Sequential PAttern Mining using A Bitmap Representation

Jay Ayres, Johannes Gehrke, Tomi Yiu, and Jason Flannick
Dept. of Computer Science
Cornell University

ABSTRACT

We introduce a new algorithm for mining sequential patterns. Our algorithm is especially efficient when the sequential patterns in the database are very long. We introduce a novel *depth-first* search strategy that integrates a depth-first traversal of the search space with effective pruning mechanisms.

Our implementation of the search strategy combines a vertical bitmap representation of the database with efficient support counting. A salient feature of our algorithm is that it incrementally outputs new frequent itemsets in an online fashion.

In a thorough experimental evaluation of our algorithm on standard benchmark data from the literature, our algorithm outperforms previous work up to an order of magnitude.

1. INTRODUCTION

Finding sequential patterns in large transaction databases is an important data mining problem. The problem of mining sequential patterns and the support-confidence framework were originally proposed by Agrawal and Srikant [2, 10]. Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of items. We call a subset $X \subseteq I$ an *itemset* and we call $|X|$ the *size* of X. A *sequence* $s = (s_1, s_2, \ldots, s_m)$ is an ordered list of itemsets, where $s_i \subseteq I, i \in \{1, \ldots, m\}$. The size, m, of a sequence is the number of itemsets in the sequence, i.e. $|s|$. The length l of a sequence $s = (s_1, s_2, \ldots, s_m)$ is defined as

$$l \stackrel{\text{def}}{=} \sum_{i=1}^{m} |s_i|.$$

A sequence with length l is called an l-sequence. A sequence $s_a = (a_1, a_2, \ldots, a_n)$ is contained in another sequence $s_b = (b_1, b_2, \ldots, b_m)$ if there exist integers $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ such that $a_1 \subseteq b_{i_1}$, $a_2 \subseteq b_{i_2}$, $\ldots$, $a_n \subseteq b_{i_n}$. If sequence s_a is contained in sequence s_b, then we call s_a a *subsequence* of s_b and s_b a *supersequence* of s_a.

A database D is a set of tuples (cid, tid, X), where cid is a customer-id, tid is a transaction-id based on the transaction time, and X is an itemset such that $X \subseteq I$. Each tuple in D is referred to as a *transaction*. For a given customer-id, there are no transactions with the same transaction ID. All the transactions with the same cid can be viewed as a sequence of itemsets ordered by increasing tid. An analogous representation for the database is thus a set of sequences of transactions, one sequence per customer, and we refer to this dual representation of D as its *sequence representation*.

Customer ID (CID)	TID	Itemset
1	1	$\{a, b, d\}$
1	3	$\{b, c, d\}$
1	6	$\{b, c, d\}$
2	2	$\{b\}$
2	4	$\{a, b, c\}$
3	5	$\{a, b\}$
3	7	$\{b, c, d\}$

Table 1: Dataset sorted by CID and TID

CID	Sequence
1	$(\{a, b, d\}, \{b, c, d\}, \{b, c, d\})$
2	$(\{b\}, \{a, b, c\})$
3	$(\{a, b\}, \{b, c, d\})$

Table 2: Sequence for each customer

The *absolute support* of a sequence s_a in the sequence representation of a database D is defined as the number of sequences $s \in D$ that contain s_a, and the *relative support* is defined as the percentage of sequences $s \in D$ that contain s_a. We will use absolute and relative support interchangeably in the rest of the paper. The support of s_a in D is denoted by $sup_D(s_a)$. Given a support threshold $minSup$, a sequence s_a is called a *frequent sequential pattern on* D if $sup_D(s_a) \geq minSup$. The problem of mining sequential patterns is to find all frequent sequential patterns for a database D, given a support threshold sup.

Table 1 shows the dataset consisting of tuples of (customer id, transaction id, itemset) for the transaction. It is sorted by customer id and then transaction id. Table 2 shows the database in its sequence representation. Consider the sequence of customer 2; the size of this sequence is 2, and the length of this sequence is 4.

Suppose we want to find the support of the sequence $s_a = (\{a\}, \{b, c\})$. From Table 2, we know that s_a is a subsequence of the sequences for customer 1 and customer 3 but is not a subsequence of the sequence for customer 2.

SIGKDD '02 Edmonton, Alberta, Canada

Hence, the support of s_a is 2 (out of a possible 3), or 0.67. If the user-defined minimum support value is less than 0.67, then s_a is deemed frequent.

1.1 Contributions of This Paper

In this paper, we take a systems approach to the problem of mining sequential patterns. We propose an efficient algorithm called SPAM (**S**equential **PA**ttern **M**ining) that integrates a variety of old and new algorithmic contributions into a practical algorithm. *SPAM* assumes that the entire database (and all data structures used for the algorithm) completely fit into main memory. With the size of current main memories reaching gigabytes and growing, many moderate-sized to large databases will soon become completely memory-resident. Considering the computational complexity that is involved in finding long sequential patterns even in small databases with wide records, this assumption is not very limiting in practice. Since all algorithms for finding sequential patterns, including algorithms that work with disk-resident databases, are CPU-bound, we believe that our study sheds light on the most important performance bottleneck.

SPAM is to the best of our knowledge the first depth-first search strategy for mining sequential patterns. An additional salient feature of SPAM is its property of online outputting sequential patterns of different length — compare this to a breadth-first search strategy that first outputs all patterns of length one, then all patterns of length two, and so on. Our implementation of SPAM uses a vertical bitmap data layout allowing for simple, efficient counting.

2. THE SPAM ALGORITHM

In this section, we will describe the lexicographic tree of sequences upon which our algorithm is based. We will also discuss the way we traverse the tree and the pruning methods that we use to reduce the search space.

2.1 Lexicographic Tree for Sequences

This part of the paper describes the conceptual framework of the sequence lattice upon which our approach is based. A similar approach has been used for the problem of mining frequent itemsets in *MaxMiner* [3] and *MAFIA* [5]. We use this framework to describe our algorithm and some pertinent related works. Assume that there is a lexicographical ordering $\leq$ of the items I in the database. If item i occurs before item j in the ordering, then we denote this by $i \leq_I j$. This ordering can be extended to sequences by defining $s_a \leq s_b$ if s_a is a subsequence of s_b. If s_a is not a subsequence of s_b, then there is no relationship in this ordering.

Consider all sequences arranged in a *sequence tree* (abbreviated, *tree*) with the following structure. The root of the tree is labeled with $\emptyset$. Recursively, if n is a node in the tree, then n's children are all nodes n' such that $n \leq n'$ and $\forall m \in T : n' \leq m \implies n \leq m$. Notice that given this definition the tree is infinite. Due to a finite database instance as input to the problem, all trees that we deal with in practice are finite.

Each sequence in the sequence tree can be considered as either a sequence-extended sequence or an itemset-extended sequence. A sequence-extended sequence is a sequence generated by adding a new transaction consisting of a single item to the end of its parent's sequence in the tree. An itemset-extended sequence is a sequence generated by adding an item to the last itemset in the parent's sequence, such that the item is greater than any item in that last itemset. For example, if we have a sequence $s_a = (\{a, b, c\}, \{a, b\})$, then $(\{a, b, c\}, \{a, b\}, \{a\})$ is a sequence-extended sequence of s_a and $(\{a, b, c\}, \{a, b, d\})$ is an itemset-extended sequence of s_a.

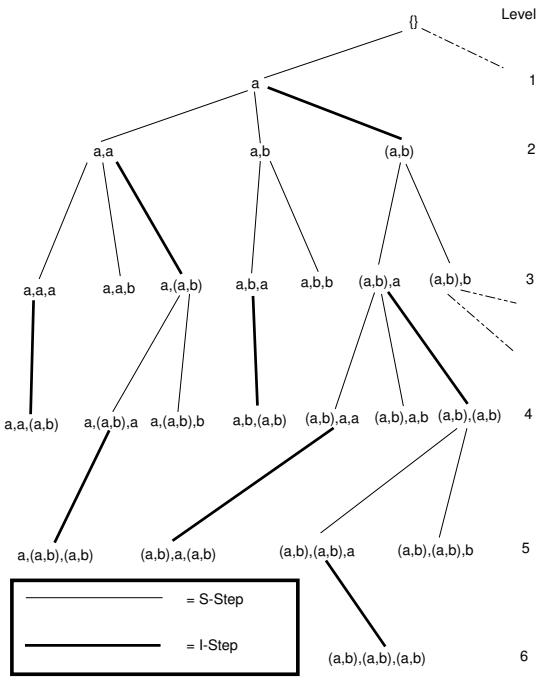

Figure 1: The Lexicographic Sequence Tree

If we generate sequences by traversing the tree, then each node in the tree can generate sequence-extended children sequences and itemset-extended children sequences. We refer to the process of generating sequence-extended sequences as the *sequence-extension step* (abbreviated, the *S-step*), and we refer to the process of generating itemset-extended sequences as the *itemset-extension step* (abbreviated, the *I-step*). Thus we can associate with each node n in the tree two sets: S_n, the set of candidate items that are considered for a possible *S-step* extensions of node n (abbreviated *s-extensions*), and I_n, which identifies the set of candidate items that are considered for a possible *I-step* extensions (abbreviated, *i-extensions*.

Figure 1 shows a sample of a complete sequence tree for *two* items, a and b, given that the maximum size of a sequence is *three*. The top element in the tree is the null sequence and each lower level k contains all of the k-sequences, which are ordered lexicographically with sequence-extended sequences ordered before itemset-extended sequences. Each element in the tree is generated only by either an *S-step* or an *I-step*, e.g. sequence $(\{a, b\}, \{b\})$ is generated from sequence $(\{a, b\})$ and not from sequence $(\{a\}, \{b\})$ or $(\{b\}, \{b\})$.

2.2 Depth First Tree Traversal

SPAM traverses the sequence tree described above in a standard depth-first manner. At each node n, the support of each sequence-extended child and each itemset-extended child is tested. If the support of a generated sequence s is greater than or equal to *minSup*, we store that sequence

and repeat DFS recursively on s. (Note that the maximum length of any sequence is limited since the input database is finite.) If the support of s is less than *minSup*, then we do not need to repeat DFS on s by the Apriori principle, since any child sequence generated from s will not be frequent [2]. If none of the generated children are frequent, then the node is a leaf and we can backtrack up the tree.

2.3 Pruning

The previous algorithm has a huge search space. To improve the performance of our algorithm, we can prune candidate s-extensions and i-extensions of a node n in the tree. Our pruning techniques are Apriori-based and are aimed at minimizing the size of S_n and I_n at each node n. At the same time, the techniques guarantee that all nodes corresponding to frequent sequences are visited.

2.3.1 S-step Pruning

One technique used prunes *S-step* children. Consider a sequence s at node n and suppose its sequence-extended sequences are $s_a = (s, \{i_j\})$ and $s_b = (s, \{i_k\})$. Suppose s_a is frequent but s_b is not frequent. By the Apriori principle, $(s, \{i_j\}, \{i_k\})$ and $(s, \{i_j, i_k\})$ can not be frequent, since both contain the subsequence s_b. Hence, we can remove i_k from both S_m and I_m, where m is any node corresponding to a frequent sequence-extended child of s.

Example

Suppose we are at node $(\{a\})$ in the tree and suppose that

$$S_{(\{a\})} = \{a, b, c, d\}, I_{(\{a\})} = \{b, c, d\}.$$

The possible sequence-extended sequences are $(\{a\}, \{a\})$, $(\{a\}, \{b\})$, $(\{a\}, \{c\})$ and $(\{a\}, \{d\})$. Suppose that $(\{a\}, \{c\})$ and $(\{a\}, \{d\})$ are not frequent. By the Apriori principle, we know that none of the following sequences can be frequent either: $(\{a\}, \{a\}, \{c\})$, $(\{a\}, \{b\}, \{c\})$, $(\{a\}, \{a, c\})$, $(\{a\}, \{b, c\})$, $(\{a\}, \{a\}, \{d\})$, $(\{a\}, \{b\}, \{d\})$, $(\{a\}, \{a, d\})$, or $(\{a\}, \{b, d\})$. Hence, when we are at node $(\{a\}, \{a\})$ or $(\{a\}, \{b\})$, we do not have to perform *I-step* or *S-step* using items c and d, i.e. $S_{(\{a\},\{a\})} = S_{(\{a\},\{b\})} = \{a, b\}$, $I_{(\{a\},\{a\})} = \{b\}$, and $I_{(\{a\},\{b\})} = \emptyset$.

2.3.2 I-step Pruning

The second technique prunes *I-step* children. Consider the itemset-extended sequences of $s = (s', \{i_1, \ldots, i_n\})$. Suppose the sequences are $s_a = (s', \{i_1, \ldots, i_n, i_j\})$ and $s_b = (s', \{i_1, \ldots, i_n, i_k\})$, and suppose that $i_j < i_k$. If s_a is frequent and s_b is not frequent, then by the Apriori principle, $(s', \{i_1, \ldots, i_n, i_j, i_k\})$ cannot be frequent. Hence, we can remove i_k from I_m, where m is any node corresponding to a frequent itemset-extended child of s. We can remove from S_m the same items as we did using *S-step* pruning. For suppose that during *S-step* pruning $(s, \{i_l\})$ was discovered to be infrequent. Then so too will $(s_a, \{i_l\})$ be infrequent for any frequent itemset-extended child s_a of s.

Figure 2 shows the pseudocode of our algorithm after both *I-step* pruning and *S-step* pruning are added. After pruning the correct items from S_m and I_m, we pass the new lists to the next recursive call. The lists are pruned exactly as described in the previous sections.

DFS-Pruning(node $n = (s_1, \ldots, s_k)$, S_n, I_n)

```
(1)   S_temp = ∅
(2)   I_temp = ∅
(3)   For each (i ∈ S_n)
(4)       if ( (s_1, ..., s_k, {i}) is frequent )
(5)           S_temp = S_temp ∪ {i}
(6)   For each (i ∈ S_temp)
(7)       DFS-Pruning((s_1, ..., s_k, {i}), S_temp,
              all elements in S_temp greater than i)
(8)   For each (i ∈ I_n)
(9)       if ((s_1, ..., s_k ∪ {i}) is frequent)
(10)          I_temp = I_temp ∪ {i}
(11)  For each (i ∈ I_temp)
(12)      DFS-Pruning((s_1, ..., s_k ∪ {i}), S_temp,
              all elements in I_temp greater than i)
```

Figure 2: Pseudocode for DFS with pruning

Example

Let us consider the same node $(\{a\})$ described in the previous section. The possible itemset-extended sequences are $(\{a, b\})$, $(\{a, c\})$, and $(\{a, d\})$. If $(\{a, c\})$ is not frequent, then $(\{a, b, c\})$ must also not be frequent by the Apriori principle. Hence, $I_{(\{a,b\})} = \{d\}$, $S_{(\{a,b\})} = \{a, b\}$, and $S_{(\{a,d\})} = \{a, b\}$.

3. DATA REPRESENTATION

3.1 Data Structure

To allow for efficient counting, our algorithm uses a vertical bitmap representation of the data. A vertical bitmap is created for each item in the dataset, and each bitmap has a bit corresponding to each transaction in the dataset. If item i appears in transaction j, then the bit corresponding to transaction j of the bitmap for item i is set to *one*; otherwise, the bit is set to *zero*.

To enable efficient counting and candidate generation, we partition the bitmap such that all of the transactions of each sequence in the database will appear together in the bitmap (Figure 3). If transaction m is before transaction n in a sequence, then the index of the bit corresponding to m is smaller than that of the bit corresponding to n. We also use different sized bitmaps depending on how many transactions are present in a sequence; we postpone a discussion of this until we discuss support counting.

This bitmap idea extends naturally to itemsets. Suppose we have a bitmap for item i and a bitmap for item j. The bitmap for the itemset $\{i, j\}$ is simply the bitwise *AND* of these two bitmaps. Sequences can also be represented using bitmaps. If the last itemset of the sequence is in transaction j and all the other itemsets of the sequence appear in transactions before j, then the bit corresponds to j will be set to *one*; otherwise, it will be set to *zero*. We define $B(s)$ as the bitmap for sequence s.

Efficient counting of support is one of the main advantages of the vertical bitmap representation of the data. To achieve this, we partition the customer sequences into different sets based on their lengths when we read in the dataset. If the size of a sequence is between $2^k + 1$ and 2^{k+1}, we consider it as a 2^{k+1}-bit sequence. The minimum value of k is 1 (i.e.

CID	TID	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$
1	1	1	1	0	1
1	3	0	1	1	1
1	6	0	1	1	1
-	-	0	0	0	0
2	2	0	1	0	0
2	4	1	1	1	0
-	-	0	0	0	0
-	-	0	0	0	0
3	5	1	1	0	0
3	7	0	1	1	1
-	-	0	0	0	0
-	-	0	0	0	0

Figure 3: Bitmap Representation of the dataset in Figure 1

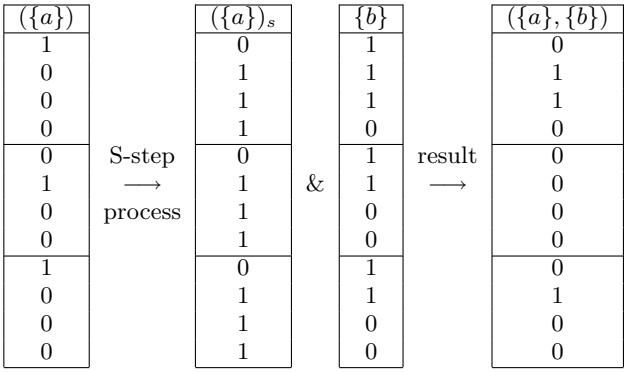

Figure 4: S-step processing on sequence bitmap $(\{a\})$ shown in Figure 3

any sequence of size smaller than or equal to 4 is considered as a 4-bit sequence). We ignore any sequence of size larger than 64 because in most practical databases this situation will not arise. Each set of 2^k-bit sequences will correspond to a different bitmap, and in that bitmap each section will be 2^k-bits long. Support counting for each customer then becomes a simple check as to whether the corresponding bitmap partition contains all zeros or not.

Let us consider the dataset that is shown in Table 1. The bitmap representation of this dataset is shown in Figure 3. Each vertical bitmap is partitioned into three sections, and each section corresponds to a customer's sequence. Since all of the customers' sequences have less than four transactions, all of them are represented as 4-bit sequences. Transaction 6 contains items b,c, and d, so the bit that corresponds to that transaction in each of the bitmaps b,c, and d is set to *one*. Since transaction 6 does not contain the item a, the bit corresponding to that transaction in bitmap a is set to *zero*.

3.2 Candidate Generation

In this section, we will describe how we perform candidate generations using the bitmap representation described above. We first consider *S-step* processing, followed by *I-step* processing.

3.2.1 S-step Process

Suppose we are given bitmaps $B(s_a)$ and $B(i)$ for sequence s_a and item i respectively, and that we want to perform a *S-step* on s_a using i. This *S-step* will append the itemset $\{i\}$ to s_a. The bitmap for the new sequence, $B(s_g)$, should have the property that if a bit has value *one*, then the corresponding transaction j must contain i, and all of the other itemsets in s_g must be contained in transactions before j.

Let us consider a section (a customer's sequence) of the bitmaps; we will refer it as a bitmap in this discussion. Suppose the index of the first bit with value *one* in $B(s_a)$ is k. Note that for bit k, the corresponding transaction j contains the last itemset of s_a, and all the other itemsets of s_a must be contained in the transactions before j. One observation is that since $\{i\}$ is appended after the last itemset in s_a, it must be present in a transaction strictly after j in s_g. Therefore, bit k in $B(s_g)$ must have value *zero*. Another observation is that if item i is contained in any transaction after j, then the corresponding bit in $B(s_g)$ should be *one*. Therefore, all of the bits indexed after bit k in $B(s_g)$ should have the same value as the corresponding bit in $B(i)$. Our method to perform *S-step* is based on these two observations.

We first generate a bitmap from $B(s_a)$ such that all bits less than or equal to k are set to *zero*, and all bits after k are set to *one*. We call this bitmap a *transformed bitmap*. We then *AND* the transformed bitmap with the item bitmap. The resultant bitmap has the properties described above and it is exactly the bitmap for the generated sequence. In our implementation, the transformation is done using a lookup table.

Figure 4 shows an example of the *S-step* process. Let us consider the bitmap $B((\{a\}))$, and suppose we want to generate $B((\{a\},\{b\}))$. Since $(\{a\},\{b\})$ is a sequence-extended sequence of $(\{a\})$, we have to perform a *S-step* process on $B((\{a\}))$. Consider the section of the bitmap for the first customer. The first bit with value *one* is the first bit in the section. Hence, in the transformed bitmap $B((\{a\})_s)$, the first bit is set to *zero* and all the other bits for in that section are set to *one*. The resultant bitmap obtained by *ANDing* the transformed bitmap and $B(\{b\})$ is exactly $B((\{a\},\{b\}))$. In the final bitmap, the second bit and the third bit for the first customer have value *one*. This means that the last itemset (i.e. $\{b\}$) of the sequence appears in both transaction 2 and transaction 3, and itemset $\{a\}$ appears in transaction 1.

3.2.2 I-step Process

Suppose we are given $B(s_a)$ and $B(i)$, and that we want to perform an *I-step* on s_a using i. This *I-step* will generate a new sequence s_g by adding item i to the last itemset of s_a. The bitmap for s_g should have the property that if a bit has value *one*, then the corresponding transaction j must contain the last itemset in s_g, and all of the other itemsets in s_g must be in transactions before j.

Now, consider the resultant bitmap $B(s_r)$ obtained by *ANDing* $B(s_a)$ and $B(i)$. A bit k in $B(s_r)$ has value *one* if and only if bit k in $B(s_a)$ is *one* and bit k in $B(i)$ is also *one*. For bit k of $B(s_r)$ to be *one*, the transaction j that corresponds to bit k must, therefore, contain both the last itemset in s_a and the item i. In addition, all of the other itemsets of s_a must appear in transactions before j. It follows that $B(s_r)$ satisfies each of the requirements that $B(s_g)$ must satisfy; $B(s_r)$ is therefore exactly the bitmap

$(\{a\},\{b\})$		$\{d\}$		$(\{a\},\{b,d\})$
0		1		0
1		1		1
1		1		1
0		0		0
0		0	result	0
0	&	0	$\longrightarrow$	0
0		0		0
0		0		0
0		0		0
1		1		1
0		0		0
0		0		0

Figure 5: I-step processing on sequence bitmap $(\{a\},\{b\})$ shown in Figure 4

Symbol	Meaning
D	Number of customers in the dataset
C	Average number of transactions per customer
T	Average number of items per transaction
S	Average length of maximal sequences
I	Average length of transactions within the maximal sequences

Table 3: Parameters used in dataset generation

for the generated sequence.

Figure 5 shows an example of an *I-step* process. Let us consider $B((\{a\},\{b\}))$ from the previous section, and suppose we want to generate $B((\{a\},\{b,d\}))$. Since $(\{a\},\{b,d\})$ is an itemset-extended sequence of $(\{a\},\{b\})$, we have to perform an *I-step* process on $B((\{a\},\{b\}))$. The bitmap that results from the *ANDing* of $B((\{a\},\{b\}))$ and $B(\{d\})$ is exactly $B((\{a\},\{b,d\}))$. In the final bitmap, the second bit and the third bit for the first customer have value *one*, which means that the last itemset (i.e. $\{b,d\}$) of the sequence appears in both transaction 2 and transaction 3, and the other itemsets in the sequence (i.e. $\{a\}$) appear in transaction 1.

4. EXPERIMENTAL EVALUATION

In this section, we report our experimental results on the performance of *SPAM* in comparison with *SPADE* [12] and *PrefixSpan* [9]. We obtained the source code of *SPADE* from Mohammed Zaki, and an executable of *PrefixSpan* from Jiawei Han.

All the experiments were performed on a 1.7GHz Intel Pentium 4 PC machine with 1 gigabyte main memory, running Microsoft Windows 2000. All three algorithms are written in C++, and *SPAM* and *SPADE* were compiled using g++ in cygwin with compiler option `-O3`. During all of the tests, output of the frequent sequences was turned off so that the true running times of the algorithms were recorded. Our tests show that *SPAM* outperforms *SPADE* and *PrefixSpan* by up to an order of magnitude.

4.1 Synthetic data generation

To test our algorithm we generated numerous synthetic datasets using the IBM AssocGen program [2]. There are several factors that we considered while comparing *SPAM* against *SPADE*. These factors are listed in Table 3. All of these factors can be specified as parameters when running AssocGen. We also compared the performance of the algorithms as the minimum support was varied for several datasets of different sizes.

4.2 Comparison With SPADE and PrefixSpan

We compared *SPAM* with *SPADE* and *PrefixSpan* via two different methods. First, we compared the three algorithms on several small, medium, and large datasets for various minimum support values. This set of tests shows that *SPAM* outperforms *SPADE* by about a factor of 2.5 on small datasets and better than an order of magnitude for reasonably large datasets. *PrefixSpan* outperforms *SPAM* slightly on very small datasets, but on large datasets *SPAM* outperforms *PrefixSpan* by over an order of magnitude. The results of these tests are shown in Figures 6 to 11.

The primary reason that *SPAM* performs so well for large datasets is due to the bitmap representation of the data for efficient counting. The counting process is critical because it is performed many times at each recursive step, and *SPAM* handles it in an extremely efficient manner. For short datasets, the initial overhead needed to set up and use the bitmap representation in some cases outweighs the benefits of faster counting, and because of this *PrefixSpan* runs slightly faster for small datasets. As candidate sequences become longer, we recurse more levels down the tree and counting becomes more and more important. Overall, our runtime tests show that *SPAM* excels at finding the frequent sequences for many different types of large datasets.

Our second method of testing compared the performance of the algorithms as several parameters in the dataset generation were varied. We investigated the impact of different parameters of the data generation on the running time of each algorithm. The parameters that we varied were the number of customers in the dataset, the average number of transactions per customer, the average number of items per transaction, the average size of itemsets in the maximal sequences, and the average length of the maximal sequences. For each test, one parameter was varied and the others, including minimum support, were kept fixed. The results of these tests are shown in Figures 12 to 15.

Our experiments show that as the average number of items per transaction and the average number of transactions per customer increases, and as the average length of maximal sequences decreases, the performance of *SPAM* increases even further relative to the performance of *SPADE* and *PrefixSpan*. This performance increase is due to similar factors as in the case of increasing dataset size. While *SPAM* continues to outperform *SPADE* and *PrefixSpan* as the average size of the itemsets in the maximal sequences and the number of customers in the dataset increaseds the discrepancy between the running times did not increase as significantly as with the other parameters. The reasonfor the low inrease in performance is that an increase in these parameters does not make efficient counting more important as much as does the increase of the other parameters; however, *SPAM* is still much faster than *SPADE* and *PrefixSpan* in all cases with relatively large datasets.

4.3 Consideration of space requirements

Because *SPAM* uses a depth-first traversal of the search space, it is quite space-inefficient in comparison to *SPADE*. We can make an estimation of the blow-up in space require-

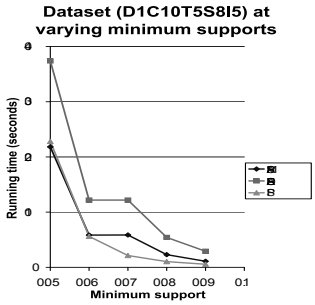

Figure 6: Varying support for small dataset #1

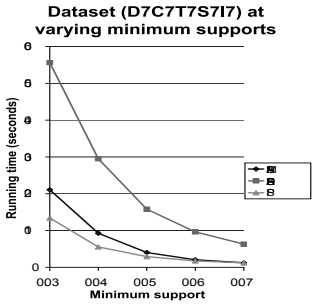

Figure 7: Varying support for small dataset #2

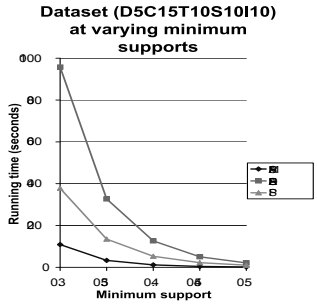

Figure 8: Varying support for medium-sized dataset #1

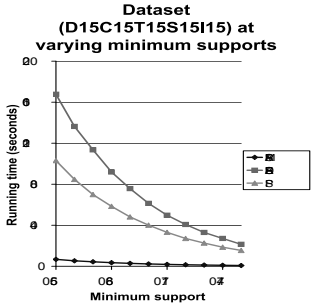

Figure 9: Varying support for medium-sized dataset #2

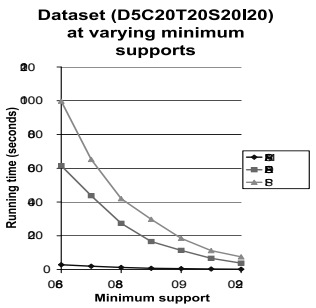

Figure 10: Varying support for large dataset #1

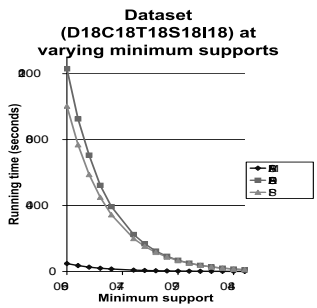

Figure 11: Varying support for large dataset #2

ments based on the parameters of an input dataset. Let D be the number of customers in the database, C the average number of transactions per customer, and N the total number of items across all of the transactions. Using the vertical bitmap representation to store transactional data, for each item we need one bit for transaction in the database. Since there are $D \times C$ transactions, *SPAM* requires $(D \times C \times N)/8$ bytes to store all of the data. Our representation is inefficient since even when an item is not present in a transaction, we are storing a *zero* to represent that fact.

The representation of the data that *SPADE* uses is more efficient. For each transaction, each item in that transaction must be recorded. Let us denote the average number of items per transaction as T. In that case, we need to record approximately $D \times C \times T$ items. Assuming it costs about 2 bytes to store a given item, *SPADE* uses about $D \times C \times T \times 2$ bytes.

Thus, *SPAM* is less space-efficient then *SPADE* as long as $16T < N$. In our sample datasets there are thousands of items and roughly 10 to 20 items per transaction. As a result, *SPADE* is expected to roughly outperform *SPAM* in terms of space requirements by factors ranging from 5 to 20, mirroring the performance increase of *SPAM*. Thus the choice between *SPADE* and *SPAM* is clearly a space-time tradeoff.

If space becomes a major issue, it is possible to compress the bitmaps that *SPAM* uses. Essentially, we can remove transactions from the bitmap for a sequence when those transactions will never contribute to the support count of the sequence. We are unable to include a full discussion of bitmap compression due to space constraints.

5. RELATED WORK

Agrawal and Srikant introduced the sequential pattern mining problem in [2]. Many methods, which are based the Apriori property [1], have been proposed for mining sequential patterns [2, 11, 9, 6, 4, 7, 8]. Apriori principle states that the fact that any superseqeuence of a non-frequent sequence must not be frequent. *SPADE* [12] and *PrefixSpan* [9] are two of the fastest algorithms that mine sequential patterns.

6. CONCLUSION

In this paper we presented an algorithm to quickly find all frequent sequences in a list of transactions. The algorithm utilizes a depth-first traversal of the search space combined with a vertical bitmap representation to store each sequence. Experimental results demonstrated that our algorithm outperforms *SPADE* and *PrefixSpan* on large datasets by over an order of magnitude. Our new algorithmic components including bitmap representation, *S-step/I-step* traversal, and *S-step/I-step* pruning all contribute to this excellent runtime.

Acknowledgements. We thank Mohammeed Zaki for providing us with the source code of Spade, and to Jiawei

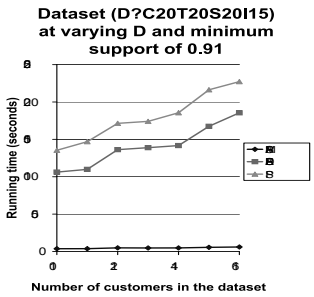

Figure 12: Varying number of customers

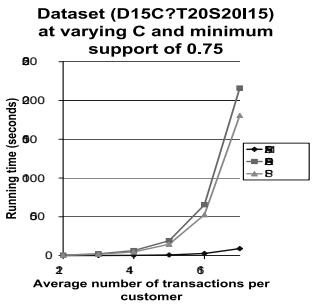

Figure 13: Varying number of transactions per customer

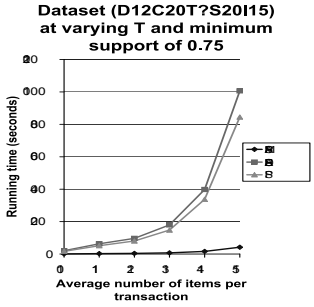

Figure 14: Varying number of items per transaction

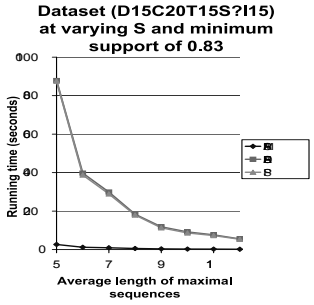

Figure 15: Varying average length of maximal sequences

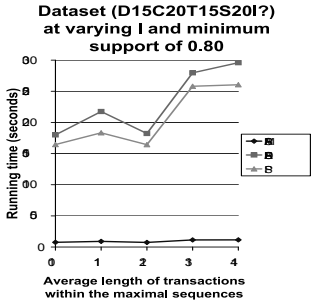

Figure 16: Varying average length of transactions within maximal sequences

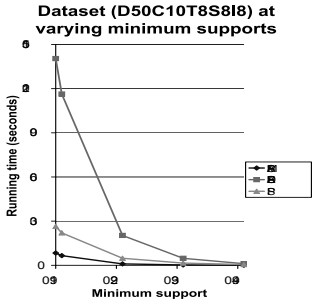

Figure 17: Varying support with large number of customers in dataset

Han for providing us with the executable of PrefixSpan. This work was supported in part by NSF grants IIS-0084762 and IIS-0121175, by a gift from Intel Corporation, and by the Cornell Intelligent Information Systems Institute.

7. REFERENCES

[1] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 487–499, Santiago, Chile, 1994.

[2] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *ICDE 1995*, Taipei, Taiwan, March 1995.

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

[4] C. Bettini, X. S. Wang, and S. Jajodia. Mining temporal relationships with multiple granularities in time sequences. *Data Engineering Bulletin*, 21(1):32–38, 1998.

[5] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE 2001*, Heidelberg, Germany, 2001.

[6] M. Garofalakis, R. Rastogi, and K. Shim. SPIRIT: Sequential pattern mining with regular expression constraints. In *VLDB 1999*, pages 223–234, San Francisco, Sept. 1999. Morgan Kaufmann.

[7] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *ICDE 1999*, pages 106–115, Sydney, Australia, Mar. 1999.

[8] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *KDD 1995*, pages 210–215, Montreal, Quebec, Canada, 1995.

[9] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan mining sequential patterns efficiently by prefix projected pattern growth. In *ICDE 2001*, pages 215–226, Heidelberg, Germany, Apr. 2001.

[10] R. Srikant and R. Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *EDBT 1996*, Avignon, France, March 1996.

[11] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *EDBT 1996*, pages 3–17, 25–29 Mar. 1996.

[12] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1/2):31–60, 2001.