OSSM: A Segmentation Approach to Optimize Frequency Counting

Carson Kai-Sang Leung
The University of British Columbia
2366 Main Mall
Vancouver, BC
Canada V6T 1Z4
kleung@cs.ubc.ca

Raymond T. Ng
The University of British Columbia
2366 Main Mall
Vancouver, BC
Canada V6T 1Z4
rng@cs.ubc.ca

Heikki Mannila
University of Helsinki and
Helsinki University of Technology
PO Box 26, FIN-00014 Helsinki
Finland
Heikki.Mannila@cs.helsinki.fi

Abstract

Computing the frequency of a pattern is one of the key operations in data mining algorithms. We describe a simple yet powerful way of speeding up any form of frequency counting satisfying the monotonicity condition. Our method, the optimized segment support map (OSSM), *is a light-weight structure which partitions the collection of transactions into m segments, so as to reduce the number of candidate patterns that require frequency counting. We study the following problems: (1) What is the optimal number of segments to be used; and (2) Given a user-determined m, what is the best segmentation/composition of the m segments? For Problem 1, we provide a thorough analysis and a theorem establishing the minimum value of m for which there is no accuracy lost in using the OSSM. For Problem 2, we develop various algorithms and heuristics, which efficiently generate OSSMs that are compact and effective, to help facilitate segmentation.*

Keywords: *Data mining, frequent patterns, support counting, data structure, performance analysis*

1. Introduction and motivation

Computing the frequency (or support) of a pattern is one of the key operations in data mining algorithms. For the algorithms to discover frequently occurring patterns, they have to investigate specific patterns to find cardinalities of subgroups or significance of deviations, etc. Typically, the patterns, whose frequencies are needed, are conjunctions of atomic patterns. A prime example is given by the frequent set concept underlying association rules [2, 3]. Moreover, the patterns defined for correlation [6, 7], causality [18], sequential patterns [4], episodes [13], constrained frequent sets [11, 14, 19], long patterns [1, 5], closed sets [16], and many other important data mining tasks have the same basic form. In all these cases, we have instances of the following abstract problem. Given a collection $\mathcal{P}$ of atomic patterns or conditions, compute for collections $S \subseteq \mathcal{P}$ the support $sup(S)$ of S. The monotonicity condition $sup(S) \leq sup(\{a\})$ holds for all $a \in S$, since the frequency or support of the collection S is defined to be the number of observations in the data for which all atomic patterns in S are true.

In this paper, we describe a simple yet powerful way of speeding up any form of frequency counting satisfying the monotonicity condition. Our method, the **optimized segment support map (OSSM)**, is based on a simple observation about data: Real life data sets are not random. More concretely, the frequencies of patterns will be different in different parts of the data set. Computing the frequencies separately in different parts of the data set makes it possible to obtain better/tighter upper bounds for the frequencies of the collections of patterns and thus enables one to prune much more effectively. The results will be useful to the mining of any of the above classes of patterns.

The frequencies of patterns are computed typically in a collection of "transactions", i.e., $T = \{t_1, \ldots, t_n\}$.[1] Most algorithms mining the types of patterns mentioned above do so by generating candidate patterns $S_1, \ldots, S_w$, and checking their frequencies against T.[2] The OSSM is a light-weight, easy-to-compute structure, which partitions T into m segments, i.e., $T = T_1 \cup \ldots \cup T_m$ and $T_i \cap T_j = \emptyset$, with *the goal of reducing the number of candidate patterns (i.e., w) for which frequency counting is required*. We make the following specific contributions:

- In Section 3, we present the OSSM and show how it provides a light-weight mechanism to reduce the number of candidate patterns.
- In Section 4, we consider the **segment minimization problem**. Given an OSSM $\mathcal{M}_m$ of m segments $T_1, \ldots, T_m$, the OSSM provides for any set S of items an upper bound for the support of S, denoted by $\widehat{sup}(S, \mathcal{M}_m)$. The segment minimization problem is to find the smallest value of m such that the upper bound, in fact, becomes the actual support of S. In other words, there is no loss of accuracy using the OSSM. We show that that the minimum value of m, denoted as m_{min}, can be quite high in the general case.

[1]Transactions may come in different forms. In the case of association rules, a transaction is a set of items [2]. In the case of episodes, a transaction corresponds to a sequence of events in a sliding time window [13].

[2]There are algorithms that do not rely on candidate generation (e.g., FP-growth [8], CHARM [21], and GenMax [20]).

- Given that the OSSM created with m_{min} segments will consume too much space, in Section 5 we consider the **constrained segmentation problem**. Given an input value m_{user} (which is smaller than m_{min}), the constrained segmentation problem seeks to find the m_{user} segments that minimize the loss of accuracy. In other words, with the number of segments fixed to m_{user}, we seek to find the best way to partition T into $T_1, \ldots, T_{m_{user}}$. Finding the optimal solution turns out to be computationally hard. We, thus, focus on developing heuristic segmentation algorithms. We present three such algorithms and analyze their complexities. To further optimize the segmentation process, we propose to run some of these algorithms in a hybrid fashion, and propose a heuristic called the *bubble list*.
- Given the heuristic nature of our algorithms, we present experimental results in Section 6 evaluating the effectiveness of the segmentation. We show that all our segmentation algorithms bring about significant savings in frequency counting. For example, for an OSSM that occupies only 0.3 megabytes in total size, it can bring about a speedup of 50 times. Furthermore, with the help of our proposed heuristics, this OSSM can be constructed in less than 10 seconds of total time, for 5 million transactions.

An outline of the paper is as follows. The next section discusses the related work. Section 3 introduces the OSSM and the way it reduces the number of candidate patterns. Section 4 analyzes the segment minimization problem, and Section 5 studies the constrained segmentation problem. Section 6 shows experimental results. Section 7 discusses the additional benefit of using the OSSM in conjunction with efficiency/performance improving algorithms such as DHP [15], Partition [17], and DepthProject [1]. Section 8 presents conclusions.

2. Related work

The most relevant piece of related works is our case study [10] focusing on the use of a SSM structure to facilitate the Carma algorithm [9] for online mining. With such a structure, the collection of transactions is divided *arbitrarily* into several partitions. Such a case study is different from our current paper as follows. First, in the current paper, we discuss how the OSSM is applicable to *general* pattern discovery algorithms (e.g., Apriori, and many of its variants and extensions). Second, the collection of transactions is divided into segments in the OSSM *according to some optimization criteria* (e.g., to preserve accuracy when dealing with the segment minimization problem; to minimize the loss in accuracy when dealing with the constrained segmentation problem). Third, the key technical focus of the current paper is on the segment minimization problem and the constrained segmentation problem, which have not been studied previously.

The next group of related works are partitioning-based data mining algorithms (e.g., the DHP algorithm [15], the Partition algorithm [17]). On the surface, the hashing-based partitioning done by these algorithms looks similar to the OSSM. However, there are several key differences. For example, the DHP algorithm uses a hash table to partition the itemsets of the same cardinality (e.g., 2) into buckets; and the hash table is created dynamically for each run of the algorithm. In contrast, the OSSM partitions transactions, not itemsets; and the OSSM is intended to be a static structure. The Partition algorithm does not set up any structure like the OSSM to reduce the number of candidate patterns. Moreover, the segment minimization problem and the constrained segmentation problem are not considered in the DHP algorithm or the Partition algorithm. (As a preview, we will discuss in Section 7 how the OSSM can be used in *conjunction* with these two partitioning-based data mining algorithms, and will show the additional benefit brought by the OSSM to these two algorithms.)

The third group of related works are the FP-growth algorithm [8] and its variants. The FP-growth algorithm is an example of a framework not requiring candidate generation. The algorithm constructs a prefix-tree structure called the FP-tree to mine frequent patterns in a depth-first fashion. Contrarily, the OSSM framework proposed in the current paper — like most studies on frequent pattern computation — relies on candidate generation. The two frameworks are different in at least the following key aspects. First, the FP-tree structure is query-dependent; the tree is constructed based on a user-defined support threshold of the query. In contrast, the OSSM structure is "static", i.e., operating with any support threshold. Second, the FP-tree structure is main-memory based. If the structure is too large to fit into main memory, recursive projections and partitioning are required, causing additional overhead. On the other hand, once the FP-tree resides in memory, it does not take advantage of additional memory space. In contrast, given a small amount of memory space, an OSSM can still be constructed (using fewer segments) and can bring a significant speedup. Given a large amount of memory space, the OSSM can take advantage of the additional space by using more segments; this further reduces the number of candidates for which frequency counting is required (i.e., leads to more effective pruning). In sum, the OSSM is designed to work with *any* amount of memory space. Its size is independent of the size of the itemset lattice. In fact, the main focus of the current paper is to explore how to construct an OSSM, given a pre-determined amount of space.

3. The optimized segment support map and its utility

Let the collection of transactions T be partitioned into m segments, not necessarily of equal size. In later sections, we will discuss how to determine the composition of the m segments. An OSSM is a structure consisting of $sup_i(\{a\})$ for all *singleton* itemsets $\{a\}$, where $sup_i(\{a\})$ denotes the support of $\{a\}$ in the i-th segment T_i, for $1 \leq i \leq m$. The support of $\{a\}$ is then $\sum_{i=1}^{m} sup_i(\{a\})$. While the OSSM $\mathcal{M}_m$ only contains the segment supports of singleton itemsets (for the m segments $T_1, \ldots, T_m$), it can be used to give an upper bound on the support of an arbitrary itemset S:

$$\widehat{sup}(S, \mathcal{M}_m) = \sum_{i=1}^{m} min\{sup_i(\{a\}) \mid a \in S\} \quad (1)$$

Example 1 Suppose there are 4 segments in the OSSM $\mathcal{M}_4$, and the (actual) segment supports for items a, b and c are as shown:

	T_1	T_2	T_3	T_4	T
$\{a\}$	20	10	40	40	110
$\{b\}$	40	40	40	10	130
$\{c\}$	40	20	20	20	100

By equation (1), $\widehat{sup}(\{a,b\}, \mathcal{M}_4)$ is $min\{20,40\} + min\{10,40\} + min\{40,40\} + min\{40,10\}$, for a total of 80. Similarly, by equation (1), the support of $\{a,b,c\}$ is bounded from above by 60. On the other hand, if we do not use the OSSM (i.e., just by the last column of the above table), then the upper bound on the support for $\{a,b\}$ is $min\{110,130\} = 110$, while that for $\{a,b,c\}$ is $min\{110,130,100\} = 100$. ■

The above example shows how the OSSM can provide valuable filtering by reducing the number of candidate sets that need to be counted for their frequencies (e.g., when the support threshold is less than 100). For many of the frequent pattern algorithms, such as frequent-set discovery methods or episode discovery methods, one of the performance bottlenecks is the number of candidates that have to be considered: Even in cases when the true number of frequent patterns is small, there can be a huge number of candidate patterns. Another important thing to note is that for many algorithms (e.g., those based on hashing), skewed data can be problematic. In contrast, the more skewed the data, the more effective the OSSM is.

Clearly, the upper bound $\widehat{sup}(S, \mathcal{M}_m)$ provided by the OSSM $\mathcal{M}_m$ can be made tighter by increasing the number of segments m.[3] The amount of storage space required is then increased linearly. In the hypothetical extreme case, when the number of segments m equals the number of transactions n, the upper bound becomes the *actual* support count of S.

We also note that knowledge discovery is typically an iterative process: One first computes certain patterns, investigates them, and then re-computes using perhaps different thresholds. In this context, an advantage of the OSSM is that it a fixed structure that can be computed *once* at "compile-time" (pre-processing), and can be used regardless of how the support threshold is changed dynamically during "exploration-time" (query execution). In other words, the OSSM is query-independent. This is different from such mining algorithms as DHP [15] or FP-growth [8], which are query-dependent; they construct summary structures (e.g., hash table or FP-tree) based on a given support threshold of the query (cf.: Section 2).

Finally, we note that there is no searching involved when the OSSM is used. Once the singleton itemsets are enumerated based on some canonical ordering, the itemsets themselves (i.e., the first column of the above table) need not be stored, and direct addressing into the OSSM makes the use of equation (1) very efficient.

[3]An alternative way to tighten $\widehat{sup}(S, \mathcal{M}_m)$ is to generalize the OSSM by storing not only the actual segment supports of singleton patterns or itemsets, but also those of itemsets of higher cardinalities (i.e., itemsets of sizes greater than one).

Symbols	Meanings
T	the collection of reference transactions
m	the number of segments
m_{min}	the minimum m without loss of accuracy
m_{user}	the user-specified number of segments
$T_1, \ldots, T_m$	the m segments of transactions
S	a set of items
$a, a_1, \ldots, a_i, b, c$	individual items
k	the total number of individual domain items
$\mathcal{M}_m$	an OSSM consisting of m segments
$sup(S)$	the actual support of S with respect to T
$\widehat{sup}(S, \mathcal{M}_m)$	upper bound on $sup(S)$ based on $\mathcal{M}_m$
p	the number of pages occupied by T

Figure 1. Definitions of the main symbols

4. The segment minimization problem

While the previous section highlights the potential benefit of the OSSM, there are two main questions that need to be addressed:

1. *What is the best number of segments (i.e., the value of m)?*
2. *What is the best composition of the m segments?*

The primary focus of this section is the first question. In particular, we consider the *segment minimization problem*, as stated below.

Definition 1 Given a collection of transactions T, the **segment minimization problem** is to determine the minimum value m_{min} for the number of segments in the OSSM $\mathcal{M}_{m_{min}}$, such that $\widehat{sup}(S, \mathcal{M}_{m_{min}}) = sup(S)$ for all itemsets S, i.e., the upper bound on the support for any S is exactly its actual support. ■

To make it easier to understand our analysis, we begin with the simple (hypothetical) situation where there are only 2 individual items. Then we present the key theorem giving the m_{min} value for the general case with $k \geq 2$ items. Finally, we generalize the analysis to cover a more practical and relaxed notion of accuracy. For reference, Figure 1 gives a summary of the main symbols used in this paper.

4.1. Lessons learned from the situation with only two items

For ease of understanding, we begin our analysis for the hypothetical situation when there are only two items a and b.

Example 2 Suppose there is a collection of transactions as shown:

Transactions in T	Contents
t_1	$\{a\}$
t_2	$\{a,b\}$
t_3	$\{a\}$
t_4	$\{a\}$
t_5	$\{b\}$
t_6	$\{b\}$

Let us consider the following OSSM $\mathcal{M}_6$ containing 6 segments, each of which consisting of a single transaction.

	T_1	T_2	T_3	T_4	T_5	T_6
$\{a\}$	1	1	1	1	0	0
$\{b\}$	0	1	0	0	1	1

While it is impractical to consider one transaction per segment, this example does illustrate an important observation. Consider the following OSSM $\mathcal{M}_2$, which contains 2 segments. Specifically, the first segment T'_1 consists of all transactions containing a (i.e., t_1, t_2, t_3 and t_4), and the second segment T'_2 consisting of all transactions containing b but not a (i.e., t_5 and t_6).

	T'_1	T'_2
$\{a\}$	4	0
$\{b\}$	1	2

Now with this OSSM $\mathcal{M}_2$, it is easy to verify that the upper bound $\widehat{sup}(\{a,b\}, \mathcal{M}_2)$ is given by $min\{4,1\} + min\{0,2\} = 1$, which is the *exact* support of $\{a,b\}$. Furthermore, suppose that the segmentation is done slightly differently — with t_4 moved from segment T'_1 to T'_2. The resulting upper bound is then given by $min\{3,1\} + min\{1,2\} = 2$, which is no longer the *exact* support of $\{a,b\}$. ■

The following analysis shows that what we have seen from the above example is not a coincidence.

The definition of segment T'_1 in Example 2 necessitates that $sup(\{a\}) \geq sup(\{b\})$ in the segment. From now on, we use the notation $(a \succeq b)$ to describe this situation, and we refer to this description as the *configuration* of the segment. Similarly, we denote the configuration of segment T'_2 as $(b \succeq a)$.

The following lemma shows that when we merge two segments T_i and T_j that are of the same configuration, we preserve the configuration and do not change the upper bound due to the two segments.

Lemma 1 Let T_i and T_j be two segments of the same configuration from a general collection of transactions. If we merge T_i and T_j into one combined segment T', then the merged segment is the same configuration, and $\widehat{sup}(\{a,b\}, T_i) + \widehat{sup}(\{a,b\}, T_j) = \widehat{sup}(\{a,b\}, T')$. ■

To understand this lemma, let T_i and T_j be of the configuration $(a \succeq b)$. Then we have:

	T_i
$\{a\}$	v_1
$\{b\}$	v_2

	T_j
$\{a\}$	v_3
$\{b\}$	v_4

with $v_1 \geq v_2$, and $v_3 \geq v_4$. Now with T_i and T_j merged, we have:

	$T' = T_i \cup T_j$
$\{a\}$	$v_1 + v_3$
$\{b\}$	$v_2 + v_4$

It is easy to verify that $\widehat{sup}(\{a,b\}, T_i) + \widehat{sup}(\{a,b\}, T_j) = \widehat{sup}(\{a,b\}, T')$. The case for two segments of the configuration $(b \succeq a)$ is similar.

4.2. Generalization to the situation with k items

In general, for k items $a_1, \ldots, a_k$, the configuration of a segment is of the form $(a_{i_1} \succeq \ldots \succeq a_{i_k})$, where $(a_{i_1}, \ldots, a_{i_k})$ is a permutation of the k items. This descriptor indicates that in this segment, it is the case that $sup(\{a_{i_1}\}) \geq \ldots \geq sup(\{a_{i_k}\})$.[4] Clearly then, there are $k!$ possible configurations of segments.

A close examination reveals that the number of possible configurations can be reduced. Given a collection of transactions of k items, it produces at most $2^k - 1$ distinct non-empty itemsets. Among these itemsets, k of them have the same configuration. Hence, the number of possible configurations of segments can be reduced to $2^k - k$.

Theorem 1 Let T be a general collection of n transactions for k items. If we allow T to be rearranged, then the minimum number of segments m_{min} required in the general case for the upper bound on $sup(S)$ to be exact for all S, is the number of segments with distinct configurations, i.e., $m_{min} \leq min\{n, 2^k - k\}$. ■

To understand the above theorem, suppose there are initially more than $2^k - k$ segments. Then there must be at least two segments with the same configuration, say $(a_{i_1} \succeq \ldots \succeq a_{i_k})$. Then the reasoning articulated in Lemma 1 applies. Thus, even if there are initially more than $2^k - k$ segments, repeated merging would reduce the number of segments to be no more than $2^k - k$.

To go one step further, let us consider merging two segments T_1, T_2 with different configurations. While the general case can be proved by induction, let us focus on the following simple case to understand the situation — the configurations being:

$$(a_{i_1} \succeq \ldots \succeq a_{i_j} \succeq a_{i_{j+1}} \succeq \ldots \succeq a_{i_k}) \text{ for } T_1,$$
$$(a_{i_1} \succeq \ldots \succeq a_{i_{j+1}} \succeq a_{i_j} \succeq \ldots \succeq a_{i_k}) \text{ for } T_2,$$

i.e., two successive items swapped in the permutation. The following shows the part of the OSSM corresponding to T_1 and T_2:

	T_1	T_2
$\{a_{i_j}\}$	v_1	v_3
$\{a_{i_{j+1}}\}$	v_2	v_4

for some non-negative integers $v_1, \ldots, v_4$. Because $a_{i_j} \succeq a_{i_{j+1}}$ in T_1, it is the case that $v_1 \geq v_2$. However, in T_2, it is the other way, namely $v_4 \geq v_3$. If we merge these two segments, we get:

	$T' = T_1 \cup T_2$
$\{a_{i_j}\}$	$v_1 + v_3$
$\{a_{i_{j+1}}\}$	$v_2 + v_4$

Now consider estimating the support $sup(\{a_{i_j}, a_{i_{j+1}}\})$. It is always the case that $min\{(v_1 + v_3), (v_2 + v_4)\} \geq (v_2 + v_3)$. The inequality is strict, unless $v_1 = v_2$ and $v_3 = v_4$. This shows that *merging segments with differing configurations can cause the upper bound on $sup(S)$ for some set S to be inexact*. Hence, if the transactions themselves give rise to all $2^k - k$ possible configurations, m_{min} is exactly $2^k - k$.

[4]To be more precise, if there are ties (i.e., $sup(\{a_{i_j}\}) = sup(\{a_{i_{j+1}}\})$), then we adopt the following convention. Without loss of generality, we assume that there is a canonical enumeration of the individual items. Ties are broken by following the canonical enumeration, i.e., $i_j < i_{j+1}$.

4.3. A relaxed notion of accuracy: The page version

The above analysis addresses the segment minimization problem formally. However, from a utility standpoint, the results obtained indicate clearly that segment minimization is hard, i.e., requiring in the general case $2^k - k$ segments to ensure that the upper bound on $sup(S)$ is exact for all S. The number $2^k - k$ is simply too huge to be practical.

A natural question to ask is: If we were to relax the notion of accuracy in segment minimization, would we be able to reduce the number of required segments to a much more manageable number? Specifically, as analyzed so far, the segment minimization admits no loss of accuracy, i.e., the upper bound from the OSSM on $sup(S)$ exactly matches the actual support for all S. As transactions are stored in pages, one natural form of relaxation of the segment minimization problem is to consider the page version of the problem. Recall that in the earlier analysis, we have made the implicit assumption that *initially* we know the frequency of every item in every transaction in the segment. The page version of the segment minimization problem begins with a higher granularity level. Initially, we have the aggregate frequency of every item *per page*. Suppose that the collection of transactions T is physically organized into p pages, denoted by $T'_1, \ldots, T'_p$. Then the *page version of the segment minimization problem* can be described as follows.

Definition 2 Given a collection of transactions T organized in p pages $T'_1, \ldots, T'_p$, the **page version of the segment minimization problem** is to determine the minimum value m_{min} for the number of segments in the OSSM $\mathcal{M}_{m_{min}}$, such that $\widehat{sup}(S, \mathcal{M}_{m_{min}}) = \widehat{sup}(S, \mathcal{M}'_p)$ for all itemsets S, i.e. the upper bound suffers no loss of accuracy compared to the initial p segments in $\mathcal{M}'_p = \langle T'_1, \ldots, T'_p \rangle$. ■

Corollary 1 Let T be a collection of transactions for k items in p pages $T'_1, \ldots, T'_p$. If we allow these pages to be rearranged, then the minimum number of segments m_{min} required in the general case for $\widehat{sup}(S, \mathcal{M}_{m_{min}}) = \widehat{sup}(S, \mathcal{M}'_p)$ for all S, is the number of segments with distinct configurations, i.e., $m_{min} \leq min\{p, 2^k - k\}$. ■

For academic interest, this corollary of Theorem 1 is a positive result because it indicates that all the previous analysis carries over to the page version of segment minimization. However, from a practical perspective, this corollary is negative saying that even if we were to relax the notion of accuracy from a per-transaction basis to a per-page basis, we will still end up with a huge number of segments for segment minimization. This motivates us to explore the constrained segmentation problem.

5. The constrained segmentation problem

Recall that concerning the creation of an OSSM, there are two issues: (i) the number of segments, and (ii) the composition of the segments. The segment minimization problem we have analyzed so far begins by focusing on having the minimum number of segments. In this section, we look at these issues from a different angle. Here we study the *constrained segmentation problem*, as stated below.

Definition 3 Given a collection of transactions in p pages and a fixed number of segments $m_{user} \ll m_{min}$ (and $m_{user} \ll p$) to be formed, the **constrained segmentation problem** is to determine the best composition of the m_{user} segments that minimizes the loss of accuracy. ■

We first introduce a way to quantify the loss of accuracy when merging two segments of differing configurations. Given the properties of this quantification, we argue that finding the optimal solution is hard. Thus, in the rest of this section, we focus on developing different heuristics to solve the constrained segmentation problem. These heuristics will be compared in the next section.

5.1. Properties of merging segments of differing configurations

Given the p pages of the transactions, we assume without loss of generality that they are all of different configurations. (If this is not true, then we can simply merge those segments with the same configuration into one combined segment, due to Lemma 1.) To reduce p to m_{user}, there is no avoidance of merging segments with differing configurations together. Thus, in the following, we explore various aspects of this operation.

When it comes to merging segments with differing configurations, there exists one immediate complication — namely, the resultant segment may have a totally different configuration. This can be seen by a simple example below.

Example 3 Suppose for 3 items, the configurations for segments T_1 and T_2 are ($a \succeq b \succeq c$) and ($c \succeq b \succeq a$) respectively. This implies that T_1 satisfies the condition that $sup_1(\{a\}) \geq sup_1(\{b\}) \geq sup_1(\{c\})$, and T_2 satisfies $sup_2(\{c\}) \geq sup_2(\{b\}) \geq sup_2(\{a\})$. But once these two segments are merged, the rank ordering of $[sup_1(\{a\}) + sup_2(\{a\})]$, $[sup_1(\{b\}) + sup_2(\{b\})]$, and $[sup_1(\{c\}) + sup_2(\{c\})]$ can be of any one of the 6 possible permutations. ■

Apart from this complication, we know from the earlier analysis that merging segments of differing configurations leads to loss in accuracy. To solve the constrained segmentation problem, we need to capture this loss precisely, leading to the definition of the following quantity. Let $\mathcal{T} = \{T_1, \ldots, T_v\}$ be a set of segments with $v \geq 2$. Then:

$$subop(\mathcal{T}) = \sum_{a_i, a_j} [\widehat{sup}(\{a_i, a_j\}, \mathcal{M}'_1) - \widehat{sup}(\{a_i, a_j\}, \mathcal{M}'_v)] \quad (2)$$

The first term is the upper bound on $sup(\{a_i, a_j\})$ based on $\mathcal{M}'_1$, which consists of 1 *combined* segment formed by merging all v segments in $\mathcal{T}$. The second term is the upper bound based on $\mathcal{M}'_v$, which keeps the v segments $T_1, \ldots, T_v$ *separated*. The difference between the two terms quantifies the amount of "sub-optimality" on the set $\{a_i, a_j\}$ to have the v segments merged. The quantity $subop(\mathcal{T})$ then sums over all pairs of items to measure the total loss.

The quantity $subop()$ satisfies several natural properties, as stated in the following lemma.

```
Algorithm Greedy
INPUT: $p$ initial segments in $\mathcal{T} = \{T_1, \ldots, T_p\}$
OUTPUT: $\mathcal{T}$ with only $m_{user} < p$ segments
0. initialize a priority queue
1. for each pair of segments $T_1, T_2 \in \mathcal{T}$ do
       compute $subop(\{T_1, T_2\})$ and insert it into the priority queue
2. for ($i = p$ downto $m + 1$) do
3.     obtain the pair $T_1, T_2$ having the min $subop()$ in the queue
4.     merge $T_1, T_2$ to give $T_{new} = T_1 \cup T_2$
5.     remove all pairs in the priority queue involving $T_1$ or $T_2$;
       remove $T_1, T_2$ from $\mathcal{T}$
6.     for all remaining segments $T' \in \mathcal{T}$ do
           insert $subop(\{T_{new}, T'\})$ into the queue, add $T_{new}$ to $\mathcal{T}$
```

Figure 2. A skeleton of the Greedy algorithm

Lemma 2 Let $\mathcal{T}$ be a set of segments:

(a) If all segments are of the same configuration, then $subop(\mathcal{T}) = 0$.

(b) If there exist at least two segments with differing configurations, then $subop(\mathcal{T}) > 0$.

(c) If $\mathcal{T}'$ is another set of segments such that $\mathcal{T} \subseteq \mathcal{T}'$, then $subop(\mathcal{T}) \leq subop(\mathcal{T}')$. ■

While the above lemma shows that the quantity $subop()$ is well-behaved, a complication with the constrained segmentation problem is that there are really too many possibilities to form m_{user} segments from p pages. The total number of possibilities can be enumerated exactly; but because it is not a closed form formula, we omit the details here. We only show below a simple example to illustrate the explosion.

Example 4 Suppose $p = 5$ and $m_{user} = 3$. Then the total number of combinations is given by: (i) the number of combinations for one segment to be of size 3 pages, and the other two of size 1 page each, plus (ii) the number of combinations for two segments to be of size 2 pages each, and the other segment of size 1 page. Thus, there are 25 possible combinations. For $m_{user} = 3$, if p is raised to 6 and to 7, the number of combinations quickly jumps to 90 and to 301 respectively. ■

5.2. Heuristic algorithms for the constrained segmentation problem

Given p initial pages/segments, the constrained segmentation problem amounts to finding m_{user} final segments to minimize the total $subop()$ quantity. Partly due to the complication illustrated in Example 3, and partly due to the huge number of possibilities to make up the m_{user} segments, the optimal segmentation is too expensive to be computed. Thus, in the remainder of this section, we focus on developing heuristic algorithms.

One obvious heuristic is the **Greedy algorithm** shown in Figure 2. Given the p initial segments, it first computes the $subop(\{T_1, T_2\})$ quantity for each pair of segments T_1, T_2. Then the pair of segments with the minimum $subop()$ value is selected and merged. However, the merged segment $T_{new} = T_1 \cup T_2$ may be of a different configuration (see Example 3). Thus, it is necessary to compute $subop(\{T_{new}, T'\})$ for every remaining segment T'. Then the next pair of segments with the minimum $subop()$ value is selected and merged, and so on. This continues until m_{user} segments are obtained.

```
Algorithm RC
INPUT: $p$ initial segments in $\mathcal{T} = \{T_1, \ldots, T_p\}$
OUTPUT: $\mathcal{T}$ with only $m_{user} < p$ segments
1. for ($i = p$ downto $m + 1$) do
2.     pick a random segment $T_1 \in \mathcal{T}$
3.     find the closest segment $T_2$ to $T_1$, i.e.,
       $subop(\{T_1, T_2\}) = min\{subop(\{T_1, T'\}) \mid T' \neq T_1\}$
4.     remove $T_1, T_2$ from $\mathcal{T}$, but add $(T_1 \cup T_2)$ to $\mathcal{T}$
```

Figure 3. A skeleton of the RC algorithm

Let us determine the complexity of the Greedy algorithm. According to equation (2), each computation of the $subop()$ quantity for a pair of segments requires $O(k^2)$ effort, due to $O(k^2)$ pairs of items to sum over. Because there are $O(p^2)$ pairs of segments to compute, the complexity of Step 1 is $O(p^2k^2)$. For Step 2, each insertion and deletion from the priority queue takes $O(\log p^2) = O(\log p)$ effort. But for each iteration of the loop, the total effort is $O(p \times (k^2 + \log p))$. Because the loop in Step 2 is executed $(p - m)$ times, the total complexity of the Greedy algorithm becomes $O(p^2k^2 + p^2 \log p)$. Note that segmentation is a compile-time operation, i.e., it is done only *once* and the result can be used by *many* different mining queries.

Next we describe another heuristic method, the **RC (Random Closest) algorithm** for finding the m_{user} segments that minimize the total $subop()$ quantity. See Figure 3. During each iteration of the loop in the RC algorithm, a segment is randomly selected. Then the segment closest to it is found, i.e., in terms of the smallest $subop()$ value. Merging two segments that are close to each other is a natural strategy because this merge does not incur a high level of "sub-optimality", i.e., it has a low $subop()$ value. On this issue, the RC algorithm is similar to the Greedy algorithm. The key difference, though, is that the RC algorithm starts with a random segment, and does not require a priority queue to be maintained to find the pair of segments that give the lowest $subop()$ value. In terms of complexity, each iteration takes $O(pk^2)$ effort. Because the loop is executed $(p - m)$ times, the total complexity of the RC algorithm is $O(p^2k^2)$.

From the Greedy algorithm to the RC algorithm, we relax in the latter algorithm the requirement in the former algorithm that the pair of segments with the absolute minimum $subop()$ value be found. To go in the same direction one step further, we can even relax the requirement in the RC algorithm that the closest segment to T_1 be found. This leads to the **Random algorithm**. It is almost identical to the skeleton shown in Figure 3, except that Step 3 is now replaced by a simple random selection of a segment T_2.[5]

Before we dismiss this algorithm as too simplistic, there are two reasons why we include this heuristic in our discussion and experimentation. First, the Random algorithm is fast, with a complexity of $O(p)$. So in our experimentation, it forms a good baseline for us to judge whether the other more sophisticated heuristics are cost-effective. Second, as

[5]Similar to the construction of the SSM structure [10], the Random algorithm constructs the OSSM by arbitrarily/randomly partitioning pages of transactions into segments.

will be shown very soon, the Random algorithm, because of its speed, can form part of a hybrid strategy for efficient segmentation.

5.3. The bubble list optimization: Dealing with the k^2 factor

Note that in the Random algorithm, there is no need to compute any $subop()$ value. This is why there is no k^2 factor in the complexity of the Random algorithm. But in the other two algorithms, because selection of segments is based entirely on the $subop()$ values, k^2 becomes a dominant factor in their complexity. To eliminate the k^2 factor, one heuristic is to find a constant number of items which are "on the bubble". These are the items whose frequencies barely satisfy, and are the closest to, the support threshold. Then in the $subop()$ calculation, we restrict the summation in equation (2) only over all pairs of items in the "bubble list". This is reasonable because the filtering offered by the OSSM is expected to be the most applicable to those items whose frequencies are closest to the threshold. For example, there may be $k = 1000$ items in all, but the bubble list may contain only $w = 100$ items. In this case, for *each* computation of $subop()$, the number of pairs considered in the summation reduces from $\binom{k}{2} = 499\,500$ to $\binom{w}{2} = 4950$.

Note that the content of the bubble list depends on a support threshold. This bubble list then turns the focus of the segmentation squarely on the items in the list. But it is important to remember that while segmentation with the bubble list requires some support threshold, the OSSM produced can be used for *any* support threshold. In the next section, we will evaluate the effectiveness of an OSSM that was produced with a bubble list based on one support threshold, but is used dynamically to handle different support thresholds.

5.4. Hybrid segmentation strategies: Dealing with the p^2 factor

With the bubble list heuristic, the k^2 factor disappears from the complexity of the Greedy and RC algorithms. What about the p^2 factor? As a heuristic to cope with situations when p is large, we offer hybrid segmentation strategies. Examples include **Random-RC** and **Random-Greedy**. That is, for a large initial value p, in the first phase we use the Random algorithm to merge the p pages to p_{mid} segments (where $p_{mid} > m_{user}$). Then in the final phase, the RC or the Greedy algorithm is used to merge the p_{mid} segments into the m_{user} segments. In the next section, we will evaluate whether these hybrid strategies work at all.

6. Experimental evaluation

In this section, we provide extensive experimental evaluation of all the segmentation algorithms and heuristics proposed above. Specifically, we provide empirical evaluation on:

1. the speedup an OSSM provides at query execution or exploration time; and
2. the segmentation cost of producing the OSSM, with or without the bubble list and the hybrid strategies.

We conclude this section with a recommended recipe for different circumstances.

6.1. Experimental setup

The segment minimization and the constrained segmentation problems apply to the many algorithms that conduct mining of various kinds of patterns based on candidate generation (see Section 1). The results reported here are based on a specific instance of these algorithms — namely, the classical Apriori algorithm for association rule mining. Nonetheless, the presented ideas and algorithms apply to other instances as well.

All the experiments were performed on a time-sharing environment on a 700 MHz machine. All the programs were written in C. There are two kinds of execution time. The first is the segmentation time, which includes all CPU and I/O costs incurred for segmentation. The second is the runtime of Apriori with or without the OSSM, which includes all CPU and I/O costs to find all the frequent sets. The reported figures are based on the average of multiple runs. There are three data sets we used:

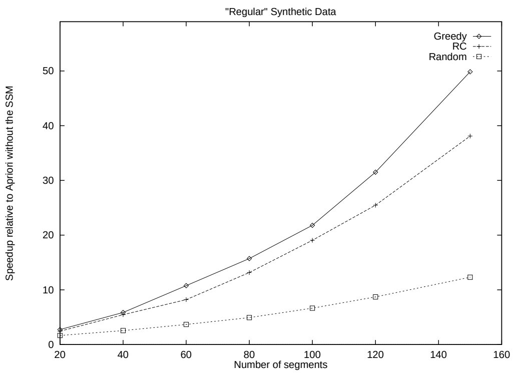
1. **Nokia data set**, which is a real data set from Nokia on a sequence file containing about 5000 transactions of about 200 distinct types of telecommunications network alarms. For proprietary reasons, we cannot describe this data set further.
2. **regular-synthetic data set**, which is a synthetic data set generated using the program developed at IBM Almaden Research Center [3]. The exact number of transactions is not important, because the key parameter is the number of pages p. In our experimentation, p varies from 200 to 50 000, and the number of items is $k = 1000$.
3. **skewed-synthetic data set**, which is a synthetic data set that has skewed "seasonal" behavior. Specifically, 50% of the items have a higher probability of appearing in the first half of the collection of transactions, and the other 50% have a higher probability of appearing in the second half. The reason for using this data set is that in the real world, there are many databases that do not follow a regular or uniform distribution (e.g., a supermarket database consisting of transactions over a few months from summer to winter).

6.2. Runtime speedup

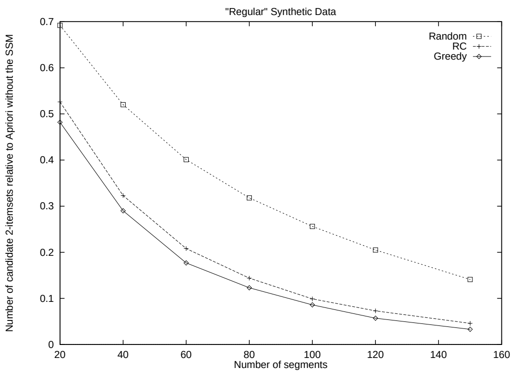
The first set of experiments evaluates whether the OSSM provides significant speedup. The speedup for segmentation algorithm X is defined to be the ratio of the execution time of the Apriori algorithm without the OSSM, to that with the OSSM produced by algorithm X (where X is the Random, RC, or Greedy algorithm). This gives the y-axis of the graph in Figure 4(a). The x-axis shows the number of segments m_{user} allowed in the OSSM.

Segmentation algorithms aside, the first observation from Figure 4(a) is that the OSSM is a useful structure. It is light-weight. For $m_{user} = 100$ segments, and for 1000 items, the OSSM consumes only about 0.2 megabytes. But it can bring about a speedup from around 7 times to about 22 times. And for m_{user} = 150 segments (thus consuming 0.3 megabytes), the speedup can be about 50 times.

The second observation is that as expected, the larger the



(a) Relative speedup



(b) Fraction of candidate 2-itemsets not pruned

Figure 4. Effectiveness of the segmentation algorithms

value of m_{user}, the higher is the speedup. Interestingly, even the Random algorithm can offer a speedup that is better than an order of magnitude. In terms of the speedup, the rank in descending order is always the Greedy algorithm, the RC algorithm and the Random algorithm. The gaps between them open up gradually as m_{user} becomes larger.

The speedup can be directly linked to the number of candidates that require frequency counting, i.e., candidates that are not pruned based on equation (1). Figure 4(b) compares the number of candidate 2-itemsets required with or without the OSSM. Again the ratio is shown on the y-axis, the ratio 1 being the case without the OSSM. Clearly, the OSSM provides significant pruning. For example with $m_{user} = 150$, and the OSSM produced by the Greedy algorithm, only about 3% of candidate 2-itemsets checked by Apriori ordinarily are not pruned by the information kept in the OSSM. Figure 4(b) only shows the reduction for candidate 2-itemsets. The OSSM applies to candidate sets of higher cardinalities. Given our synthetic data sets, the reduction for higher cardinalities is minimal. But this is not as negative as it may seem, because it is well known that the main bottleneck of the Apriori algorithm is on its processing of candidate 2-itemsets [15]; this is precisely the area where OSSM excels.

The results reported in Figure 4 are based on the regular-synthetic data set, and a support threshold of 1%. In addition, we have experimented with other data sets mentioned

Pure strategy	Segmentation time	Speedup
Random	0.02 ± 0.00s	2.6 ± 0.1
RC	2791 ± 7s	5.5 ± 0.4
Greedy	5439 ± 6s	5.9 ± 0.3

(a) Pure strategies with p = 500

Hybrid strategy	Segmentation time	Speedup
Random-RC	521 ± 2s	4.3 ± 0.0
Random-Greedy	1051 ± 1s	5.2 ± 0.1

(b) Hybrid strategies with p = 50 000, p_{mid} = 200

Figure 5. Segmentation costs: Pure and hybrid strategies with m_{user} = 40

above and various support thresholds. Please refer to our technical report [12] for more details.

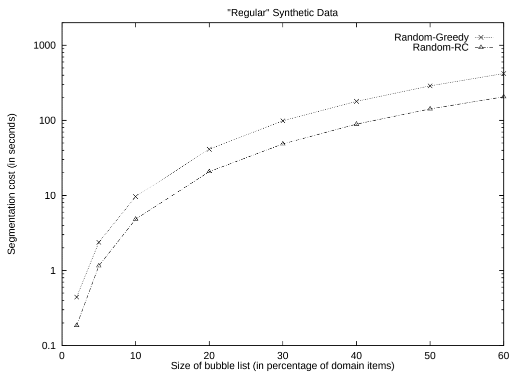
6.3. Segmentation cost

In light of the significant speedup offered by an OSSM, the natural question to ask is whether this speedup is achieved by a high “compile-time” segmentation cost. The key parameters here are p and m_{user}. While we experimented with many combinations, we only report some of them below, due to a lack of space. For example, Figure 5(a) reports the results for $p = 500$ and $m_{user} = 40$.

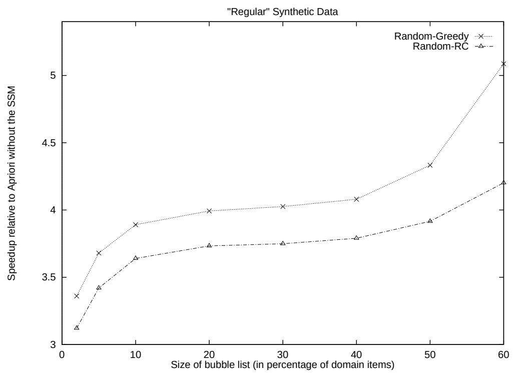
Effectiveness of the pure segmentation strategies: As expected, both the RC and the Greedy algorithms take a long time to produce the OSSM. But is it too long? Is the additional one-time cost of 5439 seconds worthwhile to increase the speedup from 2.6 to 5.9 for *each* mining query? The answer to this question is subjective, depending on the relative importance of segmentation cost to dynamic query execution cost for the specific application. It also depends on the amount of space the OSSM can occupy, i.e., the value of m_{user}. In figures, we deliberately take a smaller value $m_{user} = 40$ to allow the segmentation process to take longer. If the application can afford a larger value of m_{user}, the segmentation cost *drops* while the speedup *increases*. In this case, the choice becomes more obvious — namely, it pays off to use a more elaborate segmentation algorithm to produce a higher-quality OSSM.

Effectiveness of the hybrid segmentation strategies: Given the numbers in Figure 5(a), a natural question to ask is how well the more elaborate algorithms scale up with respect to p. If it takes 5439 seconds for the Greedy algorithm for $p = 500$ pages, how long does it take for $p =$ 50 000 pages? (For a page size of 4 kilobytes, each page can contain roughly 100 transactions. Thus, 50 000 pages correspond to 5 million transactions.) This is an important question because data mining applications are supposed to have a huge number of transactions (and pages).

Fortunately, the Random algorithm comes to the rescue. Recall from Figure 5(a) that the Random algorithm takes a negligible amount of segmentation time, but still delivers reasonable speedup. This opens the possibility of hybrid segmentation strategies for large values of p. Figure 5(b) shows the situation for the hybrid strategies Random-RC and Random-Greedy. More specifically, the initial value of p is 50 000. The Random algorithm is used to quickly reduce p to $p_{mid} = 200$ segments. From that point on, either



(a) Segmentation time



(b) Speedup

Figure 6. Effectiveness of the bubble list optimization

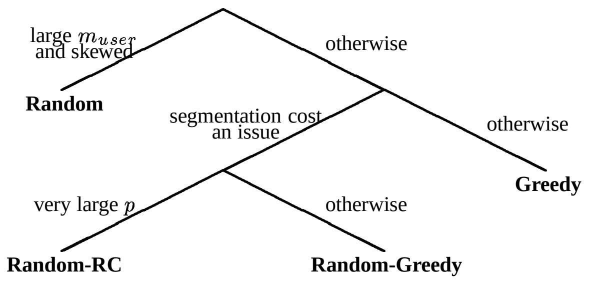


Figure 7. Recommended recipe

the RC or the Greedy algorithm is used to select the final $m_{user} = 40$ segments.

Notice that for Random-RC, the total segmentation time for 50 000 pages is only 521 seconds, as supposed to 2791 seconds for only 500 pages using purely the RC algorithm. Yet there is a minimal drop in speedup. Similar observations can be obtained for Random-Greedy. In general, given a large initial p value, a good strategy is to use the Random algorithm to reduce p to a much smaller p_{mid} value (e.g., between 100 to 500 pages), and then to use an elaborate algorithm to complete the segmentation.

Effectiveness of the bubble list optimization: Apart from the hybrid strategies, we also propose the bubble list heuristic, with the objective of focusing the computation to the $subop()$ value only to those items that are on the bubble list. Figure 6 shows the situation when the bubble list was formed based on the support threshold 0.25%, and yet during query evaluation, the actual support threshold is 1%. The x-axis shows the varying size of the bubble list, expressed as the percentage of k, the total number of items in the domain. The key observation is that the segmentation cost is drastically reduced. For example, the Random-Greedy hybrid strategy with the bubble list can produce an OSSM in about 10 seconds of total time, for 5 million transactions (i.e. 50 000 pages), as opposed to 1051 seconds for that without the bubble list (cf.: Figure 5(b)). A similar reduction applies to the Random-RC hybrid strategy. This shows that the bubble list heuristic is very effective in reducing the segmentation cost.

Figure 6(b) shows that even though the segmentation time is significantly reduced, the quality of the OSSM produced by the hybrid strategies is not compromised significantly. Furthermore, even though the support threshold used during segmentation is different from the one used at query execution time, the speedup offered by the OSSM is still significant. As expected, the longer the bubble list, the higher is the quality of the OSSM, and thus the speedup.

6.4. Summary: A recommended recipe

In sum, we have provided extensive experimental results evaluating the various proposed segmentation algorithms and heuristics. Figure 7 shows a "recipe" we recommend for deciding on which segmentation algorithm to use for various kinds of applications. First, if the application can afford a lot of space for the OSSM (i.e., m_{user} is large), and the data is skewed, the Random algorithm, which is the simplest, is sufficient for segmentation. Otherwise, if the segmentation cost is not an issue at all, the Greedy algorithm with the bubble list is the choice. However, if the number of initial pages p is large, then we recommend either the Random-RC or the Random-Greedy algorithm, with the bubble list.

7. Discussion

In previous sections, we have shown how the OSSM helps to improve the efficiency of the Apriori algorithm (an instance of pattern discovery algorithms). Being a generic data structure, the OSSM can be equally applicable to the discovery of sequential patterns, episodes, constrained frequent sets, etc. As mentioned in Section 2, the OSSM technique is rather different from the DHP algorithm [15] and the Partition algorithm [17]. But an observant reader may wonder whether the OSSM provides better pruning than the two existing algorithms. However, this is the wrong question to ask because the OSSM can be made to work in *conjunction* with the two algorithms.

Recall that the DHP algorithm hashes a k-itemset (e.g., $k = 2$) to a bucket, which may eventually be pruned due to an insufficient number of itemsets being hashed into such a bucket. However, if an OSSM is used simultaneously, then known infrequent k-itemsets are not generated in the first place. Itemsets that pass through the pruning by the OSSM can now be further pruned by the DHP algorithm. A preliminary result, presented in the following table, shows the additional benefit brought by an OSSM (constructed using the Random-RC algorithm with m = 40 segments) to the DHP algorithm with 32 768 buckets.

Algorithms	Runtime	No. of C_2
DHP without the OSSM	4.01 ± 0.13s	292
DHP with the OSSM	1.96 ± 0.01s	142

Here, when the DHP algorithm is used in conjunction with the OSSM, the number of candidate 2-itemsets (C_2) is about half and the speedup is about 2 times (when compared to the DHP algorithm without the OSSM).

Similarly, the OSSM can bring additional benefits to some other algorithms. For lack of space, we briefly discuss below how the OSSM can be applicable to the Partition algorithm [17] and to the DepthProject algorithm [1].

For the Partition algorithm, if an OSSM is built for each partition, the execution time for each partition will be significantly reduced because known local infrequent itemsets are pruned by the OSSM. To improve the performance further, if the OSSMs for all the partitions are available, then many of the global candidates (i.e., itemsets that are locally frequent in a partition), which are known to be globally infrequent with respect to the OSSMs, can in fact be pruned.

Recently, a pattern discovery algorithm, called Depth-Project, was proposed to generate long patterns by using depth-first search on a lexicographic tree of itemsets. More precisely, at each step, the algorithm generates possible frequent lexicographic extensions (i.e. candidates) of a tree node and tests for frequency. If an OSSM is used simultaneously, then known infrequent candidates can be pruned before the frequency counting.

8. Conclusions

In this paper, we proposed a light-weight structure called optimized segment support map (OSSM). In addition to improving the pruning in pattern discovery algorithms, it also provides direct information about the variability of frequencies in different segments of the transactions. Unlike many algorithms which cannot handle skewed data, the strength of the OSSM is to exploit the variability. Concerning the OSSM, we studied two main problems: (i) the minimum number of segments for an OSSM to incur no loss in accuracy (the segment minimization problem), and (ii) the best composition of the segments given a user-determined number of segments (the constrained segmentation problem).

For the first problem, we provided a thorough analysis and hardness results, showing that no loss in accuracy using the OSSM requires too many segments in general. For the second problem, we developed the heuristic segmentation algorithms called the Random, RC, and Greedy algorithms. To further reduce segmentation cost, we proposed optimizations that use a bubble list and run the algorithms in a hybrid fashion. Our experimental results are strong indicating that for a small OSSM (e.g., 0.3 megabytes), the speedup can be very significant (e.g., 50 times), yet the segmentation cost is small (e.g., 10 seconds for 5 million transactions). We concluded by presenting a recommended recipe for various circumstances. While our experiments were based on finding frequent sets using the Apriori algorithm, it is important to remember the generality of the OSSM: It is applicable to many pattern discovery algorithms (many of which are listed in the introduction). We expect the OSSM to be equally effective for those algorithms.

References

[1] R.C. Agarwal, C.C. Aggarwal, and V.V.V. Prasad. Depth first generation of long patterns. In *Proc. KDD 2000*, pp 108–118.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. SIGMOD 1993*, pp 207–216.

[3] R. Agrawal, H. Mannila, et al. Fast discovery of association rules. *Advances in Knowledge Discovery and Data Mining*, pp 307–328. AAAI/MIT Press, 1996.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. ICDE 1995*, pp 3–14.

[5] R.J. Bayardo. Efficiently mining long patterns from databases. In *Proc. SIGMOD 1998*, pp 85–93.

[6] S. Brin, R. Motwani, and C. Silverstein. Beyond market basket: Generalizing association rules to correlations. In *Proc. SIGMOD 1997*, pp 265–276.

[7] G. Grahne, L.V.S. Lakshmanan, and X. Wang. Efficient mining of constrained correlated sets. In *Proc. ICDE 2000*, pp 512–521.

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. SIGMOD 2000*, pp 1–12.

[9] C. Hidber. Online association rule mining. In *Proc. SIGMOD 1999*, pp 145–156.

[10] L.V.S. Lakshmanan, C.K.-S. Leung, and R.T. Ng. The segment support map: Scalable mining of frequent itemsets. *SIGKDD Explorations*, **2**(2), pp 21–27, Dec. 2000.

[11] L.V.S. Lakshmanan, R. Ng, et al. Optimization of constrained frequent set queries with 2-variable constraints. In *Proc. SIGMOD 1999*, pp 157–168.

[12] C.K.-S. Leung, R.T. Ng, and H. Mannila. The optimized segment support map for the mining of frequent patterns. Technical Report CS TR-2001-18, The University of British Columbia, Canada, 2001.

[13] H. Mannila, H. Toivonen, and A.I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, **1**(3), pp 259–289, Sept. 1997.

[14] R.T. Ng, L.V.S. Lakshmanan, et al. Exploratory mining and pruning optimizations of constrained association rules. In *Proc. SIGMOD 1998*, pp 13–24.

[15] J.S. Park, M.-S. Chen, and P.S. Yu. Using a hash-based method with transaction trimming for mining association rules. *IEEE TKDE*, **9**(5), pp 813–825, Sept./Oct. 1997.

[16] N. Pasquier, Y. Bastide, et al. Discovering frequent closed itemsets for association rules. In *Proc. ICDT 1999*, pp 398–416.

[17] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. VLDB 1995*, pp 432–443.

[18] C. Silverstein, S. Brin, et al. Scalable techniques for mining causal structures. In *Proc. VLDB 1998*, pp 594–605.

[19] R. Srikant, Q. Vu, and R. Agrawal. Mining associations rules with item constraints. In *Proc. KDD 1997*, pp 67–73.

[20] M.J. Zaki and K. Gouda. Fast vertical mining using diffsets. Technical Report 01-1, Rensselaer Polytechnic Institute, USA, 2001.

[21] M.J. Zaki and C.-J. Hsiao. CHARM: An efficient algorithm for closed association rule mining. Technical Report 99-10, Rensselaer Polytechnic Institute, USA, 1999.