PRISM: A Prime-Encoding Approach for Frequent Sequence Mining

Karam Gouda*, Mosab Hassaan*, Mohammed J. Zaki†
*Mathematics Dept., Faculty of Science, Benha, Egypt
†Department of Computer Science, RPI, Troy, NY, USA
karam_g@hotmail.com, mosab_ha@yahoo.com, zaki@cs.rpi.edu

Abstract

Sequence mining is one of the fundamental data mining tasks. In this paper we present a novel approach called PRISM*, for mining frequent sequences.* PRISM *utilizes a vertical approach for enumeration and support counting, based on the novel notion of* prime block encoding*, which in turn is based on prime factorization theory. Via an extensive evaluation on both synthetic and real datasets, we show that* PRISM *outperforms popular sequence mining methods like SPADE [10], PrefixSpan [6] and SPAM [2], by an order of magnitude or more.*

1 Introduction

Many real world applications, such as in bioinformatics, web mining, text mining and so on, have to deal with sequential/temporal data. Sequence mining helps to discover frequent sequential patterns across time or positions in a given data set. Mining frequent sequences is one of the basic exploratory mining tasks, and has attracted a lot of attention [1, 2, 4–6, 9, 10].

Problem Definition: The problem of mining sequential patterns can be stated as follows: Let $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ be a set of m distinct attributes, also called *items*. An *itemset* is a non-empty unordered collection of items (without loss of generality, we assume that items of an itemset are sorted in increasing order). A *sequence* is an ordered list of itemsets. An itemset i is denoted as $(i_1 i_2 \cdots i_k)$, where i_j is an item. An itemset with k items is called a *k-itemset*. A sequence S is denoted as $(s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_q)$, where each *element* s_j is an itemset. The number of itemsets in the sequence gives its *size* (q), and the total number of items in the sequence gives its *length* ($k = \sum_j |s_j|$). A sequence of length k is also called a *k-sequence*. For example, $(b \rightarrow ac)$ is a 3-sequence of size 2. A sequence $S = (s_1 \rightarrow \cdots \rightarrow s_n)$ is a *subsequence* of (or is *contained* in) another sequence $R = (r_1 \rightarrow \cdots \rightarrow r_m)$, denoted as $S \subseteq R$, if there exist integers $i_1 < i_2 < \cdots < i_n$ such that $s_j \subseteq r_{i_j}$ for all s_j. For example the sequence $(b \rightarrow ac)$ is a subsequence of $(ab \rightarrow e \rightarrow acd)$, but $(ab \rightarrow e)$ is not a subsequence of (abe), and vice versa.

Given a database $\mathcal{D}$ of sequences, each having a unique sequence identifier, and given some sequence $S = (s_1 \rightarrow \cdots \rightarrow s_n)$, the absolute *support* of S in D is defined as the total number of sequences in $\mathcal{D}$ that contain S, given as $sup(S, \mathcal{D}) = |\{S_i \in \mathcal{D} | S \subseteq S_i\}|$. The relative support of S is given as the fraction of database sequences that contain S. We use absolute and relative supports interchangeably. Given a user-specified threshold called the *minimum support* (denoted *minsup*), we say that a sequence is *frequent* if occurs more than *minsup* times. A frequent sequence is *maximal* if it is not a subsequence of any other frequent sequence. A frequent sequence is *closed* if it is not a subsequence of any other frequent sequence with the same support. Given a database $\mathcal{D}$ of sequences and *minsup*, the problem of mining sequential patterns is to find all frequent sequences in the database.

Related Work: The problem of mining sequential patterns was introduced in [1]. Many other approaches have followed since then [2, 4–10]. Sequence mining is essentially an enumeration problem over the sub-sequence partial order looking for those sequences that are frequent. The search can be performed in a breadth-first or depth-first manner, starting with more general (shorter) sequences and extending them towards more specific (longer) ones. The existing methods essentially differ in the data structures used to "index" the database to facilitate fast enumeration. The existing methods utilize three main approaches to sequence mining: horizontal [1, 5, 7], vertical [2, 4, 10] and projection-based [6, 9].

Our Contributions: In this paper we present a novel approach called PRISM (which stands for the bold letters in: **PRI**me-Encoding Based **S**equence **M**ining) for mining frequent sequences. PRISM utilizes a vertical approach for enumeration and support counting, based on the novel notion of *prime block encoding*, which in turn is based on prime factorization theory. Via an extensive evaluation on both synthetic and real datasets, we show that PRISM outperforms popular sequence mining methods like SPADE [10], PrefixSpan [6] and SPAM [2], by an order of magnitude or more.

2 Preliminary Concepts

Prime Factors & Generators: An integer p is a *prime integer* if $p > 1$ and the only positive divisors of p are 1 and p. Every positive integer n is either 1 or can be expressed as a product of prime integers, and this factorization is unique except for the order of the factors [3]. Let $p_1, p_2, \cdots, p_r$ be the *distinct* prime factors of n, arranged in order, so that $p_1 < p_2 < \cdots < p_r$. All repeated factors can be collected together and expressed using exponents, so that $n = p_1^{m_1} p_2^{m_2} \cdots p_r^{m_r}$, where each m_i is a positive integer, called the *multiplicity* of p_i, and this factor-

ization of n is called the *standard form* of n. For example, $n = 31752 = 2^3 \cdot 3^4 \cdot 7^2$.

Given two integers $a = \prod_{i=1}^{r_a} p_{ia}^{m_{ia}}$ and $b = \prod_{i=1}^{r_b} p_{ib}^{m_{ib}}$ in their standard forms, the *greatest common divisor* of the two numbers is given as $gcd(a, b) = \prod_i p_i^{m_i}$, where $p_i = p_{ja} = p_{kb}$ is a factor common to both a and b, and $m_i = \min(m_{ja}, m_{kb})$, with $1 \leq j \leq r_a$, $1 \leq k \leq r_b$. For example, if $a = 7056 = 2^4 \cdot 3^2 \cdot 7^2$ and $b = 18900 = 2^2 \cdot 3^3 \cdot 5^2 \cdot 7$, then $gcd(a, b) = 2^2 \cdot 3^2 \cdot 7 = 252$.

For our purposes, we are particularly interested in *square-free* integers n, defined as integers, whose prime factors p_i all have multiplicity $m_i = 1$ (note: the name square-free suggests that no multiplicity is 2 or more, i.e., the number does not contain a square of any factor). Given a set G, let $P(G)$ denote the set of all subsets of G. If we assume that G is ordered and indexed by the set $\{1, 2, \cdots, |G|\}$, then any subset $S \in P(G)$ can be represented as a $|G|$-length bit-vector (or binary vector), denoted $S^{\mathfrak{B}}$, whose i-th bit (from left) is 1 if the i-th element of G is in S, or else the i-th bit is 0. For example, if $G = \{2, 3, 5, 7\}$, and $S = \{2, 5\}$, then $S^{\mathfrak{B}} = 1010$.

Given a set $S \in P(G)$, we denote by $\otimes S$, the value obtained by applying the multiplication operator $\otimes$ to all members of S, i.e., $\otimes S = s_1 \cdot s_2 \cdot \ldots \cdot s_{|S|}$, with $s_i \in S$. If $S = \emptyset$, define $\otimes S = 1$. Let $\otimes P(G) = \{\otimes S : S \in P(G)\}$ be the set obtained by applying the multiplication operator on all sets in $P(G)$. In this case we also say that G is a *generator* of $\otimes P(G)$ under the multiplication operator.

We say that a set G is a *square-free generator* if each $X \in \otimes P(G)$ is square-free. In case a generator G consists of only prime integers, we call it a *prime generator*. Recall that a *semi-group* is a set that is closed under an associative binary operator $\otimes$. We say that a set P is a *square-free semi-group* iff for all $X, Y \in P$, if $Z = X \otimes Y$ is square-free, then $Z \in P$.

Theorem 2.1 *A set P is a square-free semi-group with operator $\otimes$ iff it has a square-free prime generator G. In other words, P is a square-free semi-group iff $P = \otimes P(G)$.*

As an example, let $G = \{2, 3, 5, 7\}$ be the set of the first four prime numbers. Then $\otimes P(G) = \{1, 2, 3, 5, 7, 6, 10, 14, 15, 21, 35, 30, 42, 70, 105, 210\}$. It is easy to see that G is a square-free generator of $\otimes P(G)$, which in turn is a square-free semi-group, since the product of any two of its elements that is square-free is already in the set.

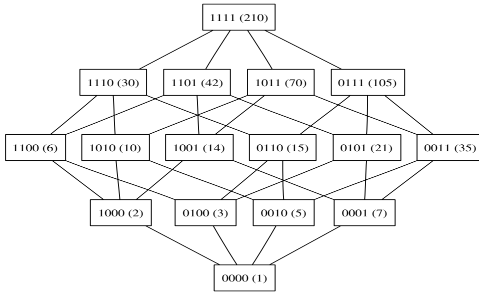


Figure 1. Lattice over $\otimes P(G)$. Each node shows a set $S \in P(G)$ using its bit-vector $S^{\mathfrak{B}}$ and the value obtained by multiplying its elements $\otimes S$.

The set $P(G)$ induces a *lattice* over the semi-group $\otimes P(G)$ as shown in Figure 1. In this lattice, the *meet operation* ($\wedge$) is set intersection over elements of $P(G)$, which corresponds to the gcd of the corresponding elements of $\otimes P(G)$. The *join operation* ($\vee$) is set union (over $P(G)$), which corresponds to the *least common multiple (lcm)* over $\otimes P(G)$. For example, $1010(10) \wedge 1001(14) = 1000(2)$, confirming that $gcd(10, 14) = 2$, and $1010(10) \vee 1001(14) = 1011(70)$, indicating that $lcm(10, 14) = 70$. More formally, we have:

Theorem 2.2 *Let $\otimes P(G)$ be a square-free semi-group with prime generator G, and let $X, Y \in \otimes P(G)$ be two distinct elements, then $gcd(X, Y) = \otimes(S_X \cap S_Y)$, and $lcm(X, Y) = \otimes(S_X \cup S_Y)$, where $X = \otimes S_X$ and $Y = \otimes S_Y$, and $S_X, S_Y \in P(G)$ are the prime factors of X and Y, respectively.*

Define the *factor-cardinality*, denoted $\|X\|_G$, for any $X \in \otimes P(G)$, as the number of prime factors from G in the factorization of X. Let $X = \otimes S_X$, with $S_X \subseteq G$. Then $\|X\|_G = |S_X|$. For example, $\|21\|_G = \{3, 7\} = 2$. Note that $\|\{1\}\|_G = 0$, since 1 has no prime factors in G.

Corollary 2.3 *Let $\otimes P(G)$ be a square-free semi-group with prime generator G, and let $X, Y \in \otimes P(G)$ be two distinct elements, then $gcd(X, Y) \in \otimes P(G)$.*

Prime Block Encoding: Let $\mathcal{T} = [1 : N] = \{1, 2, \ldots, N\}$ be the set of the first N positive integers, let G be a base set of prime numbers sorted in increasing order. Without loss of generality assume that N is a multiple of $|G|$, i,.e., $N = m \cdot |G|$. Let $B \in \{0, 1\}^N$ be a bit-vector of length N. Then B can partitioned into $m = \frac{N}{|G|}$ consecutive blocks, where each block $B_i = B\,[(i-1) \cdot |G| + 1 \;:\; i \cdot |G|]$, with $1 \leq i \leq m$. In fact, each $B_i \in \{0, 1\}^{|G|}$, is the indicator bit-vector $S^{\mathfrak{B}}$ representing some subset $S \subseteq G$. Let $B_i[j]$ denote the j-th bit in B_i, and let $G[j]$ denote the j-th prime in G. Define the *value* of B_i with respect to G as follows, $\nu(B_i, G) = \otimes\{G[j]^{B_i[j]}\}$. For example if $B_i = 1001$, and $G = \{2, 3, 5, 7\}$, then $\nu(B_i, G) = 2^1 \cdot 3^0 \cdot 5^0 \cdot 7^1 = 2 \cdot 7 = 14$. Note also that if $B_i = 0000$ then $\nu(B_i, G) = 1$.

Define $\nu(B, G) = \{\nu(B_i, G) : 1 \leq i \leq m\}$, as the *prime block encoding* of B with respect to the base prime set G. It should be clear that each $\nu(B_i, G) \in \otimes P(G)$. Note that when there is no ambiguity, we write $\nu(B_i, G)$ as $\nu(B_i)$, and $\nu(B, G)$ as $\nu(B)$. As an example, let $\mathcal{T} = \{1, 2, \ldots.., 12\}$, $G = \{2, 3, 5, 7\}$, and $B = 100111100100$. Then there are $m = 12/4 = 3$ blocks, $B_1 = 1001$, $B_2 = 1110$ and $B_3 = 0100$. We have $\nu(B_1) = \otimes S_G(B_1) = \otimes\{2, 7\} = 2 \cdot 7 = 14$, and the prime block encoding of B is given as $\nu(B) = \{14, 30, 3\}$. We also define the inverse operation $\nu^{-1}(\{14, 30, 3\}) = \nu^{-1}(14)\nu^{-1}(30)\nu^{-1}(3) = 100111100100 = B$. Also a bit-vector of all zeros (of any length) is denoted as $\mathbf{0}$, and its corresponding value/encoding is denoted as $\mathbf{1}$. For example, if $C = 00000000$, then we also write $C = \mathbf{0}$, and $\nu(C) = \{1, 1\} = \mathbf{1}$.

Let G be the base prime set, and let $A = A_1 A_2 \cdots A_m$, and $B = B_1 B_2 \cdots B_m$ be any two bit-vectors in $\{0, 1\}^N$, with $N = m \cdot |G|$, and $A_i, B_i \in \{0, 1\}^{|G|}$. Define $gcd(\nu(A), \nu(B)) = \{gcd(\nu(A_i), \nu(B_i)) : 1 \leq$

$i \leq m\}$. For example, for $\nu(B) = \{14, 30, 5\}$ and $\nu(A) = \{2, 210, 2\}$, we have $gcd(\nu(B), \nu(A)) = \{gcd(14, 2), gcd(30, 210), gcd(5, 2)\} = \{2, 30, 1\}$.

Let $A = A_1 A_2 \cdots A_m$ be a bit-vector of length N, where each A_i is a $|G|$ length bit-vector. Let $f_A = \arg\min_j\{A[j] = 1\}$ be the position of the first '1' in A, across all blocks A_i. Define a *masking* operator $(A)^{\triangleright}$ as follows:

$$(A)^{\triangleright}[j] = \begin{cases} 0, & j \leq f_A \\ 1, & j > f_A \end{cases}$$

In other words, $(A)^{\triangleright}$ is the bit vector obtained by setting $A[f_A] = 0$ and setting $A[j] = 1$ for all $j > f_A$. For example, if $A = 001001100100$, then $f_A = 3$, and $(A)^{\triangleright} = 000111111111$. Likewise, we can define the masking operator for a prime block encoding as follows: $(\nu(A))^{\triangleright} = \nu((A)^{\triangleright})$. For example, $(\nu(A))^{\triangleright} = \nu((001001100100)^{\triangleright}) = \nu(000111111111) = \nu(0001)\ \nu(1111)\ \nu(1111) = \{7, 210, 210\}$. In other words, $(\{5, 15, 3\})^{\triangleright} = \{7, 210, 210\}$, since $\nu(A) = \nu(001001100100) = \nu(0010)\ \nu(0110)\ \nu(0100) = \{5, 15, 3\}$.

3 The PRISM Algorithm

Sequence mining involves a combinatorial enumeration or search for frequent sequences over the sequence partial order. There are three key aspects of PRISM that need elucidation: i) the search space traversal strategy, ii) the data structures used to represent the database and intermediate candidate information, and iii) how support counting is done for candidates. PRISM uses the prime block encoding approach to represent candidates sequences, and uses join operations over the prime blocks to determine the frequency for each candidate.

Search Space: The partial order induced by the subsequence relation is typically represented as a search tree, defined recursively as follows: The root of the tree is at level zero and is labeled with the null sequence $\emptyset$. A node labeled with sequence S at level k, i.e., a k-sequence, is repeatedly extended by adding one item from $\mathcal{I}$ to generate a child node at the next level $(k+1)$, i.e., a $(k+1)$-sequence. There are two ways to extend a sequence by an item: *sequence extension* and *itemset extension*. In a sequence extension, the item is appended to the sequential pattern as a new itemset. In an itemset extension, the item is added to the last itemset in the pattern, provided that the item is lexicographically greater than all items in the last itemset. Thus, a sequence-extension always increases the size of the sequence, whereas, an itemset-extension does not. For example, if we have a node $S = ab \rightarrow a$ and an item b for extending S, then $ab \rightarrow a \rightarrow b$ is a sequence-extension, and $ab \rightarrow ab$ is an itemset extension.

Prime Block Encoding: Consider the example database in Figure 2(a), consisting of 5 sequences over the items $\mathcal{I} = \{a, b, c\}$. Let $G = \{2, 3, 5, 7\}$ be the base square-free prime generator set. Let's see how PRISM constructs the prime block encoding for a single item a. In the first step, PRISM constructs the prime encoding of the positions within each sequence. For example, since a occurs in positions 1,4, and 6 (assuming positions/indexes starting at 1) in sequence 1, we obtain the bit-encoding of a's occurrences: 100101. PRISM next pads this bit-vector so that it is a multiple of $|G| = 4$, to obtain $A = 100101\mathbf{00}$ (note: bold bits denote padding). Next we compute $\nu(A) = \nu(1001)\nu(0100) = \{14, 3\}$. The position encoding for a over all the sequences is shown in Figure 2 (b).

sid	database sequence
1	$ab \rightarrow b \rightarrow b \rightarrow ab \rightarrow b \rightarrow a$
2	$ab \rightarrow b \rightarrow b$
3	$b \rightarrow ab$
4	$b \rightarrow b \rightarrow b$
5	$ab \rightarrow ab \rightarrow ab \rightarrow a \rightarrow bc$

(a)

sid	Bit-encoded pos	Prime-encoded pos
1	1001,01**00**	{14, 3}
2	100**0**	{2}
3	01**00**	{3}
4	000**0**	{1}
5	1111,0**000**	{210,1}

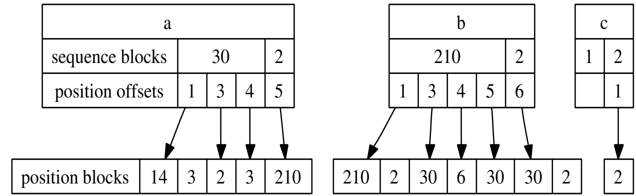
(b)

Bit-encoded sid	Prime-encoded sid
1110,1**000**	{30,2}

(c)

Item	Sequence Blocks	Position Blocks
a	{30,2}	{14, 3}, {2}, {3}, {1}, {210,1}
b	{210,2}	{210, 2}, {30}, {6}, {30}, {30,2}
c	{1,2}	{1, 1}, {1}, {1}, {1}, {1,2}

(d)

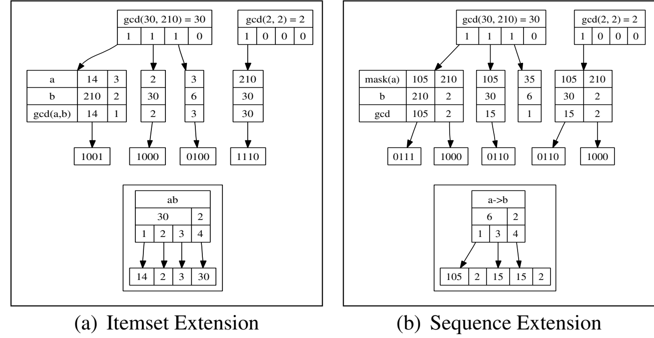


(e)

Figure 2. Example of Prime Block Encoding: (a) Example Database. (b) Position Encoding for a. (c) Sequence Encoding for a. (d) Full Prime Blocks for a, b and c. (e) Prime Block Encoding for a, b and c.

PRISM next computes the prime encoding for the sequence ids. Since a occurs in all sequences, except for 4, we can represent a's sequence occurrences as a bit-vector $A = 11101\mathbf{000}$ after padding. This yields the prime encoding shown in Figure 2(c), since $\nu(A) = \nu(1110)\nu(1000) = \{30, 2\}$. The full prime encoding for item a consists of all the sequence and position blocks, as shown in Figure 2(d). A block $A_i = 0000 = \mathbf{0}$, with $\nu(A_i) = \{1\} = \mathbf{1}$, is also called an *empty* block. Note that the full encoding retains all the empty position blocks, for example, a does not occur in the second position block in sequence 5, and thus its bit-vector is 0000, and the prime code is $\{1\}$. In general, since items are expected to be sparse, there may be many blocks within a sequence where an item does not appear.

To eliminate those empty blocks, PRISM retains only the non-empty blocks in the prime encoding. To do this it needs



(a) Itemset Extension (b) Sequence Extension

Figure 3. Extensions via Prime Block Joins

to keep an index with each sequence block to indicate which non-empty position blocks correspond to a given sequence block. Figure 2 (e) shows the actual (compact) prime block encoding for item a. The first sequence block is 30, with factor-cardinality $\|30\|_G = 3$, which means that there are 3 valid (i.e., with non-empty position blocks) sequences in this block, and for each of these, we store the offsets into the position blocks. For example, the offset of sequence 1 is 1, with the first two position blocks corresponding to this sequence. Thus the offset for sequence 2 is 3, with only one position block, and finally, the offset of sequence 3 is 4. Note that the sequences which represent the sequence block 30, can be found directly from the corresponding bit-vector $\nu^{-1}(30) = 1110$, which indicates that sequence 4 is not valid. The second sequence block for a is 2 (corresponding to $\nu^{-1}(2) = 1000$), indicating that only sequence 5 is valid, and its position blocks begin as position 5. The benefit of this sparse representation becomes clear when we consider the prime encoding for c. Its full encoding (see Figure 2(d)) contains a lot of redundant information, which has been eliminated in the compact prime block encoding (see Figure 2(e)).

It is worth noting that the support of a sequence S can be directly determined from its sequence blocks in the prime block encoding. Let $\mathcal{E}(S) = (\mathcal{S}_S, \mathcal{P}_S)$ denote the prime block encoding for sequence S, where $\mathcal{S}_S$ is the set of all encoded sequence blocks, and $\mathcal{P}_S$ is the set of all encoded position blocks for S. The support of a sequence S with prime block encoding $\mathcal{E}(S) = (\mathcal{S}_S, \mathcal{P}_S)$ is given as $sup(S) = \sum_{v_i \in \mathcal{S}_S} \|v_i\|_G$. For example, for $S = a$, since $\mathcal{S}_a = \{30, 2\}$, we have $sup(a) = \|30\|_G + \|2\|_G = 3 + 1 = 4$. Given a list of full or compact position blocks $\mathcal{P}_S$ for a sequence S, we use the notation $\mathcal{P}_S^i$ to denote those positions blocks, which come from sequence id i. For example, in $\mathcal{P}_a^1 = \{14, 3\}$. In the full encoding $\mathcal{P}_a^5 = \{210, 1\}$, but in the compact encoding $\mathcal{P}_a^5 = \{210\}$ (see Figure 2(d)-(e)).

Support Counting via Prime Block Joins: The frequent sequence enumeration process starts with the root of the search tree as the prefix node $P = \emptyset$, and PRISM assumes that initially we know the prime block encodings for all single items. PRISM then recursively extends each node in the search tree, computes the support via the prime block joins, and retains new candidates (or extensions) only if they are frequent. The search is essentially depth-first, the main difference being that for any node S, all of its extensions are evaluated before the depth-first recursive call. When there are no new frequent extensions found, the search stops. To complete the description, we now detail the prime block join operations. We will illustrate the prime block itemset and sequence joins using the prime encodings $\mathcal{E}(a)$ and $\mathcal{E}(b)$, for items a and b, respectively, as shown in Figure 2(e).

Itemset Extensions: Let's first consider how to obtain the prime block encoding for the itemset extension $\mathcal{E}(ab)$, which is illustrated in Figure 3(a). Note that the sequence blocks $\mathcal{S}_a = \{30, 2\}$ and $\mathcal{S}_b = \{210, 2\}$ contain all information about the relevant sequence ids where a and b occur, respectively. To find the sequence block for itemset extension ab, we simply have to compute the gcd for the corresponding elements from the two sequence blocks, namely $gcd(30, 210) = 30$ (which corresponds to the bit-vector 1110), and $gcd(2, 2) = 2$ (which corresponds to bit-vector 1000). We say that a sequence id $i \in gcd(\mathcal{S}_a, \mathcal{S}_b)$ if the i-th bit is set in the bit vector $\nu^{-1}(gcd(\mathcal{S}_a, \mathcal{S}_b))$. Since $\nu^{-1}(gcd(\mathcal{S}_a, \mathcal{S}_b)) = \nu^{-1}(\{30, 2\}, \{210, 2\}) = \nu^{-1}(30, 2) = 11101000$, we find that sids 1, 2, 3 and 5 are the ones that contain occurrences of both a and b.

All that remains to be done is to determine, by looking at the position blocks, if a and b, in fact, occur simultaneously at some position in those sequences. Let's consider each sequence separately. Looking at sequence 1, we find in Figure 2(e) that its positions blocks are $\mathcal{P}_a^1 = \{14, 3\}$ in $\mathcal{E}(a)$ and $\mathcal{P}_b^1 = \{210, 2\}$ in $\mathcal{E}(b)$. To find where a and b co-occur in sequence 1, all we have to do is compute the gcd of these position blocks to obtain $gcb(a, b) = \{gcd(14, 210), gcd(3, 2)\} = \{14, 1\}$, which indicates that ab only occur at positions 1 and 4 in sequence 1 (since $\nu^{-1}(14) = 1001$). A quick look at Figure 2(a) confirms that this is indeed correct. If we continue in like manner for the remaining sequences (2, 3 and 5), we obtain the results shown in Figure 3(a), which also shows the final prime block encoding $\mathcal{E}(ab)$. Note that there is at least one non-empty block for each of the sequences, even though for sequence 1, the second position block is discarded in the final prime encoding. Thus $sup(ab) = \|30\|_G + \|2\|_G = 3 + 1 = 4$.

Sequence Extensions: Let's consider how to obtain the prime block encoding for the sequence extension $\mathcal{E}(a \rightarrow b)$, which is illustrated in Figure 3(b). The first step involves computing the gcd for the sequence blocks as before, which yields sequences 1, 2, 3 and 5, as those which may potentially contain the sequence $a \rightarrow b$.

For sequence 1, we have the positions blocks $\mathcal{P}_a^1 = \{14, 3\}$ for a and $\mathcal{P}_b^1 = \{210, 2\}$ for b. The key difference with the itemset extension is the way in which we process each sequence. Instead of computing $gcd(\{14, 3\}, \{210, 2\})$, we compute $gcd((\{14, 3\})^{\triangleright}, \{210, 2\}) = gcd(\{105, 210\}, \{210, 2\}) = \{gcd(105, 210), gcd(210, 2)\} = \{105, 2\}$. Note that $\nu^{-1}(\{105, 2\}) = 01111000$, which precisely indicate those positions in sequence 1, where b occurs after an a. Thus, sequence joins always keep track of the positions of the last item in the sequence. Proceeding in like manner for sequences 2,3, and 5, we obtain the results shown in Figure 3(b). Note that for sequence 3, even though it contains both items a and b, b never occurs after an a, and thus sequence 3 does not contribute to the support of $a \rightarrow b$. This is also confirmed by computing $gcd((3)^{\triangleright}, 6) = gcd(35, 6) =$

1, which leads to an empty block ($\nu^{-1}(1) = 0000$). Thus in the compact prime encoding of $\mathcal{E}(a \rightarrow b)$, sequence 3 drops out. The remaining sequences 1, 2, and 5, contribute at least one non-empty block, which yields $\mathcal{S}_{a\rightarrow b} = \nu(11001000) = \{6, 2\}$, as shown in Figure 3(b), with support $sup(a \rightarrow b) = \|6\|_G + \|2\|_G = 2 + 1 = 3$.

Optimizations: Since computing the gcd is one of main operations in PRISM, we use a pre-computed table called GCD to facilitate rapid gcd computations. Note that in our examples above, we used only the first four primes as the base generator set G. However, in our actual implementation, we used $|G| = 8$ primes as the generator set, i.e., $G = \{2, 3, 5, 7, 11, 13, 17, 19\}$. Thus each block size is now 8 instead of 4. Note that with the new G, the largest element in $\otimes P(G)$ is $\otimes G = 9699690$. In total there are $|\otimes P(G)| = 256$ possible elements in semi-group $\otimes P(G)$.

In a naive implementation, the GCD lookup table can be stored as a two-dimensional array with cardinality 9699690×9699690, where $GCD(i, j) = gcd(i, j)$ for any two integers $i, j \in [1 : 9699690]$. This is clearly grossly inefficient, since there are in fact only 256 distinct (square-free) products in $\otimes P(G)$, and we thus really need a table of size 256×256 to store all the gcd values. We achieve this by representing each element in $\otimes P(G)$ by its rank, as opposed to its value.

Let $S \in P(G)$, and let $S^{\mathfrak{B}}$ its $|G|$-length indicator bit-vector, whose i-th bit is '1' iff the i-element of G is in S. Then the rank of $\otimes S$ is equal to the decimal value of $S^{\mathfrak{B}}$ (with the left-most bit being the least significant bit). In other words $rank(\otimes S) = decimal(S^{\mathfrak{B}})$. For example, the $rank(1) = decimal(00000000) = 0$, $rank(13) = decimal(00000100) = 32$, $rank(35) = decimal(00110000) = 12$, and $rank(9699690) = decimal(11111111) = 255$. Let $S, T \in P(G)$, and let $S^{\mathfrak{B}}, T^{\mathfrak{B}}$ be their indicator bit-vectors with respect to generator set G. Then $rank(gcd(\otimes S, \otimes T)) = decimal(S^{\mathfrak{B}} \wedge T^{\mathfrak{B}})$. Consider for example, $gcd(35, 6) = 1$. We have $rank(gcd(35, 6)) = decimal(00110000 \wedge 11000000) = decimal(00000000) = 0$, which matches the computation $rank(gcd(35, 6)) = rank(1) = 0$. Instead of using direct values, all gcd computations are performed in terms of the ranks of the corresponding elements. Thus each cell in the GCD table stores: $GCD(rank(i), rank(j)) = rank(gcd(i, j))$, where $i, j \in \otimes P(G)$. This brings down the storage requirements of the GCD table to just $256 \times 256 = 65536$ bytes, since each $rank$ requires only one byte of memory (since $rank \in [0 : 255]$).

Once the final sequence blocks are computed for after a join operation, we need to determine the actual support, by adding the factor cardinalities for each sequence block. To speed up this support determination, PRISM maintains a one-dimensional look-up array called $CARD$ to store the *factor-cardinality* for each element in the set $\otimes P(G)$. That is we store $CARD(rank(x)) = \|x\|_G$ for all $x \in \otimes P(G)$. For example, since $\|35\|_G = 2$, we have $CARD(rank(35)) = CARD(12) = 2$.

Furthermore, in sequence block joins, we need to compute the masking operation for each position block. For this PRISM maintains another one dimensional array called $MASK$, where $MASK(rank(x)) = rank((x)^{\triangleright})$ for each $x \in \otimes P(G)$. For example $MASK(rank(2)) = rank((2)^{\triangleright}) = rank(4849845) = 254$. Finally, as an optimization for fast joins, once we determine gcd_{XY} or $gcd_{X\rightarrow Y}$ in the prime itemset/sequence block joins, if the number of supporting sequences is less than *minsup*, we can stop further processing of position blocks, since the resulting extensions cannot be frequent in this case.

4 Experiments

In this section we study the performance of PRISM by varying different database parameters and by comparing it with other state-of-the-art sequence mining algorithms like SPADE [10], PrefixSpan [6] and SPAM [2]. The codes/executables for these methods were obtained from their authors. All experiments were performed on a laptop with 2.4GHz Intel Celeron processor, and with 512MB memory, running Linux.

Synthetic and Real Datasets: We used several synthetic datasets, generated using the approach outlined in [1]. The datasets are generated using the following process. First N_I maximal itemsets of average size I are generated by choosing from N items. Then N_S maximal sequences of average size S are created by assigning itemsets from N_I to each sequence. Next a customer (or input sequence) of average C transactions (or itemsets) is created, and sequences in N_S are assigned to different customer elements, respecting the average transaction size of T. The generation stops when D input-sequences have been generated. For example, the dataset `C20T50S20I10N1kD100k`, means that it has D=100k sequences, with C=20 average transactions, T=50 average transaction size, chosen from a pool with average sequence size S=20 and average transaction size I=10, with N=1k different items. The default itemset and sequence pool sizes are always set to $N_S = 5000$ and $N_I = 25000$, respectively.

We also compared the methods on two real datasets taken from [9]. `Gazelle` was part of the KDD Cup 2000 challenge dataset. It contains log data from a (defunct) web retailer. It has 59602 sequences, with an average length of 2.5, length range of [1, 267], and 497 distinct items. The `Protein` dataset contains 116142 proteins sequences downloaded from the Entrez database at NCBI/NIH. The average sequence length is 482, with length range of [400,600], and 24 distinct items (the different amino acids).

Performance Comparison: Figure 4 shows the performance comparison of the four algorithms, namely, SPAM [2], PrefixSpan [6], SPADE [10] and PRISM, on different synthetic and real datasets, with varying minimum support. As noted earlier, for PRISM we used the first 8 primes as the base prime generator set G. Figure 4 (a)-(b) show (small) datasets where all four methods can run for at least some support values. For these datasets, we find that PRISM has the best overall performance. For the support values where SPAM can run, it is generally in the second spot (in fact, it is the fastest on `C10T20S4I4N0.1kD10k`). However, SPAM fails to run for lower support values. PRISM outperforms SPADE by about 4 times, and PrefixSpan by over an order of magnitude.

Figure 4 (c)-(d) show larger datasets (with D=100k sequences). On these SPAM could not run on our laptop, and

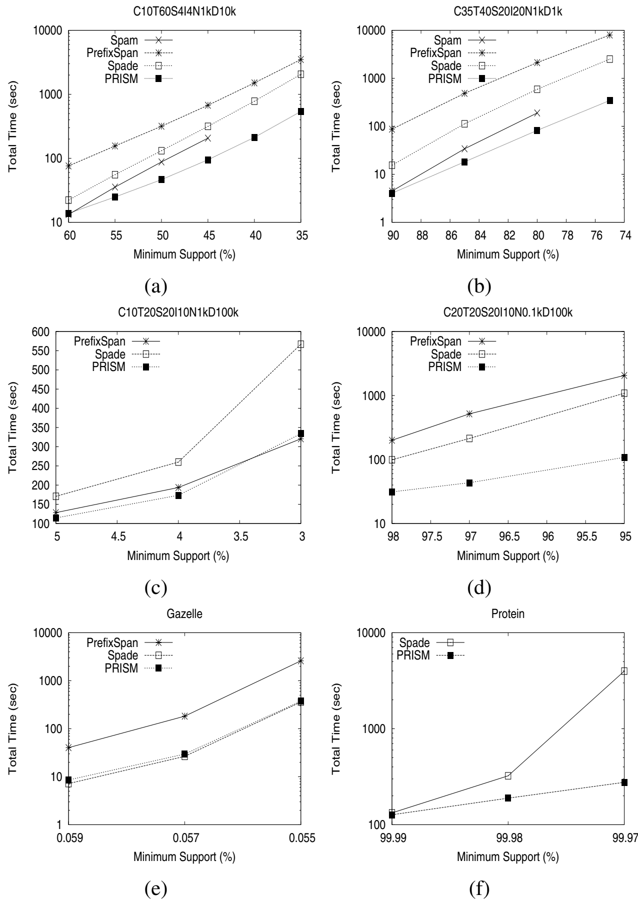
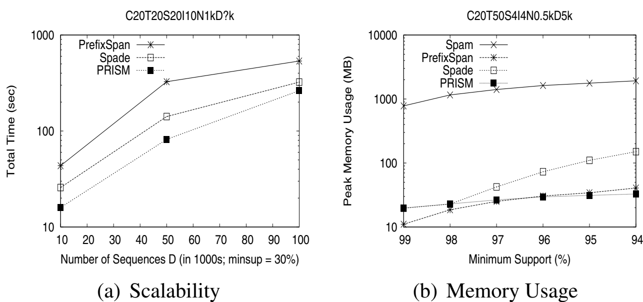


Figure 4. Performance Comparison



(a) Scalability (b) Memory Usage

Figure 5. Scalability & Memory Consumption

thus is not shown. PRISM again outperforms PrefixSpan and SPADE, by up to an order of magnitude. Finally, Figure 4 (e)-(f) show the performance comparison on the real datasets, `Gazelle` and `Protein`. SPAM failed to run on both these datasets on our laptop, and PrefixSpan did not run on `Protein`. On `Gazelle` PRISM is an order of magnitude faster than PrefixSpan, but is comparable to SPADE. On `Protein` PRISM outperforms SPADE by an order of magnitude (for lower support).

Based on these results on diverse datasets, we can observe some general trends. Across the board, our new approach, PRISM, is the fastest (with a few exceptions), and runs for lower support values than competing methods. SPAM generally works only for smaller datasets due to its very high memory consumption (see below); when it runs, SPAM is generally the second best. SPADE and PrefixSpan do not suffer from the same problems as SPAM, but they are much slower than PRISM, or they fail to run for lower support values, when the database parameters are large.

Scalability: Figure 5(a) shows the scalability of the different methods when we vary the number of sequences 10k to 100k (using as base values: C=20, T=20, S=20, I=10, and N=1k). Since SPAM failed to run on these larger datasets, we could not report on its scalability. We find that the effect of increasing the number of sequences is approximately linear.

Memory Usage: Figure 5(b) shows the memory consumption of the four methods on a sample of the datasets. The figures plot the peak memory consumption during execution (measured using the `memusage` command in Linux). Figure 5(b) quickly demonstrates why SPAM is not able to run on all except very small datasets. We find that its memory consumption is well beyond the physical memory available (512MB), and thus the program aborts when the operating system runs out of memory. We can also note that SPADE generally has a 3-5 times higher memory consumption than PrefixSpan and PRISM. The latter two have comparable and very low memory requirements.

Conclusion: Based on the extensive experimental comparison with popular sequence mining methods, we conclude that, across the board, PRISM is one of the most efficient methods for frequent sequence mining. It outperforms existing methods by an order of magnitude or more, and has a very low memory footprint. It also has good scalability with respect to a number of database parameters. Future work will consider the tasks of mining all the closed and maximal frequent sequences, as well as the task of pushing constraints within the mining process to make the method suitable for domain-specific sequence mining tasks. For example, allowing approximate matches, allowing substitution costs, and so on.

References

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *11th ICDE Conf.*, 1995.

[2] J. Ayres, J. E. Gehrke, T. Yiu, and J. Flannick. Sequential pattern mining using bitmaps. In *SIGKDD Conf.*, 2002.

[3] J. Gilbert and L. Gilbert. *Elements of Modern Algebra*. PWS Publishing Co., 1995.

[4] H. Mannila, H. Toivonen, and I. Verkamo. Discovering frequent episodes in sequences. In *SIGKDD Conf.*, 1995.

[5] F. Masseglia, F. Cathala, and P. Poncelet. The PSP approach for mining sequential patterns. In *European PKDD Conf.*, 1998.

[6] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Prefixspan: Mining sequential patterns efficiently by prefixprojected pattern growth. In *ICDE Conf.*, 2001.

[7] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Intl. Conf. Extending Database Technology*, 1996.

[8] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *ICDE Conf.*, 2004.

[9] Z. Yang, Y. Wang, and M. Kitsuregawa. Effective sequential pattern mining algorithms for dense database. In *Japanese Data Engineering Workshop (DEWS)*, 2006.

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