Volume 76 Issue 1 February 2010 ISSN 0022-0000
ELSEVIER
JOURNAL OF COMPUTER AND SYSTEM SCIENCES
Special Issue on Intelligent Data Analysis
Guest Editors
Radim Belohlavek
Rudolf Kruse
Managing Editor
E. K. Blum
Editors
Alfred V. Aho
Arthur W. Burks
Ronald Fagin
J. Hartmanis
John Hopcroft
R. M. Karp
D. E. Knuth
Raymond E. Miller
Christos Papadimitriou
Michael Rabin
Hartley Rogers, Jr.
Dana Scott
J. D. Ullman
L. A. Zadeh
Available online at
ScienceDirect
www.sciencedirect.com

Journal of Computer and System Sciences 76 (2010) 88–102

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Prism: An effective approach for frequent sequence mining via prime-block encoding[☆]

Karam Gouda[a], Mosab Hassaan[a], Mohammed J. Zaki[b,*]

[a] Mathematics Dept., Faculty of Science, Benha, Egypt
[b] Computer Science Dept., Rensselaer Polytechnic Institute, Troy, NY, USA

ARTICLE INFO

Article history:
Received 30 September 2007
Received in revised form 4 June 2008
Available online 23 May 2009

Keywords:
Frequent sequence mining
Prime encoding
Data mining

ABSTRACT

Sequence mining is one of the fundamental data mining tasks. In this paper we present a novel approach for mining frequent sequences, called Prism. It utilizes a vertical approach for enumeration and support counting, based on the novel notion of *primal 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 [M.J. Zaki, SPADE: An efficient algorithm for mining frequent sequences, Mach. Learn. J. 42 (1/2) (Jan/Feb 2001) 31–60], PrefixSpan [J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, M.-C. Hsu, PrefixSpan: Mining sequential patterns efficiently by prefixprojected pattern growth, in: Int'l Conf. Data Engineering, April 2001] and SPAM [J. Ayres, J.E. Gehrke, T. Yiu, J. Flannick, Sequential pattern mining using bitmaps, in: SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining, July 2002], 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 [1,2,8–10,13,14].

Problem definition. The problem of mining sequential patterns can be stated as follows: Let $\mathcal{I} = \{i_1, i_2, \dots, i_m\}$ 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 $(s_1 \to s_2 \to \cdots \to s_q)$, where the sequence *element* s_j is an itemset. The number of itemsets (or elements) in the sequence gives its *size*, 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, $(b \to ac)$ is a 3-sequence of size 2.

A sequence $S = (s_1 \to \cdots \to s_n)$ is a *subsequence* of another sequence $R = (r_1 \to \cdots \to r_m)$, 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 $(b \to ac)$ is a subsequence of $(ab \to e \to acd)$, but $(ab \to e)$ is not a subsequence of (abe), and vice versa. If $S \subseteq R$, we also say that R *contains* S.

☆ This work was supported in part by NSF CAREER Award IIS-0092978, and NSF grants EIA-0103708 and EMT-0432098.
* Corresponding author.
E-mail address: zaki@cs.rpi.edu (M.J. Zaki).

0022-0000/$ – see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcss.2009.05.008

Given a database $\mathcal{D}$ of sequences, each having a unique sequence identifier, and given some sequence $S = (s_1 \to \cdots \to s_n)$, the absolute *support* of S in $\mathcal{D}$ is defined as the total number of sequences in $\mathcal{D}$ that contain S, given as $\sup(S, \mathcal{D}) = |\{S_i \in \mathcal{D} \mid S \subseteq S_i\}|$. 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 [2,8–11,13,14]. Methods for mining closed sequences appear in [3,12]. 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 method*: These are exemplified by the GSP method [11]. Here the database sequences are not indexed at all, whereas the candidate frequent patterns are stored in some hash-based/prefix-based structure. Pattern frequency is determined by directly checking which candidate sequences appear in which database sequences. The methods in [1,9] also follow a horizontal approach.
- *Vertical method*: The prototype method here is SPADE [14]. Here each item/symbol is associated with an inverted index, called tidlist, which lists the sequence ids, and the positions where the symbol occurs. Frequency of a pattern is determined by performing temporal joins on these tidlists. SPAM [2] is also a vertical approach, that uses bit-vectors to represent the tidlists. The episode mining method in [8] is a variant of the vertical approach tailored to mining a single long sequence, as opposed to many sequences.
- *Projection method*: PrefixSpan [10] follows a database projection approach, which is a hybrid between the horizontal and vertical approaches. Given any prefix sequence P, the main idea is to project the horizontal database, so that the projected (or conditional) database contains only those sequences that have prefix P. The frequency of extensions of P can be directly counted in the projected database. Via recursive projections all frequent sequences can be enumerated. PrefixSpan is a hybrid method, since the projected database is equivalent to a horizontal representation of the tidlists of sequences that share a given prefix P. LAPIN [13] is another method that applies the so-called last-position induction optimization with a projection approach.

Domain specific sequence mining methods, such as in bioinformatics, have a rich literature [6,7]. Many of these techniques are geared towards finding consecutive sequences, whereas, we are interested in more unconstrained sequences. Furthermore, sequential patterns allow each sequence element to be an itemset, whereas most biosequence methods allow only the base symbols as sequence elements. On the other hand, biosequence mining methods allow other variations, such as approximate matching and symbol substitution matrices, which we do not consider here.

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. Note that a preliminary short paper on PRISM appeared in [5]. PRISM utilizes a vertical approach for enumeration and support counting, based on the novel notion of *primal 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 [14], PrefixSpan [10] and SPAM [2], by an order of magnitude or more.

2. Preliminary concepts

2.1. Prime factors & generators

An integer p is a *prime integer* if $p > 1$ and the only positive divisors of p are 1 and p.

Theorem 2.1 *(Unique factorization theorem). (See [4].) 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.*

Let $p_1, p_2, \ldots, p_r$ 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 $n = p_1^{m_1} \times p_2^{m_2} \times \cdots \times p_r^{m_r}$, where each m_i is a positive integer, called the *multiplicity* of p_i, and this factorization of n is called the *standard form* of n. For example, $n = 31\,752 = 2^3 \times 3^4 \times 7^2$.

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 $m_i = \min(m_{ja}, m_{kb})$,

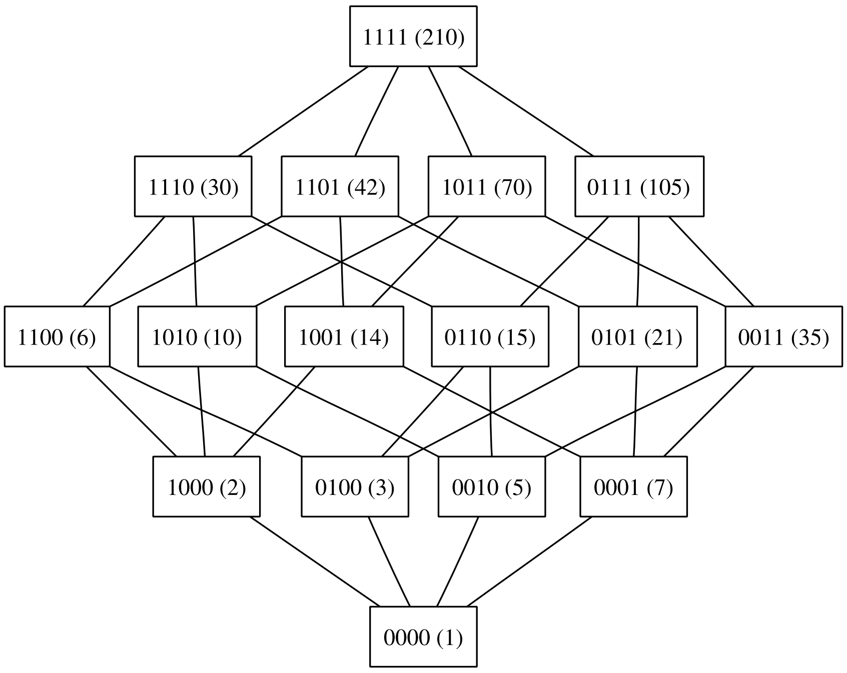


Fig. 1. Lattice over $\bigotimes P(G)$. Each node shows a set $S \in P(G)$ using its bit-vector $S^{\mathfrak{B}}$ and the value obtained by multiplying its elements $\bigotimes S$.

with $1 \leqslant j \leqslant r_a$, $1 \leqslant k \leqslant r_b$. For example, if $a = 7056 = 2^4 \times 3^2 \times 7^2$ and $b = 18\,900 = 2^2 \times 3^3 \times 5^2 \times 7$, then $gcd(a,b) = 2^2 \times 3^2 \times 7 = 252$.

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 $\{1, 2, \ldots, |G|\}$, then any subset $S \in P(G)$ can be represented as a $|G|$-length bit-vector (or binary vector), denoted $S^{\mathfrak{B}}$, whose ith bit (from left) is 1 if the ith element of G is in S, or else the ith 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 define a *multiplication operator* $\otimes$ on S, as follows: $\bigotimes S = s_1 \times s_2 \times \cdots \times s_{|S|}$, with $s_i \in S$. If $S = \emptyset$, define $\bigotimes S = 1$. In other words, $\bigotimes S$ is the value obtained by multiplying all members of S. Further, define $\bigotimes P(G) = \{\bigotimes S\colon\ S \in P(G)\}$ to 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 $\bigotimes P(G)$ under the multiplication operator.

We say that a set G is a *square-free generator* if each $X \in \bigotimes P(G)$ 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* (under operator $\otimes$) iff for all $X, Y \in P$, if $Z = X \otimes Y$ is square-free, then $Z \in P$.

Theorem 2.2. *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 $P = \bigotimes P(G)$.*

Proof. If P is a square-free semi-group, then we choose as its generator set G, the distinct prime factors over all elements of P. On the other hand, if G is a square-free prime generator, then $\bigotimes P(G)$ is a square-free semi-group. It follows that $P = \bigotimes P(G)$. □

As an example, let $G = \{2, 3, 5, 7\}$ be the set of the first four prime numbers. Then $\bigotimes P(G) = \{1, 2, 3, 5, 7, 6, 10, 14, 15, 21, 35, 30, 42, 70, 105, 210\}$. It is easy to see that G is a square-free generator of $\bigotimes P(G)$, 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.

The set $P(G)$ induces a *lattice* over the semi-group $\bigotimes P(G)$ as shown in Fig. 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 $\bigotimes P(G)$. The *join operation* ($\vee$) is set union (over $P(G)$), which corresponds to the *least common multiple (lcm)* over $\bigotimes P(G)$. 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:

Lemma 2.3. *Let $\bigotimes P(G)$ be a square-free semi-group with prime generator G, and let $X, Y \in \bigotimes P(G)$ be two distinct elements, then $gcd(X, Y) = \bigotimes(S_X \cap S_Y)$, and $lcm(X, Y) = \bigotimes(S_X \cup S_Y)$, where $X = \bigotimes S_X$ and $Y = \bigotimes S_Y$, and $S_X, S_Y \in P(G)$ are the prime factors of X and Y, respectively.*

Proof. Since X and Y are square-free, $gcd(X,Y) = \prod_i p_i$, where p_i is a factor common to both X and Y. But this is exactly the expression given by $\bigotimes(S_X \cap S_Y)$. Likewise $lcm(X,Y) = \prod_i p_i$, where p_i is a factor of either X and Y. This is the same as $\bigotimes(S_X \cup S_Y)$. $\square$

Define the *factor-cardinality*, denoted $\|X\|_G$, for any $X \in \bigotimes P(G)$, as the number of prime factors from G in the factorization of X. Let $X = \bigotimes S_X$, 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 2.4. *Let $\bigotimes P(G)$ be a square-free semi-group with prime generator G, and let $X, Y \in \bigotimes P(G)$ be two distinct elements, then $gcd(X,Y) \in \bigotimes P(G)$.*

Proof. $gcd(X,Y) = \prod_i p_i$, where $p_i \in G$ is a factor common to both X and Y. It follows that $gcd(X,Y) \in \bigotimes P(G)$. $\square$

2.2. Primal 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., $N = m \times |G|$. 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 $B_i = B[(i-1) \times |G| + 1 : i \times |G|]$, with $1 \leqslant i \leqslant m$. 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 jth bit in B_i, and let $G[j]$ denote the jth prime in G. Define the *value* of B_i with respect to G as follows, $\nu(B_i, G) = \bigotimes\{G[j]^{B_i[j]}\}$. For example if $B_i = 1001$, and $G = \{2,3,5,7\}$, then $\nu(B_i, G) = 2^1 \times 3^0 \times 5^0 \times 7^1 = 2 \times 7 = 14$. Note also that if $B_i = 0000$ then $\nu(B_i, G) = 1$.

Define $\nu(B,G) = \{\nu(B_i,G)\colon 1 \leqslant i \leqslant m\}$, as the *primal block encoding* of B with respect to the base prime set G. It should be clear that each $\nu(B_i, G) \in \bigotimes P(G)$. 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 $\nu(B_1) = \bigotimes S_G(B_1) = \bigotimes\{2,7\} = 2 \times 7 = 14$, and the primal block encoding of B is given as $\nu(B) = \{14,30,3\}$. We also define the inverse operation $\nu^{-1}(\{14,30,3\}) = \nu^{-1}(14)\nu^{-1}(30)\nu^{-1}(3) = 100111100100 = B$. We also write $i \in \nu^{-1}(\cdot)$ if the ith bit is set in $\nu^{-1}(\cdot)$. 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 $A = A_1A_2\cdots A_m$, and $B = B_1B_2\cdots B_m$ be any two bit-vectors in $\{0,1\}^N$, with $N = m \times |G|$, and $A_i, B_i \in \{0,1\}^{|G|}$. Define $gcd(\nu(A),\nu(B)) = \{gcd(\nu(A_i),\nu(B_i))\colon 1 \leqslant i \leqslant m\}$. 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 $A = A_1A_2\cdots A_m$ 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:

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

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 primal 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 primal block encoding approach to represent candidates sequences, and uses join operations over the primal blocks to determine the frequency for each candidate. We detail these three aspects below.

3.1. Search space

The partial order induced by the subsequence relation is typically represented as a search tree. The sequence search tree can be 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

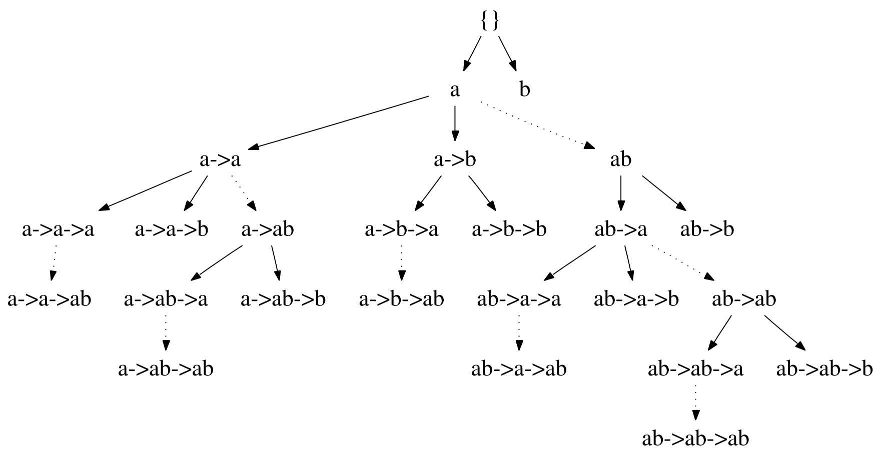


Fig. 2. Partial Sequence Search Tree: solid lines denote sequence extensions, whereas dotted lines denote itemset extensions.

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 $S = ab \to a$ and an item b for extending S, then $ab \to a \to b$ is a sequence-extension, and $ab \to ab$ is an itemset extension. Fig. 2 shows an example of a partial sequence search tree for two items, a and b, showing sequences up to size three; only the subtree under a is shown. Essentially all of the current sequence mining methods follow the sequence and itemset extension steps to traverse the search tree. The main difference lies in how the support for candidate patterns is determined. Below we describe how PRISM computes the frequency for each extension.

3.2. Primal block encoding

Consider the example sequence database shown in Fig. 3(a), consisting of 5 sequences of different lengths over the items $\mathcal{I} = \{a, b, c\}$. Let $G = \{2, 3, 5, 7\}$ be the base square-free prime generator set. Let us see how PRISM constructs the primal block encoding for a single item a. In the first step, PRISM constructs the primal 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 Fig. 3(b).

PRISM next computes the primal 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 primal encoding shown in Fig. 3(c), since $\nu(A) = \nu(1110)\nu(1000) = \{30, 2\}$. The full primal encoding for item a consists of all the sequence and position blocks, as shown in Fig. 3(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 primal 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 primal encoding. To do this it needs to keep an index with each sequence block to indicate which non-empty position blocks correspond to a given sequence block. Fig. 3(e) shows the actual (compact) primal 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 primal encoding for c. Its full encoding (see Fig. 3(d)) contains a lot of redundant information, which has been eliminated in the compact primal block encoding (see Fig. 3(e)).

It is worth noting that the support of a sequence S can be directly determined from its sequence blocks in the primal block encoding. Let $\mathcal{E}(S) = (\mathcal{S}_S, \mathcal{P}_S)$ denote the primal 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.

Theorem 3.1. *The support of a sequence S with primal block encoding $\mathcal{E}(S) = (\mathcal{S}_S, \mathcal{P}_S)$ is given as* $\text{sup}(S) = \sum_{v_i \in \mathcal{S}_S} \|v_i\|_G$.

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

(a)

sid	Bit-encoded pos	Primal-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}

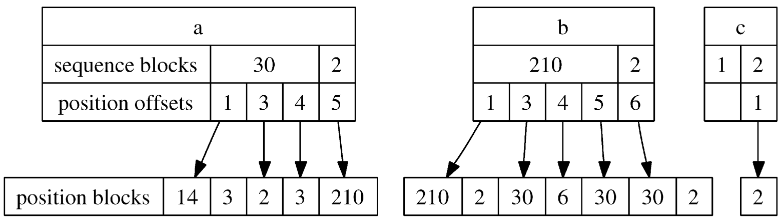
(b)

Bit-encoded sid	Primal-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)

Fig. 3. Example of primal block encoding: (a) Example database. (b) Position encoding for a. (c) Sequence encoding for a. (d) Full primal blocks for a, b and c. (e) Primal block encoding for a, b and c.

Proof. $\|v_i\|_G$ gives the number of prime factors from G in the factorization of block v_i. But a prime factor appears in block v_i at position j iff S occurs in sequence id j associated with that block. Thus $\sum_{v_i \in \mathcal{S}_S} \|v_i\|_G$ gives a count of all the sequences in the database where S occurs across all blocks, which equals $\sup(S)$. $\square$

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 Fig. 3(d)–(e)).

3.3. Support counting via primal block joins

We now show how the support for itemset and sequence extensions can be determined via primal block joins. Assume that we are at some node S in the sequence search tree, where S is a k-sequence. Let P be the parent node of S, which means that P is a $(k-1)$ length prefix of S. Let us assume that we have available the primal block encoding for P (namely $\mathcal{E}(P)$), as well as all its children. As illustrated in Fig. 2, S is extended by considering the other sibling nodes of S under parent P. For example, if $P = ab$, and $S = ab \to a$, then we can generate other extensions of S by also considering which extensions of P (the siblings of S) are frequent. In Fig. 2, the only other sibling is $T = ab \to b$. This suggests that both

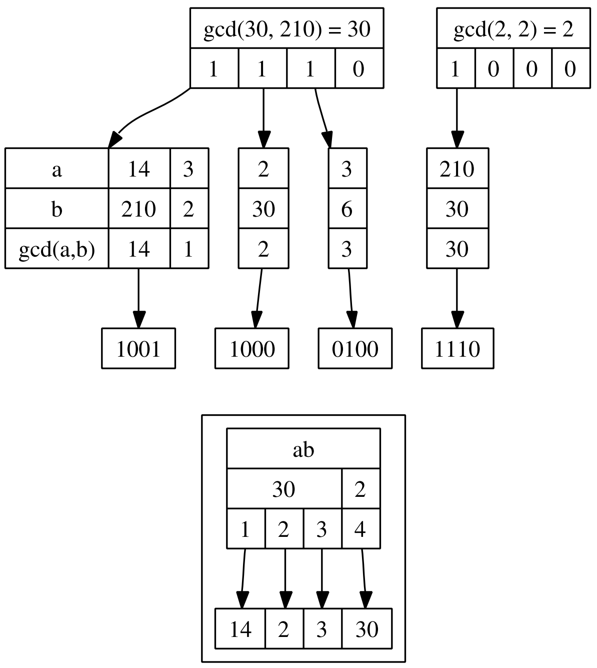


Fig. 4. Itemset Extensions via primal block joins.

items a and b can be used to extend $S = ab \to a$. There are two possible sequence extensions, namely $ab \to a \to a$ and $ab \to a \to b$, and one itemset extension $ab \to ab$. In fact, the support for any extension is always the result of joining the primal block encodings of two (possible the same) sibling nodes under the same prefix P. For example, $\mathcal{E}(ab \to a \to b)$ is obtained as result of a *primal block sequence join* operation over $\mathcal{E}(ab \to a)$ and $\mathcal{E}(ab \to b)$, whereas $\mathcal{E}(ab \to ab)$ is obtained as result of a *primal block itemset join* operation over $\mathcal{E}(ab \to a)$ and $\mathcal{E}(ab \to b)$.

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 primal block encodings for all single items. PRISM then recursively extends each node in the search tree, computes the support via the primal 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 primal block join operations. We will illustrate the primal block itemset and sequence joins using the primal encodings $\mathcal{E}(a)$ and $\mathcal{E}(b)$, for items a and b, respectively, as shown in Fig. 3(e).

Itemset extensions. Let us first consider how to obtain the primal block encoding for the itemset extension $\mathcal{E}(ab)$, which is illustrated in Fig. 4. 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 ith 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 us consider each sequence separately. Looking at sequence 1, we find in Fig. 3(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 Fig. 3(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 Fig. 4, which also shows the final primal 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 primal encoding. Thus $\sup(ab) = \|30\|_G + \|2\|_G = 3 + 1 = 4$. In general, we can state that:

Theorem 3.2. *Let X and Y be any two different siblings sharing a prefix node P in the sequence search tree, and let $\mathcal{E}(X) = (\mathcal{S}_X, \mathcal{P}_X)$ and $\mathcal{E}(Y) = (\mathcal{S}_Y, \mathcal{P}_Y)$ be their primal block encodings. Let $gcd_{XY} = gcd(\mathcal{S}_X, \mathcal{S}_Y)$. The primal block for the itemset extension of X with Y is given as $\mathcal{E}(XY) = (\mathcal{S}_{XY}, \mathcal{P}_{XY})$, where $\mathcal{P}_{XY} = \{gcd(\mathcal{P}_X^i, \mathcal{P}_Y^i) \mid i \in \nu^{-1}(gcd_{XY})\}$, and $\mathcal{S}_{XY} = \nu(B_{XY})$, where B_{XY} is the bit-vector defined as follows: $B_{XY}[i] = 1 \Leftrightarrow \mathcal{P}_{XY}^i \neq \mathbf{1}$.*

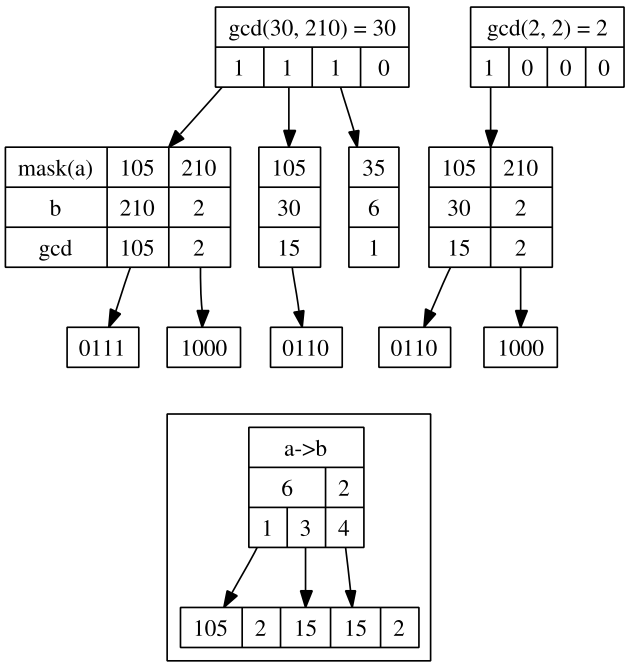


Fig. 5. Sequence extensions via primal block joins.

Proof. Note that gcd_{XY} gives exactly the set of sequences where both X and Y occur. Further $i \in \nu^{-1}(gcd_{XY})$ gives the sequence ids of those sequences. Next, $\mathcal{P}_{XY}$ is obtained by looking at precisely the set of blocks from those sequence ids. The set of positions within sequence id i where both X and Y co-occur are given by $gcd(\mathcal{P}_X^i, \mathcal{P}_Y^i)$. Finally, $\mathcal{S}_{XY}$ is derived from those position-blocks in sequence i that are non-empty (i.e., $\mathcal{P}_{XY}^i \neq \mathbf{1}$). □

For our example above, $gcd_{XY} = \{gcd(30, 210), gcd(2, 2)\} = \{30, 1\}$. Since $\nu^{-1}(\{30, 1\}) = 11101000$, we have to look at the position blocks from sids 1, 2, 3 and 5. Thus $\mathcal{P}_{XY} = \{gcd(\mathcal{P}_X^1, \mathcal{P}_Y^1), gcd(\mathcal{P}_X^2, \mathcal{P}_Y^2), gcd(\mathcal{P}_X^3, \mathcal{P}_Y^3), gcd(\mathcal{P}_X^5, \mathcal{P}_Y^5)\} = \{gcd(\{14, 3\}, \{210, 2\}), gcd(\{2\}, \{30\}), gcd(\{3\}, \{6\}), gcd(\{210\}, \{30\})\} = \{\{14, 1\}, \{2\}, \{3\}, \{30\}\}$. Eliminating the empty blocks, we have $\mathcal{P}_{XY} = \{\{14\}, \{2\}, \{3\}, \{30\}\}$. Since all sequences (i.e., those with sids 1, 2, 3, 5) have at least one non-empty block, we have $B_{XY} = 11101000$, and $\mathcal{S}_{XY} = \nu(B_{XY}) = \{30, 2\}$. This corresponds to the compact primal encoding shown in Fig. 4.

Sequence extensions. Let us consider how to obtain the primal block encoding for the sequence extension $\mathcal{E}(a \to b)$, which is illustrated in Fig. 5. 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 $a \to b$.

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 Fig. 5. 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 $a \to b$. 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 primal encoding of $\mathcal{E}(a \to b)$, sequence 3 drops out. The remaining sequences 1, 2, and 5, contribute at least one non-empty block, which yields $\mathcal{S}_{a \to b} = \nu(11001000) = \{6, 2\}$, as shown in Fig. 5, with support $\sup(a \to b) = \|6\|_G + \|2\|_G = 2 + 1 = 3$. In general, we can state that:

Theorem 3.3. *Let X and Y be any two (possibly the same) sibling sharing a prefix node P in the sequence search tree, and let $\mathcal{E}(X) = (\mathcal{S}_X, \mathcal{P}_X)$ and $\mathcal{E}(Y) = (\mathcal{S}_Y, \mathcal{P}_Y)$ be their primal block encodings. Let $gcd_{X \to Y} = gcd(\mathcal{S}_X, \mathcal{S}_Y)$. The primal block for the sequence extension of X with Y is given as $\mathcal{E}(X \to Y) = (\mathcal{S}_{X \to Y}, \mathcal{P}_{X \to Y})$, where $\mathcal{P}_{X \to Y} = \{gcd((\mathcal{P}_X^i)^{\triangleright}, \mathcal{P}_Y^i) \mid i \in \nu^{-1}(gcd_{XY})\}$, and $\mathcal{S}_{X \to Y} = \nu(B_{X \to Y})$, where $B_{X \to Y}$ is the bit-vector defined as follows: $B_{X \to Y}[i] = 1 \Leftrightarrow \mathcal{P}_{X \to Y}^i \neq \mathbf{1}$.*

Proof. The proof is similar to that in Theorem 3.2. $i \in \nu^{-1}(gcd_{XY})$ gives the sequence ids which contain both X and Y. We have look at the position blocks within each such sid i. $gcd((\mathcal{P}_X^i)^{\triangleright}, \mathcal{P}_Y^i)$ is obtained from those positions for Y that come after some position for X, since $(\mathcal{P}_X^i)^{\triangleright}$ turns on all the bits after the first set bit, which is the first position where X occurs. Finally, $\mathcal{S}_{X \to Y}$ is derived from those position-blocks in sequence i that are non-empty (i.e., $\mathcal{P}_{X \to Y}^i \neq \mathbf{1}$). □

For our example above, $gcd_{X\to Y} = \{gcd(30,210), gcd(2,2)\} = \{30,2\}$. As before, we need to further look at position blocks from sids 1, 2, 3 and 5. Thus $\mathcal{P}_{X\to Y} = \{gcd((\mathcal{P}^1_X)^{\triangleright}, \mathcal{P}^1_Y), gcd((\mathcal{P}^2_X)^{\triangleright}, \mathcal{P}^2_Y), gcd((\mathcal{P}^3_X)^{\triangleright}, \mathcal{P}^3_Y), gcd((\mathcal{P}^5_X)^{\triangleright}, \mathcal{P}^5_Y)\} = \{gcd(\{105,210\},\{210,2\}), gcd(\{105\},\{30\}), gcd(\{35\},\{6\}), gcd(\{105,210\},\{30,2\})\} = \{\{105,2\},\{15\},\{1\},\{15,2\}\}$. Eliminating the empty blocks, we have $\mathcal{P}_{X\to Y} = \{\{105,2\},\{15\},\{15,2\}\}$. Since $P^3_{X\to Y} = \mathbf{1}$, we have $B_{XY} = 11001000$, and $\mathcal{S}_{X\to Y} = \nu(B_{X\to Y}) = \{6,2\}$. This corresponds to the compact primal encoding shown in Fig. 5.

3.4. Optimizations

In our implementation, the primal block joins are performed efficiently using a collection of pre-computed, and compact, lookup tables. We detail these optimizations next.

Computing *gcd*. 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 $\bigotimes P(G)$ is $\bigotimes G = 9\,699\,690$. In total there are $|\bigotimes P(G)| = 256$ possible elements in semi-group $\bigotimes P(G)$.

In a naive implementation, the *GCD* lookup table can be stored as a two-dimensional array with cardinality $9\,699\,690 \times 9\,699\,690$, where $GCD(i,j) = gcd(i,j)$ for any two integers $i, j \in [1:9\,699\,690]$. This is clearly grossly inefficient, since there are in fact only 256 distinct (square-free) products in $\bigotimes P(G)$, 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 $\bigotimes P(G)$ 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 ith bit is '1' iff the i-element of G is in S. Then define the rank of $\bigotimes S$ as equal to the decimal value of $S^{\mathfrak{B}}$, i.e., $rank(\bigotimes S) = decimal(S^{\mathfrak{B}})$ (with the left-most bit being the least significant bit), which imposes a total order on $\bigotimes P(G)$. For example, the $rank(1) = decimal(00000000) = 0$, $rank(13) = decimal(00000100) = 32$, $rank(35) = decimal(00110000) = 12$, and $rank(9\,699\,690) = decimal(11111111) = 255$.

Lemma 3.4. *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 $rank(gcd(\bigotimes S, \bigotimes T)) = decimal(S^{\mathfrak{B}} \wedge T^{\mathfrak{B}})$.*

Proof. By Lemma 2.3 $gcd(\bigotimes S, \bigotimes T) = \bigotimes(S \cap T)$. Thus $rank(gcd(\bigotimes S, \bigotimes T)) = rank(\bigotimes(S \cap T)) = decimal(S^{\mathfrak{B}} \wedge T^{\mathfrak{B}})$. □

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 $i, j \in \bigotimes P(G)$. This brings down the storage requirements of the *GCD* table to just $256 \times 256 = 65\,536$ bytes, since each *rank* requires only one byte of memory (since $rank \in [0:255]$).

Other lookup tables. 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 $\bigotimes P(G)$. That is we store $CARD(rank(x)) = \|x\|_G$ for all $x \in \bigotimes P(G)$. 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 $x \in \bigotimes P(G)$. For example $MASK(rank(2)) = rank((2)^{\triangleright}) = rank(4\,849\,845) = 254$.

Finally, as an optimization for fast joins, once we determine gcd_{XY} or $gcd_{X\to Y}$ in the primal 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 [14], PrefixSpan [10] and SPAM [2]. The codes/executables for these methods were obtained from their authors. All experiments were performed on a laptop with 2.4 GHz Intel Celeron processor, and with 512 MB memory, running Linux.

Synthetic datasets. We used several synthetic datasets, generated using the approach outlined in [1]. These datasets mimic real-world transactions, where people buy a sequence of sets of items. Some customers may buy only some items from the sequences, or they may buy items from multiple sequences. The input sequence size and itemset size are clustered around a mean and a few of them may have many elements. 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

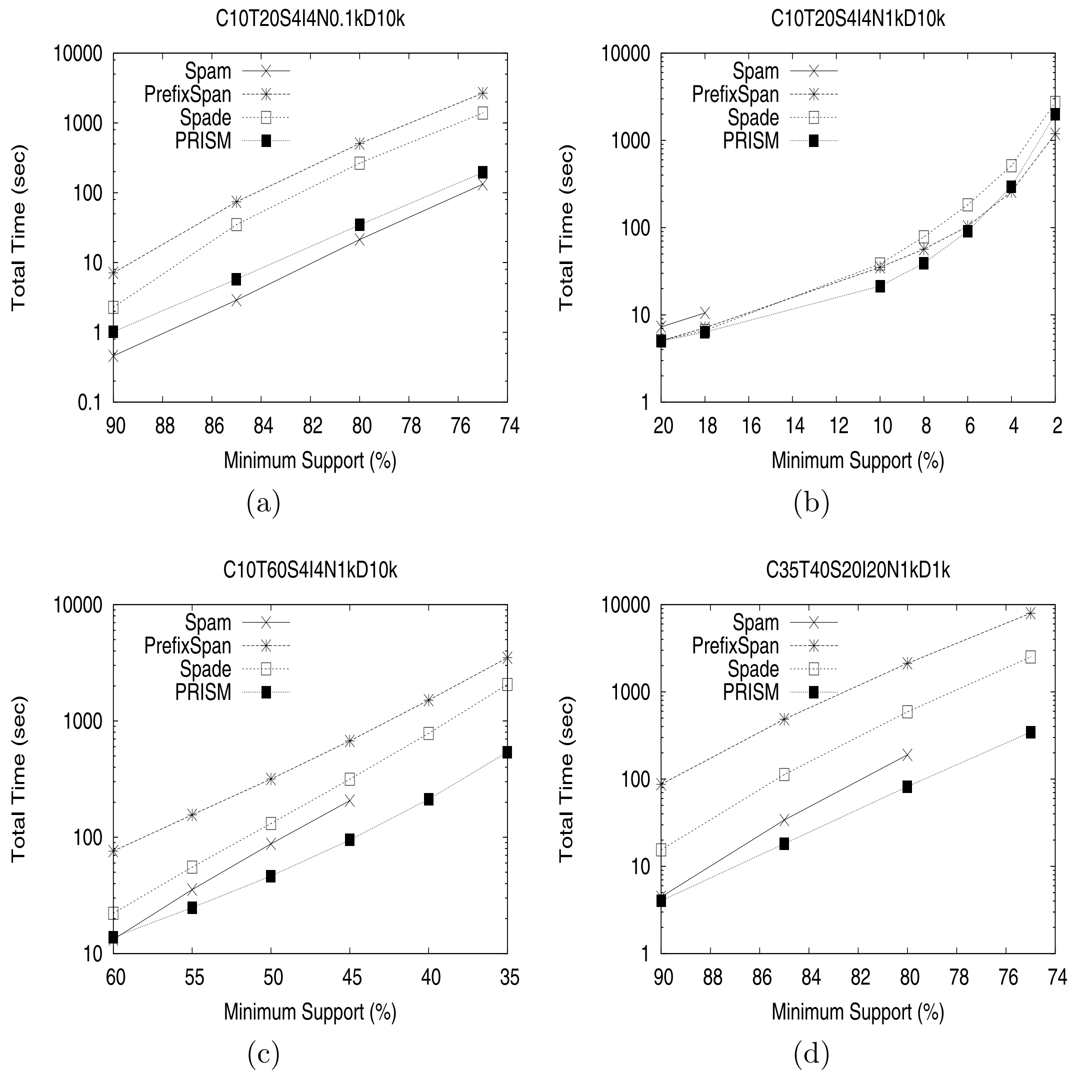


Fig. 6. Performance comparison with varying minimum support.

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 $N_I = 25\,000$, respectively.

Real datasets. We also compared the methods on two real datasets taken from [13]. `Gazelle` was part of the KDD Cup 2000 challenge dataset. It contains log data from a (defunct) web retailer. It has 59 602 sequences, with an average length of 2.5, length range of $[1, 267]$, and 497 distinct items. The `Protein` dataset contains 116 142 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).

4.1. Performance comparison

Figs. 6, 7 and 8 show the performance comparison of the four algorithms, namely, SPAM [2], PrefixSpan [10], SPADE [14] 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.

Fig. 6(a)–(d) show (small) datasets where all four methods can run for at least some support values. The datasets used in (a) and (b) are identical, except that (a) has only 100 distinct items ($N = 0.1k$), whereas (b) has 1000 ($N = 1k$); (c) has much larger average transaction length ($T = 60$) compared to (b), with ($T = 20$); (d) has larger sequence length ($C = 35$), and larger parameters compared to (a)–(c). For these datasets, we find that, with a few exceptions, 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

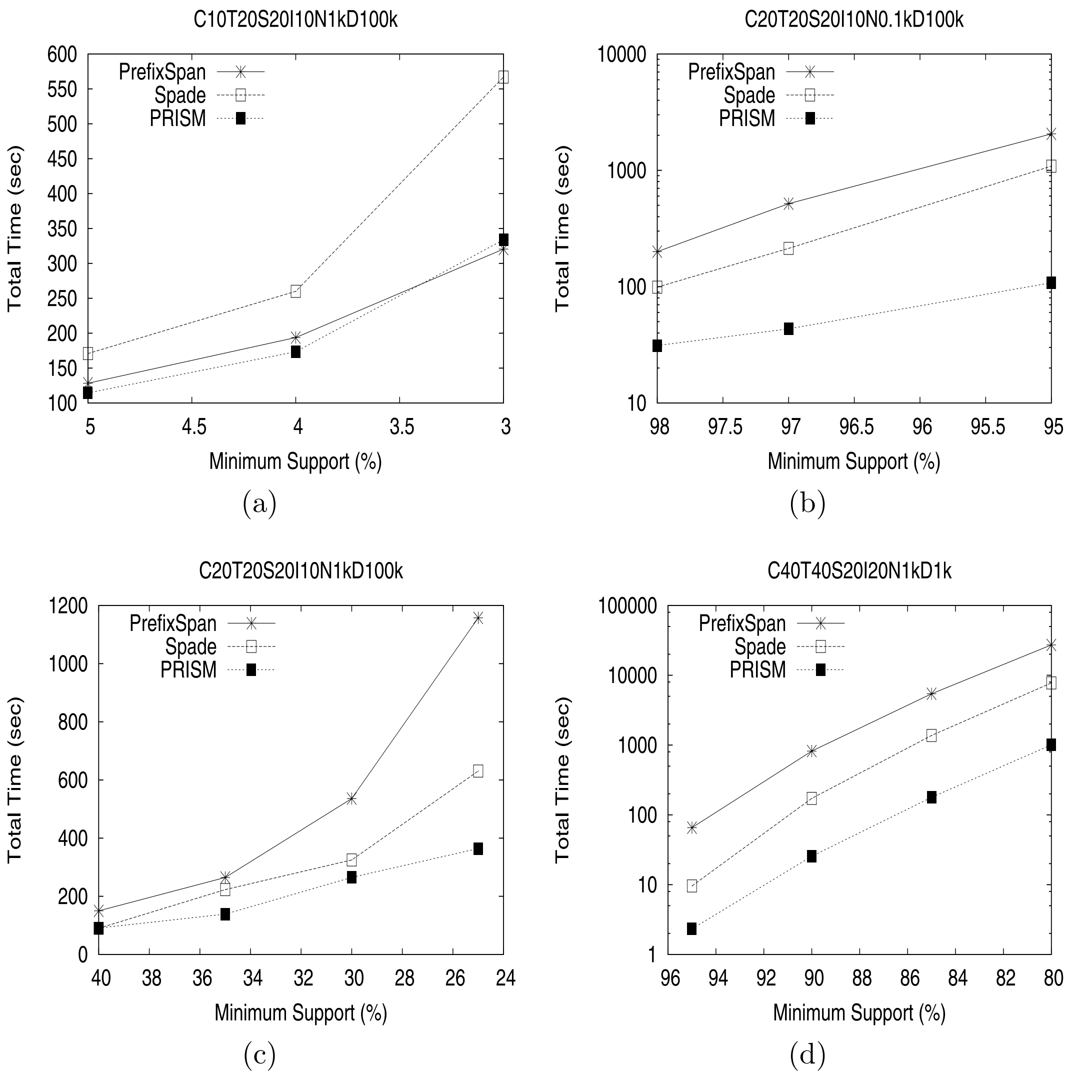


Fig. 7. Performance comparison with varying minimum support.

on C10T20S4I4N0.1kD10k). However, SPAM fails to run for lower support values. Compared to PrefixSpan and SPADE, Prism generally outperforms them by over an order of magnitude.

Fig. 7(a)–(d) show larger datasets (generally with $D = 100k$ sequences). On these SPAM could not run on our laptop, and thus is not shown in Fig. 7(a)–(d). Prism again outperforms PrefixSpan and SPADE, by up to an order of magnitude, except on (a), where PrefixSpan does the best. Nevertheless, if we reduce the number of items (to $N = 0.1k$ in (b)), increase the sequence length (to $C = 20$ in (c)) or increase other parameters (in (d)), we find that Prism is clearly superior.

Fig. 8(a)–(b) show performance on two datasets with larger average sequence ($C = 50$ in (a)) and transaction lengths ($T = 50$ in (b)). On these even PrefixSpan is unable to run, and SPADE runs only for larger support values. For these datasets, Prism is the most effective.

Finally, Fig. 8(c)–(d) 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 extensive set of 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. Since LAPIN [13] has also been shown to outperform existing approaches, especially on long, dense datasets, in the future, we would like to experimentally compare our approach with LAPIN.

Scalability. Fig. 9 shows the scalability of the different methods when we vary the different dataset parameters. The base values used are as follows: $C = 20$, $T = 20$, $S = 20$, $I = 10$, $N = 1k$, and $D = 100k$. We vary a single parameter at a time,

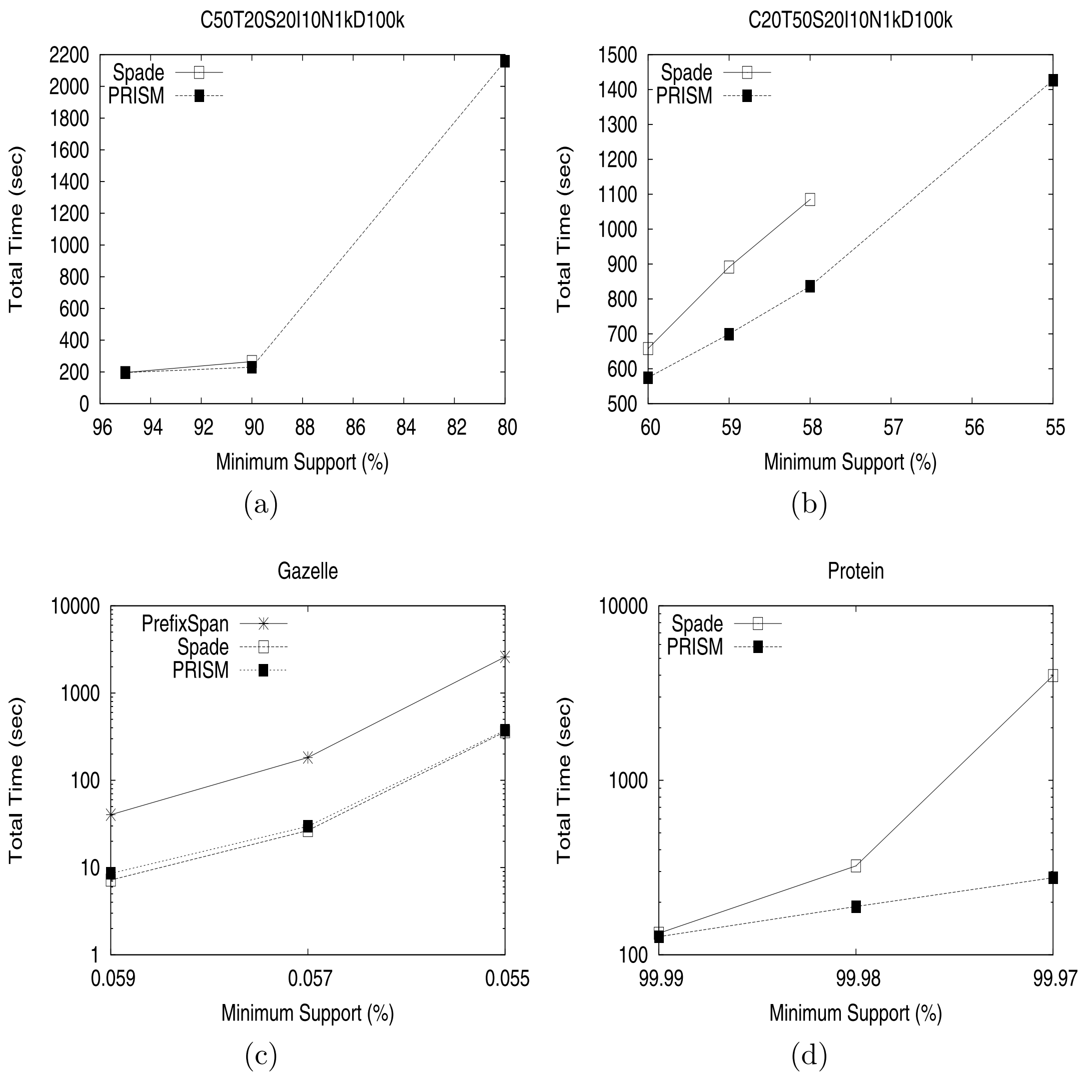
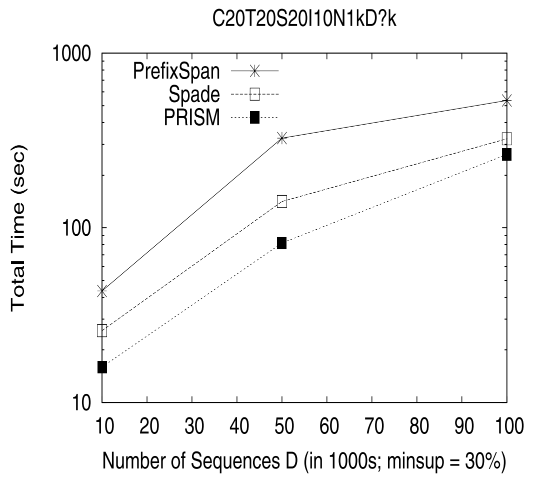


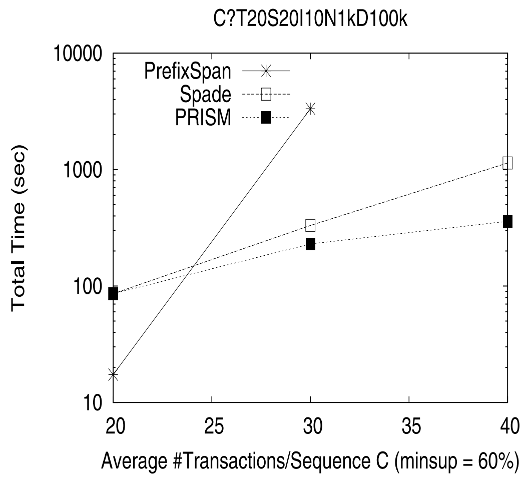
Fig. 8. Performance comparison with varying minimum support.

keeping all others fixed to the default values. Since SPAM failed to run on these larger datasets, we could not report on its scalability. Fig. 9(a) shows the effect of increasing the number of sequences from 10*k* to 100*k*. The trend is approximately linear. Fig. 9(b) shows the effect of increasing the average number of transactions per sequence (or the average sequence length). PrefixSpan is very sensitive to this parameter, displaying an exponential growth; it fails to run on larger values. Prism scales gracefully, and outperforms SPADE by over 5 times. Fig. 9(c) shows what happens when we increase the average transaction size within the sequences. All methods display an exponential growth, though PrefixSpan grows faster. Both PrefixSpan and SPADE could not run on $T = 50$. Fig. 9(d) shows the effect of larger itemsets in the dataset generation pool. The three methods are not very sensitive to this parameter, and the running time grows slowly. Finally, Fig. 9(e) shows the impact of longer sequences in the pool. We find that the time decreases exponentially as the maximal sequence size becomes longer. This mainly because fewer sequences are embedded into each customer sequence as the sequence length increases, resulting in fewer frequent sequences. It is worth noting that for shorter values, Prism is over an order of magnitude faster than PrefixSpan, and about two times faster than SPADE.

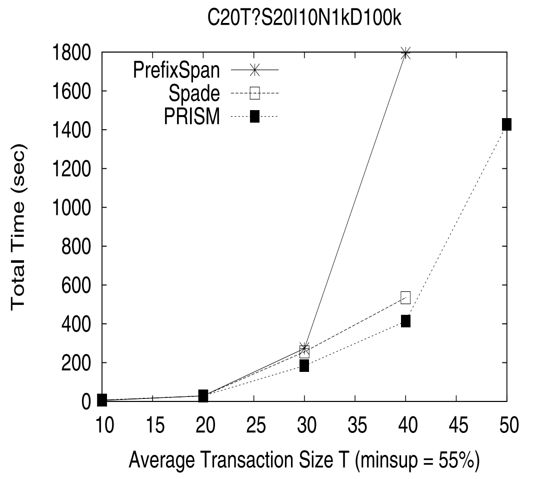
Memory usage. Fig. 10 shows the memory consumption of the four methods on a sample of the datasets. Fig. 10 plots the peak memory consumption during execution (measured using the `memusage` command in Linux). Fig. 10(a)–(b) quickly demonstrate why SPAM is not able to run on all except very small datasets. We find that its memory consumption is well beyond the physical memory available (512 MB), 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.



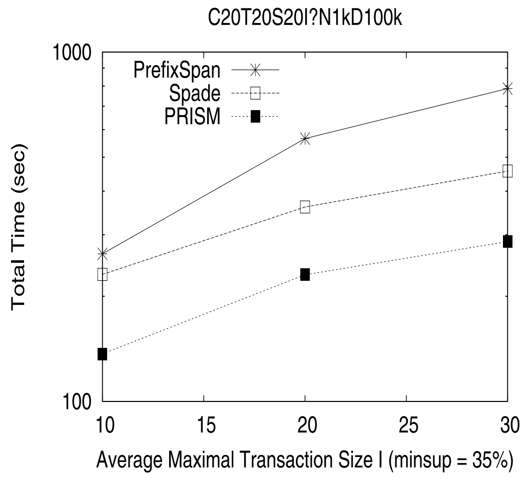
(a)



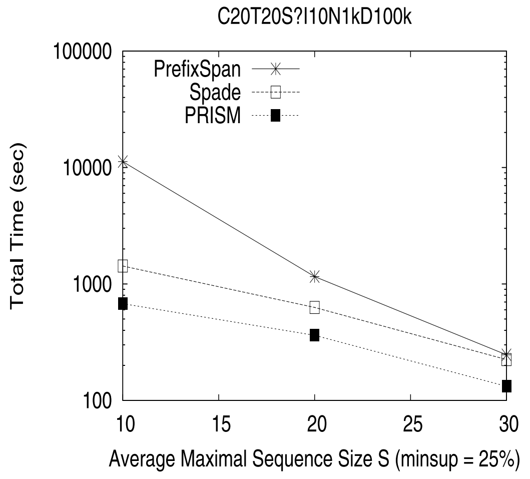
(b)



(c)



(d)



(e)

Fig. 9. Scalability with different database parameters.

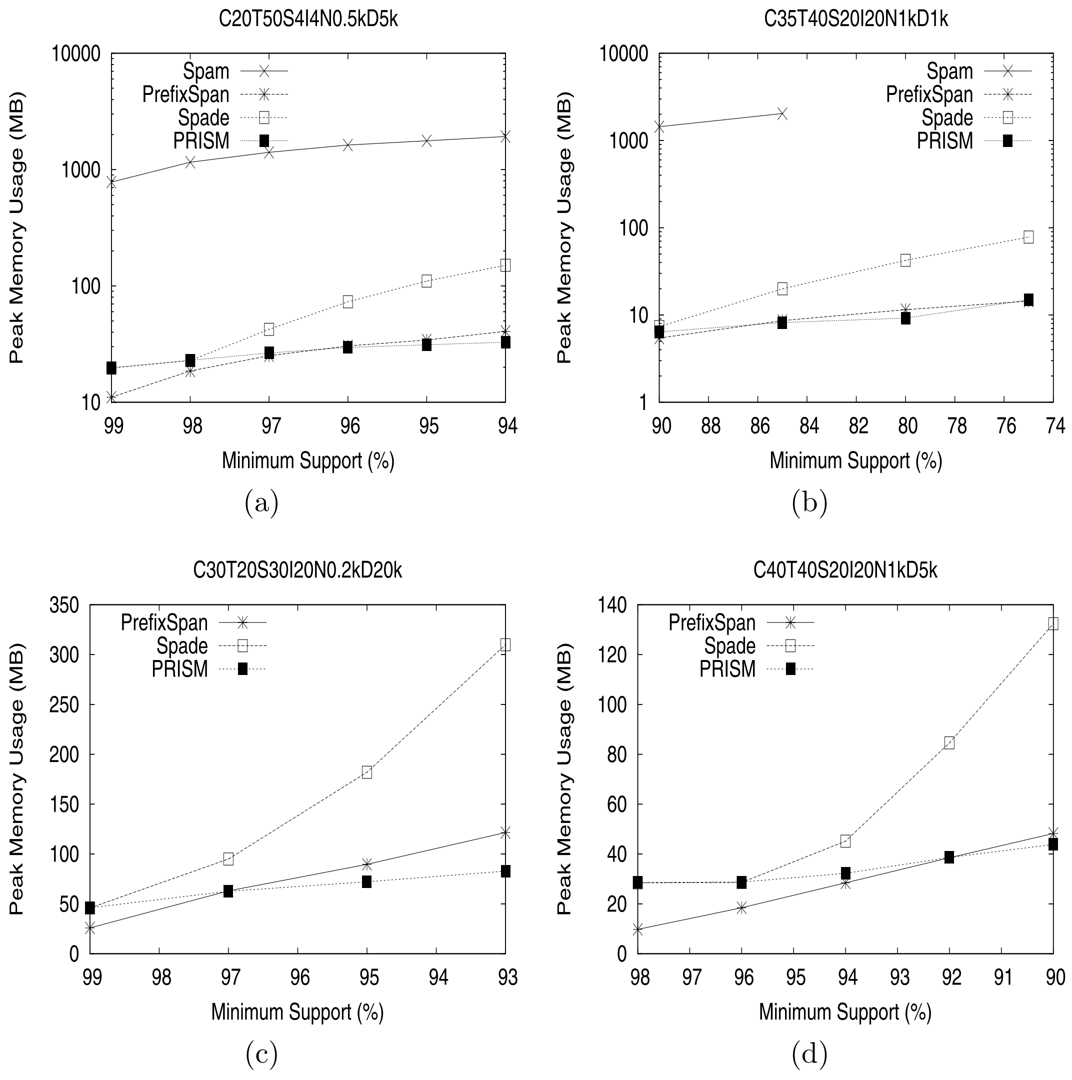


Fig. 10. Memory consumption.

5. 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, R. Srikant, Mining sequential patterns, in: 11th IEEE Intl. Conf. on Data Engineering, 1995.

[2] J. Ayres, J.E. Gehrke, T. Yiu, J. Flannick, Sequential pattern mining using bitmaps, in: SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining, July 2002.

[3] J.L. Balcazar, G. Casas-Garriga, On horn axiomatizations for sequential data, in: 10th International Conference on Database Theory, 2005.

[4] J. Gilbert, L. Gilbert, Elements of Modern Algebra, PWS Publishing Co., 1995.

[5] K. Gouda, M. Hassaan, M.J. Zaki, PRISM: A prime-encoding approach for frequent sequence mining, in: IEEE Int'l Conf. on Data Mining, October 2007.

[6] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, New York, 1997.

[7] K.L. Jensen, M.P. Styczynski, I. Rigoutsos, G.N. Stephanopoulos, A generic motif discovery algorithm for sequential data, Bioinformatics 22 (2006) 21–28.

[8] H. Mannila, H. Toivonen, I. Verkamo, Discovery of frequent episodes in event sequences, Data Min. Knowl. Discov. 1 (3) (1997) 259–289.

[9] F. Masseglia, F. Cathala, P. Poncelet, The PSP approach for mining sequential patterns, in: European Conf. on Principles of Data Mining and Knowledge Discovery, 1998.

[10] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, M.-C. Hsu, PrefixSpan: Mining sequential patterns efficiently by prefixprojected pattern growth, in: Int'l Conf. Data Engineering, April 2001.

[11] R. Srikant, R. Agrawal, Mining sequential patterns: Generalizations and performance improvements, in: 5th Intl. Conf. Extending Database Technology, March 1996.

[12] J. Wang, J. Han, BIDE: Efficient mining of frequent closed sequences, in: IEEE Int'l Conf. on Data Engineering, 2004.
[13] Z. Yang, Y. Wang, M. Kitsuregawa, Effective sequential pattern mining algorithms for dense database, in: Japanese Data Engineering Workshop (DEWS), 2006.
[14] M.J. Zaki, SPADE: An efficient algorithm for mining frequent sequences, Mach. Learn. J. 42 (1/2) (Jan/Feb 2001) 31–60.