree. In principle, these frequencies can be computed by scanning the original database for each one of the nodes; however, this is not cost-effective, especially when the support for each of those extensions is very small. For this reason, SLPMiner creates a projected database for each node of the tree, and uses this projected database (which is usually much smaller) to determine its frequent extensions. The *projected database* of a sequential pattern p has only those sequences in D that support p . For example, at the node $\langle(2, 3)\rangle$ in Figure 3, its projected database needs to contain only s_1, s_2, s_4 since s_3 does not support this pattern. Furthermore, we can eliminate preceding items in each sequence that will never be used to extend the current pattern. For example, at the node $\langle(2)\rangle$ in Figure 3, we can store sequence $s_1' = \langle(2, 3)\rangle$ instead of s_1 itself in its projected database. Also note that items that do not contribute to a frequent sequence or itemset extension get pruned from all projected databases under that node of the tree. Overall, database projection reduces the amount of sequences that need to be processed at each node and promotes efficient pattern discovery.

3.2 Performance Optimizations for Disk I/O

Expanding each node of the tree, SLPMiner performs the following two steps. First, it calculates the support of each item that can be used for itemset extension and each item that can be used for sequence extension by scanning the projected database D' once. Second, SLPMiner projects D' into a projected database for each frequent extension found in the previous step.

Since we want SLPMiner to be able to run against large input sequential databases, the access to the input database and all projected databases is disk-based. To facilitate this, SLPMiner uses two kinds of buffers: a read-buffer and a write-buffer. The read-buffer is used to load a projected database from disk. If the size of a projected database does

not fit in the read-buffer, SLPMiner reads part of the database from disk several times. The write-buffer is used to temporarily store several projected databases that are generated at a node by scanning the current projected database once using the read-buffer. There are two conflicting requirements concerning how many projected databases we should generate at a time. In order to reduce the number of database scans, we want to generate as many projected databases as possible in one scan. On the other hand, if we keep small buffers for many projected databases simultaneously within the write-buffer, it will reduce the size of the buffer assigned to each projected database, leading to expensive frequent I/O between the write-buffer and disk. In order to balance these two conflicting requirements, SLPMiner calculates the size of each projected database when calculating the support of every item in the current projected database before it actually generates new projected databases. Then, SLPMiner performs a number of database scan, and in each scan, it generates as many projected databases as they can fit in the write-buffer and then writes the entire buffer to the disk. The number of scans depends on the database size and the size of the write buffer. This method also facilitates storing each projected database in a contiguous segment on the disk, allowing us to use fast sequential disk operations which dramatically improve the efficiency of disk I/O.

3.3 Pruning Methods

In this subsection, we introduce three pruning methods that use the SVE property to substantially reduce the size of the projected databases and allow SLPMiner to efficiently find all sequential patterns that satisfy a length-decreasing support constraint.

3.3.1 Sequence Pruning, SP

The first pruning method is used to eliminate certain sequences from the projected databases. Recall that SLPMiner generates a projected database at every node. Let us assume that we have a projected database D' at a node N that represents a sequential pattern p . Each sequence in D' has p as its prefix. If p is infrequent, we know from the SVE property that in order for this pattern to grow to something indeed frequent, it must have a length of at least $f^{-1}(\sigma_D(p))$. Now consider a sequence s that is in the projected database at node N , i.e., $s \in D'$. The largest sequential pattern that s can support is of length $|s| + |p|$. Now if $|s| + |p| < f^{-1}(\sigma_D(p))$, then s is too short to support any frequent patterns that have p as prefix. Consequently, s does not need to be considered any further and can be pruned. We will refer to this pruning method as the *sequence pruning* method or *SP* for short and is formally defined as follows:

Definition 5 (Sequence Pruning) *Given a length-decreasing support constraint $f(l)$ and a projected database D' at a node representing a sequential pattern p , a sequence $s \in D'$ can be pruned from D' if*

$$f(|s| + |p|) > \sigma_D(p).$$

■

SLPMiner checks if a sequence needs to be inserted to a projected database just before inserting it onto the write-buffer. We evaluated the complexity of this method in comparison with the complexity of inserting a sequence to a projected database. There are three parameters we need to know to prune a sequence: $|s|$, $|p|$, and $\sigma_D(p)$. As the length of each sequence is part of the sequence data structure in SLPMiner, it takes a constant time to calculate $|s|$ and $|p|$. As for $\sigma_D(p)$, we know this value when we generated the projected database for the pattern p . Evaluating function f takes a constant time because SLPMiner has a lookup table that contains all possible $(l, f(l))$ pairs. Thus, the complexity of this method is just a constant time per inserting a sequence.

3.3.2 Item Pruning, IP

The second pruning method eliminates certain items from each sequence in each projected database. Let us assume that we have a projected database D' at a node N that represents sequential pattern p and consider an item i in a

sequence $s \in D'$. From the SVE property we know that the item i will contribute to a valid frequent sequential pattern only if

$$|s| + |p| \geq f^{-1}(\sigma_{D'}(i)) \quad (1)$$

where $\sigma_{D'}(i)$ is the support of item i in D' . This is because of the following. The longest sequential pattern that s can participate in is $|s| + |p|$, and we know that, in the subtree rooted at N , sequential patterns that extend p with item i have support at most $\sigma_{D'}(i)$. Now, from the SVE property, such sequential patterns must have length at least $f^{-1}(\sigma_{D'}(i))$ in order to be frequent. As a result, if equation (1) does not hold, item i can be pruned from the sequence s . Once item i is pruned, then $\sigma_{D'}(i)$ and $|s|$ decrease, possibly allowing further pruning. Essentially, this pruning method eliminates some of the infrequent items from the short sequences. We will refer to this method as the *item pruning* method, or *IP* for short and is formally defined as follows:

Definition 6 (Item Pruning) *Given a length-decreasing support constraint $f(l)$ and a projected database D' at a node representing a sequential pattern p , an item i in a sequence $s \in D'$ can be pruned from s if*

$$|s| + |p| < f^{-1}(\sigma_{D'}(i)).$$

■

A simple way to implement this pruning method is as follows: for each projected database D' , repeat scanning D' to collect support values of items and scanning D' again to prune items from each sequence until no more items can be pruned. After that, we can project the database into a projected database for each frequent item in the pruned projected database. This algorithm, however, requires multiple scans of the projected database and hence will be too costly as a pruning method.

Instead, we can scan a projected database once to collect support values and use those support values for pruning items as well as for projecting each sequence. Notice that we are using approximate support values that might be higher than the real values since the support values of some items might decrease during the pruning process. SLPMiner applies IP before generating a projected sequence s' of s and after generating s' just before inserting s' into the write-buffer. By applying IP before projecting the sequences, we can reduce the computation of sequence projection. By applying IP once again for the projected sequence s' , we can exploit the reduction of length $|s| - |s'|$ to further prune items in s' . Pruning items from each sequence is repeated until no more items can be pruned or the sequence becomes short enough to be pruned by SP.

IP can potentially prune a larger portion of the projected database than SP since it always holds that $\sigma_D(p) \geq \sigma_{D'}(i)$ and hence $f^{-1}(\sigma_D(p)) \leq f^{-1}(\sigma_{D'}(i))$. However, the pruning overhead of IP is much larger than that of SP. Given a sequence s , in the worst case, only one item will be pruned during each iteration over the items in s . Since this can be repeated as many as the number of items in the sequence, the worst case complexity for one sequence is $O(n^2)$ where n is the number of items in the sequence. Later in the paper, we will see how this overhead affects the total runtime of SLPMiner through our experimental results.

3.3.3 Structure-based Pruning

Given two sequences s_1, s_2 of the same length k , these two sequences are treated equally under SP and IP. In fact, the two sequences can be quite different from each other. For example, $\langle(1, 2, 3, 4)\rangle$ and $\langle(1), (2), (3), (4)\rangle$ support the same 1-sequence $\langle(1)\rangle, \langle(2)\rangle, \langle(3)\rangle$, and $\langle(4)\rangle$ but never support the same k -sequences for $k \geq 2$. From this observation, we considered ways to split a projected database into smaller equivalent classes. By having smaller databases instead of one large database, we may be able to reduce the depth of a certain path from the root to a leaf node of the tree.

As a structure-based pruning, we developed the min-max pruning method. Let p be a sequential pattern at a particular node, D' be its projected database, and assume that p is infrequent (i.e., $\sigma_D(p) < f(|p|)$). From the SVE property, in order for p to become frequent, we need to grow p by adding at least $f^{-1}(\sigma_D(p)) - |p|$ items. Now, consider the following two values that are defined for each sequence $s \in D'$.

1. $a(s)$ = the smallest number of itemsets in s that need to be used to grow p by $f^{-1}(\sigma_D(p)) - |p|$ items.
2. $b(s)$ = the number of itemsets in s .

These two values define an interval $[a(s), b(s)]$, that we call the *min-max interval* of sequence s . If two sequences $s, s' \in D'$ satisfy $[a(s), b(s)] \cap [a(s'), b(s')] = \emptyset$, then s and s' cannot support any common sequential pattern since their min-max intervals are disjoint. The basic idea of the min-max pruning is motivated by the above observation and its goal is to split the projected database D' into two databases D'_1 and D'_2 such that they contribute to two disjoint sets of frequent sequential patterns.

If there exists D'_1 and D'_2 that satisfy $\cup_{s \in D'_1} [a(s), b(s)] \cap \cup_{s \in D'_2} [a(s), b(s)] = \emptyset$, then D'_1 and D'_2 support distinct sets of frequent sequential patterns. In general, however, this is impossible. Instead, D' will be split into three sets A, B, C of sequences as shown in Figure 4. More precisely, these three sets are defined for some positive integer k as follows.

$$\begin{aligned} A(k) &= \{s | s \in D' \wedge b(s) < k\} \\ B(k) &= \{s | s \in D' \wedge a(s) \geq k\} \\ C(k) &= D' - (A \cup B) \end{aligned}$$

$A(k)$ and $B(k)$ support distinct sets of frequent sequential patterns, whereas $A(k)$ and $C(k)$ as well as $B(k)$ and $C(k)$ support overlapping sets of frequent sequential patterns. From these three sets, we form $D'_1 = A(k) \cup C(k)$ and $D'_2 = B(k) \cup C(k)$. If we mine frequent sequential patterns of length up to $k - 1$ from D'_1 and patterns of length no less than k from D'_2 , we will obtain the same patterns as we would from original D' .

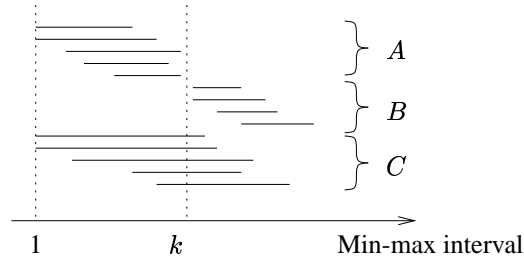


Figure 4. Min-max intervals of a set of sequences

Throughout our experiments, we observed that $|C|$ is usually close to $|D'|$; thus, mining D'_1 and D'_2 separately will cost more than mining the original database D' . We can, however, prune the entire D' if both $|D'_1|$ and $|D'_2|$ are smaller than the $\min_{l \geq 1} f(l)$. Furthermore, we can increase this minimum support by the fact that any sequential patterns that the current pattern p can extend to is of length at most $\max_{s \in D'} (|s|) + |p|$. Now, from the SVE property, we know that if both $|D'_1|$ and $|D'_2|$ are smaller than $f^{-1}(\max_{s \in D'} (|s|) + |p|)|D|$, then we can eliminate the entire D' . Essentially, this means that if we can split a projected database into two subsets each of which is too small to be able to support any frequent sequential pattern, then we can eliminate the entire original projected database. We will refer to this method as the *min-max pruning* method or *MP* for short, and is formally defined as follows:

Definition 7 (Min-Max Pruning) *Given a length-decreasing support constraint $f(l)$ and a projected database D' at a node representing a sequential pattern p , the entire D' can be pruned if there exists a positive integer k such that*

$$|D'_1| = |A(k)| + |C(k)| < f(\max_{s \in D'} (|s|) + |p|)|D|, \text{ and}$$

$$|D'_2| = |B(k)| + |C(k)| < f(\max_{s \in D'}(|s|) + |p|)|D|.$$

■

We apply MP just after a new projected database D' is generated if the entire sequences in D' is still kept in the write-buffer and if $|D'| \leq 1.2f(\max_{s \in D'}(|s|) + |p|)|D|$. The first condition is necessary to avoid costly disk I/O and the second condition is necessary to increase the probability of successfully eliminating the projected database. The algorithm for MP consists of two parts. The first part calculates the distribution of the number of sequences over possible min-max intervals. The second part finds a positive integer k that satisfies the above two equations. The first part requires scanning D' once and finding the min-max interval for each sequence. For each sequence s , SLPMiner determines $a(s)$ as the smallest number of the largest itemsets whose sizes add up to at least $f^{-1}(\sigma_D(p)) - |p|$. The other value $b(s)$ is simply the number of itemsets in s . This part requires $O(m)$ where m is the total number of itemsets in D' . The second part uses an $n \times n$ upper triangular matrix $Q = (q_{ij})$ where $q_{ij} = |\{s | a(s) = i \wedge b(s) = j \wedge s \in D'\}|$ and n is the maximum number of itemsets in a sequence in D' . Matrix Q is generated during the database scan of the first part. Given matrix Q , we have

$$\begin{aligned} |A(k)| + |C(k)| &= \sum_{i=1}^{k-1} \sum_{j=i}^n q_{ij} \\ |B(k)| + |C(k)| &= \sum_{j=k}^n \sum_{i=1}^j q_{ij} \end{aligned}$$

Using the relations

$$\begin{aligned} (|A(k+1)| + |C(k+1)|) - (|A(k)| + |C(k)|) &= \sum_{j=k}^n q_{kj} \\ (|B(k+1)| + |C(k+1)|) - (|B(k)| + |C(k)|) &= - \sum_{i=1}^k q_{ik} \end{aligned}$$

we can calculate $|A(k)| + |C(k)|$ and $|B(k)| + |C(k)|$ incrementally for all k in $O(n^2)$. So the overall complexity of the min-max pruning for one projected database is $O(m + n^2)$. In some cases, this complexity may be much larger than the runtime reduction achieved by eliminating the projected database. However, our experimental results show that the min-max pruning method alone can substantially reduce the total runtime.

4 Experimental Results

We experimentally evaluated the performance of SLPMiner using a variety of datasets generated by the synthetic sequence generator that is provided by the IBM Quest group and was used in evaluating the AprioriAll algorithm [10]. All of our experiments were performed on Linux workstations with AMD Athlon at 1.5GHz and 3GB of main memory. All the reported runtime values are in seconds.

In our experiments, we primarily used two classes of datasets DS1 and DS2, each of which contained 25K sequences. For each of the two classes we generated different problem instances as follows. For DS1, we varied the average number of itemsets in a sequence from 10 to 30 in increments of two, obtaining a total of 11 different datasets, DS1-10, DS1-12, \dots , DS1-30. For DS2, we varied the average number of items in an itemset from 2.5 to 7.0 in increments of 0.5, obtaining a total of 10 different datasets, DS2-2.5, DS2-3.0, \dots , DS2-7.0. For DS1- x , we set the average size of maximal potentially frequent sequences to be $x/2$. For DS2- x , we set the average size of maximal potentially frequent itemsets to be $x/2$. Thus, the dataset contains longer frequent patterns as x increases. The characteristics of these datasets are summarized in Table 1.

| parameter | DS1 | DS2 |
|--|-------------------------|----------------------------|
| $ D $: Number of sequences | 25000 | 25000 |
| $ C $: Average number of itemsets per sequence | $x = 10, 12, \dots, 30$ | 3 to 10 |
| $ T $: Average number of items per itemset | 2.5 | $x = 2.5, 3.0, \dots, 7.0$ |
| N : Number of items | 10000 | 10000 |
| $ S $: Average size of maximal potentially frequent sequences | $x/2$ | 5 |
| $ I $: Average size of maximal potentially frequent itemsets | 1.25 | $x/2$ |

Table 1. Parameters for datasets used in our tests

In addition to the above datasets, we also performed a limited number of experiments with another dataset DS3, for which the parameters were set as follows: $|D| = 25000$, $|C| = 20$, $|T| = 10$, $N = 10000$, $|S| = 10$, and $|I| = 5$. This dataset contains much longer sequences than DS1 and DS2 and was used to evaluate the overheads associated with the various pruning methods.

In all of our experiments, we used a minimum support constraint that decreases linearly with the length of the frequent sequential pattern. In particular, the initial value of support was set to 0.001 and it was decreased linearly down to 0.0001 for sequences of up to length $\lfloor |C||T|/2 \rfloor$. For the rest of sequences, the support was kept fixed at 0.0001. Figure 5 shows the shape of the support curve for DS1-20 for which $\lfloor |C||T|/2 \rfloor = \lfloor 20 \times 2.5/2 \rfloor = \lfloor 50/2 \rfloor = 25$.

We also ran SPADE [12] to compare runtime values with SLPMiner. When running SPADE, we used the depth first search option, which leads to better performance than the breadth first search option on our datasets. We set the minimum support value to be $\min_{l \geq 1} f(l)$.

For SLPMiner, we set the size of the read-buffer to 10MB and the write-buffer to 300MB. Similarly, we set the available memory size to 310MB for SPADE.

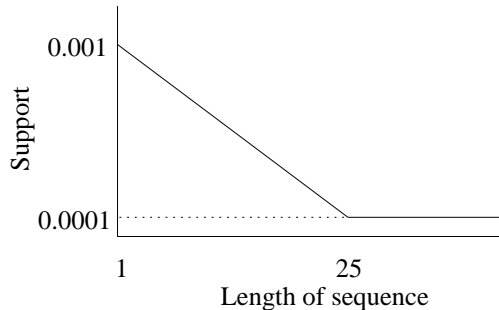


Figure 5. Support curve for DS1-20

4.1 Results

Tables 2 and 3 show the experimental results that we obtained for the DS1 and DS2 datasets respectively. Each row of the tables shows the results obtained for a different DS1- x or DS2- x dataset, specified on the first column. The column labeled “SPADE” shows the amount of time taken by SPADE. The column labeled “None” shows the amount of time taken by SLPMiner using a constant support constraint that corresponds to the smallest support of the support curve, that is 0.0001 for all datasets. The other columns show the amount of time required by SLPMiner that uses the length-decreasing support constraint and a total of five different combinations of pruning methods. For example, the column label “SP” corresponds to the pruning scheme that uses only sequence pruning, whereas the column labeled “SP+IP+MP” corresponds to the scheme that uses all the three pruning methods. Note that values with a “-” correspond to experiments that were aborted because they were taking too long time.

| Dataset | SPADE | SLPMiner | | | | | |
|---------|----------|----------|---------|---------|---------|---------|----------|
| | | None | SP | IP | MP | SP+IP | SP+IP+MP |
| DS1-10 | 10.562 | 20.219 | 11.514 | 11.570 | 12.641 | 12.006 | 11.839 |
| DS1-12 | 18.245 | 41.420 | 15.316 | 15.430 | 17.804 | 15.358 | 15.935 |
| DS1-14 | 46.216 | 98.359 | 21.290 | 21.583 | 24.453 | 21.429 | 21.297 |
| DS1-16 | 87.289 | 208.187 | 27.342 | 26.635 | 31.230 | 26.186 | 27.383 |
| DS1-18 | 273.325 | 592.886 | 39.228 | 39.030 | 43.490 | 38.790 | 40.172 |
| DS1-20 | 594.777 | 1438.932 | 46.147 | 48.440 | 54.727 | 47.864 | 47.723 |
| DS1-22 | 4702.697 | 8942.943 | 63.351 | 65.123 | 74.905 | 65.232 | 65.907 |
| DS1-24 | - | - | 82.756 | 85.622 | 94.640 | 82.377 | 83.148 |
| DS1-26 | - | - | 106.986 | 112.180 | 126.647 | 111.699 | 106.567 |
| DS1-28 | - | - | 139.369 | 142.760 | 162.062 | 137.955 | 138.411 |
| DS1-30 | - | - | 180.715 | 189.029 | 212.848 | 185.601 | 184.105 |

Table 2. Comparison of pruning methods using DS1

| Dataset | SPADE | SLPMiner | | | | | |
|---------|----------|-----------|---------|---------|---------|---------|----------|
| | | None | SP | IP | MP | SP+IP | SP+IP+MP |
| DS2-2.5 | 10.562 | 20.219 | 11.514 | 11.570 | 12.641 | 12.006 | 11.839 |
| DS2-3.0 | 21.159 | 45.887 | 16.627 | 16.940 | 18.719 | 15.871 | 15.902 |
| DS2-3.5 | 117.486 | 279.617 | 31.851 | 35.319 | 43.267 | 31.445 | 31.696 |
| DS2-4.0 | 333.786 | 899.025 | 32.783 | 32.488 | 39.805 | 31.940 | 32.107 |
| DS2-4.5 | 731.402 | 1784.572 | 35.871 | 37.955 | 43.138 | 38.030 | 36.539 |
| DS2-5.0 | 6460.641 | 17106.370 | 57.677 | 61.654 | 77.835 | 59.115 | 59.096 |
| DS2-5.5 | - | - | 59.500 | 62.617 | 73.759 | 61.187 | 61.798 |
| DS2-6.0 | - | - | 77.752 | 78.684 | 96.951 | 77.925 | 75.186 |
| DS2-6.5 | - | - | 98.061 | 105.475 | 144.387 | 101.213 | 102.184 |
| DS2-7.0 | - | - | 116.986 | 119.907 | 136.513 | 113.443 | 117.602 |

Table 3. Comparison of pruning methods using DS2

| | SP | IP | SP+IP | SP+IP+MP |
|------------------------------|-----------|-----------|-----------|-----------|
| Runtime | 15939.386 | 16019.344 | 15103.936 | 15205.960 |
| Projected Database Size (GB) | 65.990 | 47.501 | 43.206 | 41.358 |

Table 4. Comparison of pruning methods using DS3

A number of interesting observations can be made from the results in these tables. First, even though SLPMiner without any pruning method is slower than SPADE, the ratio of runtime values is stable ranging from 1.9 to 2.7 with average 2.3. This shows that the performance of SLPMiner is comparable to SPADE and a reasonably good platform for evaluating our pruning methods. Second, either one of pruning methods performs better than SLPMiner without any pruning method. In particular, SP, IP, SP+IP, and SP+IP+MP have almost the same speedup. For DS1, the speedup by SP is about 1.76 times faster for DS1-10, 7.61 times faster for DS1-16, and 141.16 times faster for DS1-22. Similar trends can be observed for DS2, in which the performance of SLPMiner with SP is 1.76 times faster for DS2-2.5, 8.78 times faster for DS2-3.5, and 296.59 times faster for DS2-5.0. Third, comparing the different pruning methods in isolation, we can see that SP leads to the largest runtime reduction, IP leads to the second largest runtime reduction, and MP achieves the smallest reduction. The problem with MP is the overhead of splitting a database into two subsets. Even so, it seems surprising to gain such a great speedup by MP alone. This shows a large part of the runtime of SLPMiner without any pruning method is accounted for by many small projected databases that never contribute to any frequent patterns. As for SP and IP, SP is slightly better than IP because IP and SP prune almost the same amount of projected databases for those datasets but IP has much larger overhead than SP. Fourth, the runtime with three pruning methods increases gradually as the average length of the sequences (and the discovered patterns) increases, whereas the runtime of SLPMiner without any pruning increases exponentially.

Finally, Table 4 shows the runtime and projected database size for the DS3 dataset. We tested SP, IP, SP+IP, SP+IP+MP for DS3 since they were the best when applied to DS1 and DS2 datasets. Even though the projected database size of IP is 1.5 times smaller than that of SP, SP and IP achieve almost the same runtime again because of the large overhead of IP. These two methods, however, can achieve the best runtime when combined as SP+IP because IP does not have to prune the part of projected databases for which SP can prune. Since DS3 contains much longer sequences than DS1 and DS2 datasets, there are more opportunities for IP to prune where SP does not work.

5 Conclusion

In this paper we presented an algorithm SLPMiner that can efficiently find all frequent sequential patterns that satisfy a length-decreasing support constraint. The key insight that enabled us to achieve high performance was the smallest valid extension property of the length-decreasing support constraint. This allowed us to develop effective database pruning methods that improved the performance of SLPMiner by up to two orders of magnitude.

The pruning methods are not specific to SLPMiner but almost all of them can be incorporated into other algorithms for sequential pattern discovery. For example, it is straight-forward to implement all three pruning methods in PrefixSpan [8] with disk-based projection. PrefixSpan with pseudo-projection can use the sequence pruning method. Even SPADE [12], which has no explicit sequence representation during pattern mining, can use the sequence pruning method by adding the length of a sequence to each record in the vertical database representation.

References

- [1] R. Agarwal, C. Aggarwal, V. Prasad, and V. Crestana. A tree projection algorithm for generation of large itemsets for association rules. *IBM Research Report*, RC21341, November 1998.
- [2] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *Knowledge Discovery and Data Mining*, pages 108–118, 2000.
- [3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Int'l Conference on Very Large Databases*, Santiago, Chile, September 1994.
- [4] R. J. Bayardo. Efficiently mining long patterns from databases. In *ACM SIGMOD Conf. Management of Data*, 1997.
- [5] V. Guralnik, N. Garg, and G. Karypis. Parallel tree projection algorithm for sequence mining. In *European Conference on Parallel Processing*, pages 310–320, 2001.
- [6] 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.

- [7] D.-I. Lin and Z. M. Kedem. Pincer search: A new algorithm for discovering the maximum frequent set. In *Extending Database Technology*, pages 105–119, 1998.
- [8] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *ICDE*, pages 215–224, 2001.
- [9] M. Seno and G. Karypis. Lpminer: An algorithm for finding frequent itemsets using length-decreasing support constraint. In *1st IEEE Conference on Data Mining*, 2001.
- [10] R. Srikant and R. Agrawal. Mining sequential patterns. In *11th Int. Conf. Data Engineering*, 1995.
- [11] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *5th Int. Conf. Extending Database Technology*, 1996.
- [12] M. J. Zaki. Fast mining of sequential patterns in very large databases. Technical Report TR668, 1997.
- [13] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. Technical Report 01-1, RPI, 2001.