Knowl Inf Syst (2018) 57:311–330
https://doi.org/10.1007/s10115-018-1161-6

REGULAR PAPER

Mining sequential patterns with itemset constraints

Trang Van[1,2] · Bay Vo[1] · Bac Le[3]

Received: 24 August 2016 / Revised: 17 November 2017 / Accepted: 18 January 2018 /
Published online: 1 February 2018

Abstract Mining sequential patterns is used to discover all the frequent sequences in a sequence database. However, the mining may return a huge number of patterns, while the users are only interested in a particular subset of these. In this paper, we consider the problem of mining sequential patterns with itemset constraints. In order to solve this problem, we propose a new algorithm named MSPIC-DBV, which is a pattern-growth algorithm that uses prefixes and dynamic bit vectors. This algorithm prunes the search space at the beginning and during the mining process. Moreover, it reduces the number of candidates that need to be checked. The experimental results show that the proposed algorithm outperforms the previous methods.

Keywords Sequential pattern · Itemset constraint · Dynamic bit vector · Prefix-tree

1 Introduction

Mining sequential patterns from a sequence database was first introduced by Agrawal and Srikant [1] and has been widely discussed in the literature [2,9,10,15,17,22,26,36]. Sequen-

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s10115-018-1161-6) contains supplementary material, which is available to authorized users.

✉ Bac Le
lhbac@fit.hcmus.edu.vn

Trang Van
vtt.trang@hutech.edu.vn; trangvtt@grad.uit.edu.vn

Bay Vo
vd.bay@hutech.edu.vn

1 Faculty of Information Technology, Ho Chi Minh City University of Technology, Ho Chi Minh City, Vietnam

2 Faculty of Computer Science, VNUHCM, University of Information Technology, Ho Chi Minh City, Vietnam

3 Faculty of Information Technology, VNUHCM, University of Science, Ho Chi Minh City, Vietnam

tial pattern mining is used to find all frequent subsequences, which are those with the number of occurrences in an input sequence database that satisfy a user-specified threshold, called the minimum support (*minSup*). It has broad applications, such as the analysis of customer purchase behavior [1], web logs [25], biological sequences [15,38] and text analysis [24].

However, with regard to the effectiveness of this approach, mining frequent sequences would return a large number of patterns, but only a small number of these actually corresponds to user requirements. If we are able to push the constraints defined by the users deep into the mining process, we can not only restrict the number and scope of discovered patterns to those that are closer to the user's interests, but also reduce the mining time. Therefore, constraint-based sequential pattern mining is proposed to find out the full set of sequential patterns satisfying a given constraint C, which is a Boolean function $C(\alpha)$ on the set of all patterns [23]. Pei et al. [23] generalized and presented seven categories of constraints, each of which has been the focus of many studies. Those categories include item constraint [33], super-pattern constraint [7,37], length constraint [18,34], aggregate constraint [3,5], regular expression constraint [6,8], duration constraint [16,26,29] and gap constraint [19,20]. Moreover, there are some other types of constraints, such as recency constraint [4] and compactness constraint [18], which can also be viewed as time constraints. However, to date there has been no research examining itemset constraints, a kind which is considered especially suitable for various applications. For example, when mining the sequences of customer transactions occurring at a certain supermarket, the analyst most likely aims to figure out the frequent subsequences that include a purchased set, such as (*shampoo, shower gel, soap*) or (*washing powder, water softener*). Likewise, when mining the medical records of patients, the goal is to determine the frequent patterns that contain a set of symptoms such as (*headache, rash, fever*). In conclusion, the itemset constraint requires that the discovered patterns must contain one of the given itemsets. Let us consider a sequential pattern of the form $\alpha = \langle a_1 a_2 \dots a_n \rangle$ and a collection of constraint itemsets $\mathbb{C} = \{c_1, c_2 \dots c_m\}$. The constraint can be expressed as

$$C_{\text{itemset}}(\alpha) \equiv \{\exists i : 1 \leq i \leq n, \exists k : 1 \leq k \leq m, a_i \supseteq c_k, c_k \in \mathbb{C}\}$$

In this paper, we deal with the problem of itemset constraints-based sequential pattern mining. The main contributions of this paper are as follows:

- First, mining sequential patterns with itemset constraints can be used to find all frequent subsequences which contain one of the itemsets indicated by the user. The problem is formally stated in Sect. 2.
- Second, we present the dynamic bit vector data structure which is applied to the sequential pattern, and propose a tree structure named the DBVS prefix-tree for efficiently mining sequential patterns with itemset constraints.
- Third, we give two propositions for pruning the search space quickly, reducing the calculations in pattern extension and limiting constraint checking.
- Finally, an efficient algorithm for mining sequential patterns with the itemset constraint, named MSPIC-DBV, is developed.

The rest of this paper is organized as follows. In Sect. 2, we introduce the main concepts related to sequential pattern mining with constraints, some definitions used throughout the paper and the problem statement. Section 3 presents the related work. The primary contributions are presented in Sect. 4, in which we present the DBV data structure, two novel propositions for improving efficiency mining and the proposed algorithm, MSPIC-DBV. The experimental results are presented in Sect. 5. Finally, we summarize our study and discuss directions for future work in Sect. 6.

2 Basic concepts and problem statement

Let I be a set of distinct items. An *itemset* is a subset of items I (without loss of generality, we assume that the items of an itemset are sorted in lexicographic order). The size of an itemset is the number of items in the itemset.

A *sequence* $s = \langle s_1 s_2 \dots s_n \rangle$ is an ordered list of itemsets. Each itemset is put in brackets. The size of a sequence is the number of itemsets in the sequence. The length of a sequence is the number of items in the sequence. A sequence with length k is called a k-sequence. For example, sequence $\alpha = \langle (\text{ABD})(\text{B})(\text{AC}) \rangle$ means that α consists of three itemsets (ABD), (B) and (AC). Its size is 3, and it is a 6-sequence.

A sequence $\beta = \langle b_1 b_2 \dots b_m \rangle$ is called a *subsequence* of another sequence $\alpha = \langle a_1 a_2 \dots a_n \rangle$, denoted as $\beta \subseteq \alpha$, if there exist integers $1 \leq i_1 < i_2 < \dots < i_m \leq n$ such that $b_1 \subseteq a_{i1}, b_2 \subseteq a_{i2}, \dots, b_m \subseteq a_{im}$. We call i_m *the position* where β is located in sequence α (here we keep the position of β's *last itemset*). For example, the sequence $\langle (\text{AB})(\text{C}) \rangle$ is a subsequence of $\langle (\text{AB})(\text{AC})(\text{C}) \rangle$ and it is located at the positions $\{2, 3\}$ (assuming positions starting at 1), but $\langle (\text{ABC}) \rangle$ is not a subsequence of $\langle (\text{AB})(\text{AC})(\text{C}) \rangle$.

The *database DB* for mining is a set of input sequences, each having a unique sequence identifier called *SID*. An input sequence s is said to *contain* a sequence α if α is a subsequence of s. In other words, α is said to be *present/appear* in s.

The *absolute support (support)* of a pattern α is defined as the number of input sequences in the database *DB* that contains α. Given a *minSup*, we say that a pattern is frequent if its support is greater than or equal to *minSup*.

Definition 1 (*Extending a sequence*) We create a new sequence by extending frequent k-sequence ($k > 0$) with a frequent item. There are two ways to extend a k-sequence: *sequence extension* and*itemset extension.* Let $\alpha = \langle a_1 a_2 \dots a_n \rangle$ be a frequent sequence and e be a frequent item. Let $SID_\alpha, SID_e, pos_\alpha, pos_e$ be the sequence IDs and positions of sequence α and item e. Extending sequence α with item e, we have new sequence α' as follows.

i. Itemset extension: $\alpha' = \langle a_1 a_2 \dots a'_n \rangle$ where item e is added to the last itemset, $a'_n = a_n \cup e$, where $SID_{\alpha'} = SID_e, pos_{\alpha'} = pos_e$ if $(SID_\alpha = SID_e) \wedge (pos_\alpha = pos_e)$.
ii. Sequence extension: $\alpha' = \langle a_1 a_2 \dots a_n e \rangle$ where item e serves as a new itemset and $SID_{\alpha'} = SID_e, pos_{\alpha'} = pos_e$ if $(SID_\alpha = SID_e) \wedge (pos_\alpha < pos_e)$.

Definition 2 (*Prefix*) A pattern $\beta = \langle b_1 b_2 \dots b_m \rangle$ is called a *prefix* of pattern $\alpha = \langle a_1 a_2 \dots a_n \rangle$ if and only if $b_i = a_i$ for all $1 \leq i \leq m-1$, $b_m \subseteq a_m, m < n$. For an example, the prefixes of pattern $\langle (\text{A})(\text{BC})(\text{D}) \rangle$ are: $\langle (\text{A}) \rangle$, $\langle (\text{A})(\text{B}) \rangle$ and $\langle (\text{A})(\text{BC}) \rangle$. According to this definition, we see that any sequence would be the prefix of its extended sequences.

Definition 3 (*Constraint satisfying*) A pattern $\beta = \langle b_1 b_2 \dots b_m \rangle$ is considered to contain an itemset c if $\exists i \in [1, m]$ such that $c \subseteq b_i$. Given a constraint itemset c, if pattern β contains the constraint itemset c, β is called a *c-satisfied pattern*.

Problem statement Given a sequence database D, a set of constraint itemsets $\mathbb{C} = \{c_1, c_2 \dots c_n\}$ and the minimum support *minSup* is specified by the user. The problem of mining sequential patterns with an itemset constraint is to find all frequent subsequences in the database which contain any itemsets in set $\mathbb{C}$.

$$FCP = \{\alpha | support(\alpha) \geq minSup \wedge \exists i : 1 \leq i \leq \text{size}(\alpha), \exists k : 1 \leq k \leq n, \alpha[i] \supseteq c_k, c_k \in \mathbb{C}\}$$

Table 1 An example database

Sequence ID	Data sequence
1	(AC)(AB)(BE)(ABD)
2	(AB)(ABC)(B)
3	(B)(ABDE)
4	(B)(AD)(B)
5	(AB)(AB)(A)
6	(AC)(B)(A)
7	(AC)(A)(A)

For an example, consider the database shown in Table 1 (used as a running example throughout this paper), absolute *minSup* = 3 and constraint itemsets $\mathbb{C} = \{(AB), (AD), (BE)\}$. Since itemset (BE) is infrequent, we have eight frequently satisfied-patterns, $FCP = \{\langle(A)(AB)\rangle: 3, \langle(AB)\rangle: 4, \langle(AB)(A)\rangle: 3, \langle(AB)(AB)\rangle: 3, \langle(AB)(B)\rangle: 3, \langle(AD)\rangle: 3, \langle(B)(AB)\rangle: 4, \langle(B)(AD)\rangle: 3\}$.

3 Related work

In the past decade, there have been many studies on the design of efficient algorithms for mining sequential patterns. The algorithms for mining sequential patterns with constraint are basically developed from these typical algorithms. Most existing pattern mining algorithms can be classified into three categories:

The first category includes horizontal database format algorithms, including the first three of the following: AprioriAll, AprioriSome, DynamicSome [1] and GSP [26]. All of these operate based on the same general idea: *starting with shorter sequences and extending them toward longer ones.* They adopt a multiple pass, candidate generation and test approach to sequential pattern mining. GSP is also a typical Apriori-like method, but it incorporates time constraints, sliding time windows and taxonomies in sequential patterns. The algorithms for mining with constraints in this category are SPIRIT [8] and MSP-Miner [6]. SPIRIT is a family of Apriori-based algorithms that uses regular language to constrain the pattern mining process. The algorithm is essentially similar to GSP, but it reduces the candidate search space by using some different relaxations of the regular expression constraint. MSP-Miner is another Apriori-based algorithm with regular expression constraints, but it is used for mining multi-sequential patterns.

The second category includes vertical database format algorithms, such as SPADE [36], SPAM [2], PRISM [9]. The variants of these algorithms are cSPADE [35] and CCSM [21] (variants of SPADE), and Pex-SPAM [11] (a variant of SPAM). These are the same as the original algorithms, except that they incorporate one or more of the syntactic constraints as checks during the mining process.

The SPADE algorithm uses a vertical *id-list* database format, which consists of a list of pairs (input sequence and event identifier) for each sequence. It is possible to directly obtain the sequence support from the sequence *id-list* without scanning the database. Therefore, this approach does not scan the database many times, as the horizontal database format approach does, but instead only does so three times. Moreover, it is considered as only having one scan if a preprocessing step is included. In addition, SPADE stores the candidates in a lattice,

including prefix-based equivalence classes, and so can independently process each sub-lattice in the main memory.

SPAM is another approach among the various vertical database format algorithms, and its major difference is in the form of data representation used. Instead of using the *id-list*, SPAM uses a bitmap representation. Each bitmap has a bit corresponding to each transaction of the sequences in the database. The data structure of SPAM is a lexicographic tree, which can be pruned by two techniques, namely S-step and I-step pruning. SPAM is much faster than SPADE but less space efficient, as the bitmap keeps the transactions even if they never participate in the support count of the sequence.

Similar to SPAM, PRISM also utilizes a vertical approach for enumeration and support counting. In particular, PRISM makes use of a prime block encoding approach to compress the bitmap of SPAM. Each prime block corresponds to 8 bits (or 16 bits). Every candidate sequence is represented by two pieces of information: sequence blocks (that indicate which input sequence ids contain the candidate) and position blocks (that indicate the positions which the candidate appears within an input sequence). PRISM only removes empty position blocks and cannot remove empty sequence blocks, even though there may be many blocks where the candidate does not appear. A new approach (DBV-ISP [31]) has been proposed to overcome this, and it uses *dynamic bit vector* architecture (DBV) to mine frequent itemsets. It is also applied in mining inter-sequence patterns [14,32], closed sequential patterns [27] and sequential rules [28].

The third category includes pattern-growth algorithms, such as FreeSpan [10] and PrefixSpan [22]. The variants with constraints are PTAC [3], DELISP [16] and PG [23]. The basic idea of these methods is to project a large database into smaller projected databases based on the frequent item sets, and then grow the patterns in these projected databases.

Among the various algorithms discussed earlier, the vertical database format algorithms are seen as better than the others [2,9,36]. This is because they do not have to make multiple database scans, like horizontal algorithms, nor do they require any computational cost for constructing projected databases, like pattern-growth algorithms. In particular, as mentioned above, the DBV approach which uses a vertical data format and data compression is more efficient in terms of execution time and memory usage. We thus study and apply this method to solve the problem of mining sequential patterns with itemset constraints.

4 Proposed algorithm

4.1 DBV architecture

4.1.1 DBV data structure

We use a dynamic bit vector approach to represent candidate sequences. As introduced in [31], a dynamic bit vector (DBV) includes a bit-vector which is a list of bytes after removing zero bytes from the head and tail, and it includes an index to indicate the location of the first nonzero byte in the bit-vector (Fig. 1).

4.1.2 DBV for Sequential patterns (DBVS)

In a sequence database, two facts are required for a candidate pattern. First, we need to know which input sequences contain the pattern. In order to represent this information, we use a bit-vector, in which a bit corresponds to an input sequence of the database. If the kth input

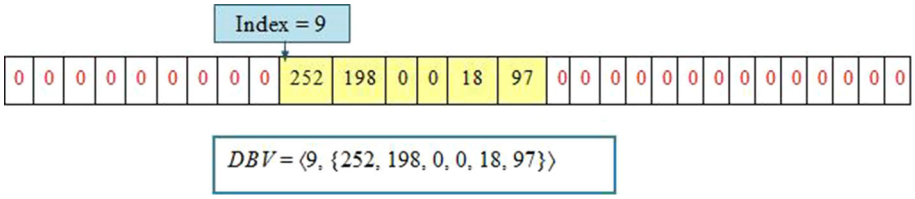

Fig. 1 An example of a bit-vector and DBV

sequence contains the pattern then the kth bit is set to '1', otherwise it is set to '0'. A byte (in decimal) represents a block which consists of eight input sequences, called a *sequence block*. The number of input sequences is large, but the sequential pattern only appears in some sequences, and so there are many zero bytes in the bit-vector. We therefore apply DBV to eliminate all zero bytes from head and tail of the bit-vector. Second, we need to know the positions where the pattern appears in the input sequence.

In brief, in order to represent the candidate sequence information, we use a data structure called DBVS (dynamic bit vector for sequence) including: a DBV and a list of positions appearing in the input sequences. The list positions in each input sequence are represented in the form of start-Pos: {list positions}, where startPos is the first appearance of the candidate in the input sequence [27].

Count support The candidate sequence's support is obtained by counting the number of '1' bits.

4.2 DBVS prefix-tree

We use a prefix-tree structure [30] to arrange the candidate patterns. In the prefix-tree, each node contains one candidate along with its DBVS. The root of the tree at level 0 is labeled with a null sequence. At level k, a node is labeled with a k-sequence. Recursively, we have nodes at the next level $(k+1)$ by extending the k-sequence with a frequent item. In this way, the k-pattern is a prefix of the $(k+1)$-pattern. It can be easily seen that each node in the prefix-tree n is certainly a prefix of all the sub-nodes which are extended from n and their descendants.

4.3 MSPIC-DBV algorithm

We use depth-first search to search for the sequential patterns stored in the prefix-tree. Our proposed algorithm proceeds as follows. First, we scan the database to find frequent 1-patterns (containing a single item), and then we recursively perform sequence extension and itemset extension (Definition 2) with one of these patterns to generate larger patterns. After that, when a pattern has no possible extension, the algorithm backtracks to generate others using other patterns. Note that, in each candidate generation, we need to check the frequency and the constraint for every pattern. Since the search space can be very large, we apply the traditional Apriori property [1] to prune it. Moreover, we effectively prune the search space by using the provided proposition.

In the following section, we introduce two propositions and present an algorithm called MSPIC-DBV for mining frequent sequential patterns with itemset constraints based on the provide propositions.

Proposition 1 (Pruning prefix) *Given an itemset c and a pattern P containing c, if c is not present in input sequence S of the database, P is not either.*

Proof Assume that c is not present in input sequence S of the database but P is present. This means that the input sequence S contains P. According to the Apriori property, the input sequence Salso contains itemset c, because c is a sub-pattern of P. This is contrary to the assumption. □

Proposition 2 (Checking constraint) *Given a pattern P and a constraint itemset c. If P satisfies constraint c, then all the patterns that have P as a prefix also satisfy constraint c.*

Proof Let $P = \langle p_1 p_2 \dots p_k \rangle$, where each p_i corresponds to an itemset. If P satisfies constraint c, then this means that $\exists i \in [1, k]$ such that $c \subseteq p_i$. Let Q is a pattern which has P as a prefix, according to the definition of the prefix in Sect. 2, $Q = \langle p_1 p_2 \dots p_k x_1 x_2 \dots x_n \rangle$, it is thus clear that Q contains the constraint itemset c. □

From Proposition 1, we propose a transformation to prune the sub-trees by prefix. From Proposition 2, if node X satisfies the constraint, we do not need to check the constraint for all nodes in the sub-tree which have prefix X. The MSPIC-DBV algorithm has four main steps, as presented below.

MSPIC-DBV algorithm

Input: *DB*, $\mathbb{C} = \{c_1, c_2 \dots, c_k\}$, *minSup*.

Output: All sequential patterns satisfying *minSup* and itemset constraints.

$FCP = \{\alpha | \, sup(\alpha) \geq minSup \wedge \exists i: 1 \leq i \leq size(\alpha), \exists k: 1 \leq k \leq n, \alpha[i] \supseteq c_k\}$.

Method:

1. F_1 = find all frequent *1*-patterns and their DBVSs*;*
2. *FIND_FRE_CONSTRAINT_ITEMSET(F_1, $\mathbb{C}$, minSup);*
3. Transform DBVSs of the atoms in F_1;
4. For each node n in F_1
5. For each c in $\mathbb{C}$ do
6. If (n satisfies constraint c) then
7. $FCP = FCP \cup \{n.sequence\}$;
8. ***DBV-PREFIX-EXTENSION(n, F_1, F_1, minSup);***
9. *break;*
10. If (n does not satisfy any c, $\forall c \in \mathbb{C}$)
11. ***DBV-PREFIX-EXTENSION-CHECK(n, F_1, F_1, minSup, $\mathbb{C}$);***

Step 1: Find sequential 1-patterns The algorithm scans the database once to find all frequent 1-patterns and calculates their corresponding DBVSs (line 1). We call a frequent 1-pattern an *atom*. We will perform pattern growth from these atoms.

Step 2: Find frequent constraint itemsets Filter set $\mathbb{C}$ such that $\mathbb{C}$ only retains the frequent constraint itemsets. If $\mathbb{C} = \emptyset$ then $FCP = \emptyset$ otherwise we find FCP. This step will thus find the frequent constraint itemsets and their DBVs (line 2).

- *Remark about the frequency of an itemset*

  (i) If itemset c is frequent then every item of c is frequent. If c has an item which is not frequent then c is not frequent (based on Apriori).
  (ii) If itemset c has support $\geq$ *minSup*, c is frequent. Because the support of a pattern is directly obtained from DBVS, we can get the DBVS of itemset c by using the itemset-extension for each item in c (using bitwise AND).

Based on these remarks, we can easily check the frequency of each constraint itemset without scanning the database using Procedure 1.

Procedure 1. FIND_FRE_CONSTRAINT_ITEMSET(F_1, $\mathbb{C}$, *minSup*)

```
1.  For each c in ℂ do
2.      DBVS(c) = ∅;
3.      For each item i in c do
4.          If (i does not exist in F1) then
5.              ℂ = ℂ - {c}; break;
6.      If (∀i ∈ c, i is frequent) then
7.          DBVS(c) = Itemset-Extension(i1, ik), ik ∈ c, k ∈ [2, size(c)];
8.      If (DBVS(c).Support < minSup) ℂ = ℂ - {c};
```

Step 3: Transform the DBVSs of all atoms in F_1 (line 3, MSPIC-DBV algorithm).

Based on Proposition 1, we see that if the input sequences do not contain the constraint itemset c, then they also do not contain the c-satisfied patterns. Therefore, at the beginning of the pattern extension process, we set the kth bit in the atom's bit-vector to zero if the kth bit in the bit-vector of the constraint itemset is zero. For example, in Table 1, since itemset (AB) is present in the input sequences s_1, s_2, s_3, s_5, bit-vector(AB) = 1110100**0**. Similarly, bit-vector(C) = 1100011**0**. Because s_6, s_7 do not contain (AB), they should not contain the (AB)-satisfied patterns which are extended from the atom ⟨C⟩, and thus bit-vector(C) = 1100000**0**. In order to do this, we simply use bitwise AND between bit-vector(c) and bit-vector(*atom*), where the $c \in \mathbb{C}$, atom $\in F_1$. AND operation between two DBV's bit-vectors was shown in [31]. Note that when one bit in bit-vector(*atom*) changes from '1' to '0', we delete the list positions. Because the cardinality $|\mathbb{C}| \geq 1$, let d be a bit-vector represented for all members in $\mathbb{C}$, and we perform the following two steps:

- **Step 3.a)** Do OR operation: $d = bitwise\ OR(\text{DBV}(c), c \in \mathbb{C})$.
- **Step 3.b)** Do AND operation: $\forall atom \in F_1$, DBVS(*atom*) = bitwise *AND*(d, DBVS(*atom*)).

An example of the transformation is shown in Sect. 4.4.

- *The usefulness of transformation*

First, transformation is used to prune the search space via the prefixs and subsequences. Indeed, transformation helps to reduce the amount of '1' bits in the bit-vector, which decreases the support of the pattern. After transformation, if atom α is no longer frequent then the candidates extended from α as well as the candidates containing α are infrequent. We can thus prune the sub-tree rooted at α and the subsequences of α by removing α from F_1. This thus means that we can prune the search space at the beginning of the mining process. Consider the example in Sect. 4.4, in which atom C has become infrequent after transformation, we thus prune T_1 and T_2 (see Fig. 2).

Moreover, transformation helps to prune the search sub-spaces efficiently. For example, let $d = \{0, 01011000\}$, $minSup = 2$ and two atoms A and B. Let DBV(A) = {0, 11010000} then transformed-DBV(A) = {0, 01010000} and let DBV(B) = {0, 10011000} then transformed-DBV(B) = {0, 00011000}. If we do not transform then the candidate pattern $\langle(AB)\rangle$ with DBV($\langle(AB)\rangle$) = {0, 10010000} is frequent since its support is 2. So we must continuously perform pattern growth from prefix $\langle(AB)\rangle$. Otherwise, the candidate $\langle(AB)\rangle$ with transformed-DBV($\langle AB\rangle$) = {0, 00010000} is not frequent since its support is 1. Thus, we prune all the sequences with prefix (AB). Because we generate candidates from the atoms in F_1 and extend patterns using F_1, transformation the DBVS of the atoms leads to changes in the DBVS of all the candidates at the next levels. This potentially helps to prune a lot of sequences in the search space with the use of prefixes. The pruning could thus happen during the mining process.

Second, transformation is used to reduce the calculations in pattern extension. For example, the DBVS of atom A after transformation has reduced two data sequences (11111**11**0 changes to 11111**00**0) along with lists of positions that have changed to $\emptyset$. Thus, when extending the patterns in the sub-tree rooted at atom $\langle A\rangle$, there is no need to manipulate anything on the list of positions which has changed to $\emptyset$.

Step 4: Check constraint and extend the patterns (lines 4–11 in MSPIC-DBV algorithm).

Starting from the first level, each sub-tree rooted at a node n labeled with an atom can be processed independently. Before extending the atoms toward longer patterns, we check the constraint. If n satisfies the constraint we add it to *FCP* and perform pattern extension by using procedure 3, otherwise using procedure 2 (*based on Proposition* 2).

Procedure 2. ***DBV-PREFIX-EXTENSION-CHECK(p, S, I, minSup, ℂ)***

//Sequence – Extension

1. $S_1 = \{i \in S | sup(let\ pi = Sequence\ -\ Extension(p, i)) \geq minSup\}$;
2. For each item i in S_1 do
3. For each c in $\mathbb{C}$ do
4. If (pi satisfies constraint c) then
5. $FCP = FCP \cup \{pi.sequence\}$;
6. *DBV-PREFIX-EXTENSION(pi, S_1, item in S_1 greater than i, minSup);*
7. break;
8. If (pi does not satisfy any c, $\forall c \in \mathbb{C}$) then
9. *DBV-PREFIX-EXTENSION-CHECK(pi, S_1, item in S_1 greater than i, minSup, ℂ)*;

//Itemset - Extension: similar to Sequence - Extension

10. $I_1 = \{i \in I | sup(let\ pi = Itemset\ -\ Extension(p, i)) \geq minSup\}$;
11. For each item i in I_1 do
12. For each c in $\mathbb{C}$ do
13. If (pi satisfies constraint c) then
14. $FCP = FCP \cup \{pi.sequence\}$;
15. *DBV-PREFIX-EXTENSION(pi, S_1, item in I_1 greater than i, minSup);*
16. break;
17. If (pi does not satisfy any c, $\forall c \in \mathbb{C}$) then
18. *DBV-PREFIX-EXTENSION-CHECK(pi, S_1, item in I_1 greater than i, minSup, ℂ);*

The details of procedure 2 are as follows. As a depth-first approach, the overall process starts from sequence-extension and then itemset-extension. Let S_1 be the set of items used for sequence-extension which generate frequent patterns (line 1). For each item i in S_1, if extended pattern pi satisfies any constraint itemset then we add it to FCP and call the procedure 3, otherwise we recursively call this procedure for an extended pattern (lines 2–8). After that, itemset-extension is performed in a similar manner (lines 9–16). This recursive process is repeated until all frequent sequences have been enumerated. Procedure 3 is similar to procedure 2, but it only performs pattern extension without checking the constraint.

Procedure 3. ***DBV-PREFIX-EXTENSION(p, S, I, minSup)***

//Sequence Extension

1. $S_1 = \{i \in S | sup(\text{let } pi = \text{Sequence-Extension}(p, i)) \geq minSup\}$;
2. For each item i in S_1 do
3. $FCP = FCP \cup \{pi.sequence\}$;
4. *DBV-PREFIX-EXTENSION(pi, S_1, item in S_1 greater than i, minSup);*

//Itemset Extension is similar to Sequence Extension

Table 2 The frequent 1-patterns (atoms)

F_1	Bit-vector	DBV	Support
A	11111110	{0, 254}	7
B	11111100	{0, 252}	6
C	11000110	{0, 198}	4
D	10110000	{0, 176}	3

Table 3 Example of extending 1-pattern ⟨A⟩ via item B, and the resulting pattern is ⟨(AB)⟩

A	Bit-vector	1	1	1	1	1	1	1	0
	Positions	1: {1, 2, 4}	1: {1, 2}	2: {2}	2: {2}	1: {1, 2, 3}	1: {1, 3}	1: {1, 2, 3}	∅
B	Bit-vector	1	1	1	1	1	1	0	0
	Positions	2: {2, 3, 4}	1: {1, 2, 3}	1: {1, 2}	1: {1, 3}	1: {1, 2}	2: {2}	∅	∅
(AB)	Bit-vector	1	1	1	0	1	0	0	0
	Positions	2: {2, 4}	1: {1, 2}	2: {2}	∅	1: {1, 2}	∅	∅	∅

Table 4 The frequent constraint itemsets found after step 2

Constraint itemset	Bit-vector	DBV	Support
(AB)	11101000	{0, 232}	4
(AD)	10110000	{0, 176}	3

Table 5 Doing bitwise AND between d and DBVS(A)

d	Bit-vector	1	1	1	1	1	0	0	0
A	Bit-vector	1	1	1	1	1	1	1	0
	Positions	1: {1, 2, 4}	1: {1, 2}	2: {2}	2: {2}	1: {1, 2, 3}	1: {1, 3}	1: {1, 2, 3}	∅
Trans- A	Bit-vector	1	1	1	1	1	0	0	0
	Positions	1: {1, 2, 4}	1: {1, 2}	2: {2}	2: {2}	1: {1, 2, 3}	∅	∅	∅

4.4 Illustration of the MSPIC-DBV process

An execution of the MSPIC-DBV algorithm for the database shown in Table 1 with *minSup* = 3 and $\mathbb{C} = \{(AB), (AD), (BE)\}$ is described step by step, as follows:

Step 1 MSPIC-DBV scans the *DB* to find $F_1 = \{A, B, C, D\}$ as in Table 2.

Step 2 We check the frequency of each constraint itemset in $\mathbb{C}$. Extending 1-pattern ⟨A⟩ via item B using itemset-extension, we have a frequent itemset (AB), as shown in Table 3.

Similarly, we have itemset (AD) is frequent and (BE) is infrequent since item E is not in F_1. After filtering set $\mathbb{C}$, we have the results shown in Table 4.

Step 3 Transform the DBVSs of the atoms in F_1.

Step 3.a) Let d = DBV(AB) OR DBV(AD) = {0, 11101000} OR {0, 10110000} = {0, 11111000} = {0, 248}.

Sep 3.b) We then perform DBVS($atom$) = $AND(d$, DBVS($atom$)). The results are shown in Table 5 for atom A.

After transformation, we list the positions at the 6th bit, and the 7th bit becomes ∅. This reduces the calculations needed for pattern extension from the prefix ⟨A⟩. We use a

Table 6 DBVs transformation of the atoms in F_1

F_1	Transformed DBV	Transformed Bit-vector	Support
A	{0, 254} & {0, 248} = {0, 248}	11111000	5
B	{0, 252} & {0, 248} = {0, 248}	11111000	5
C	{0, 198} & {0, 248} = {0, 192}	11000000	2 (remove C from F_1)
D	{0, 176} & {0, 248} = {0, 176}	10110000	3

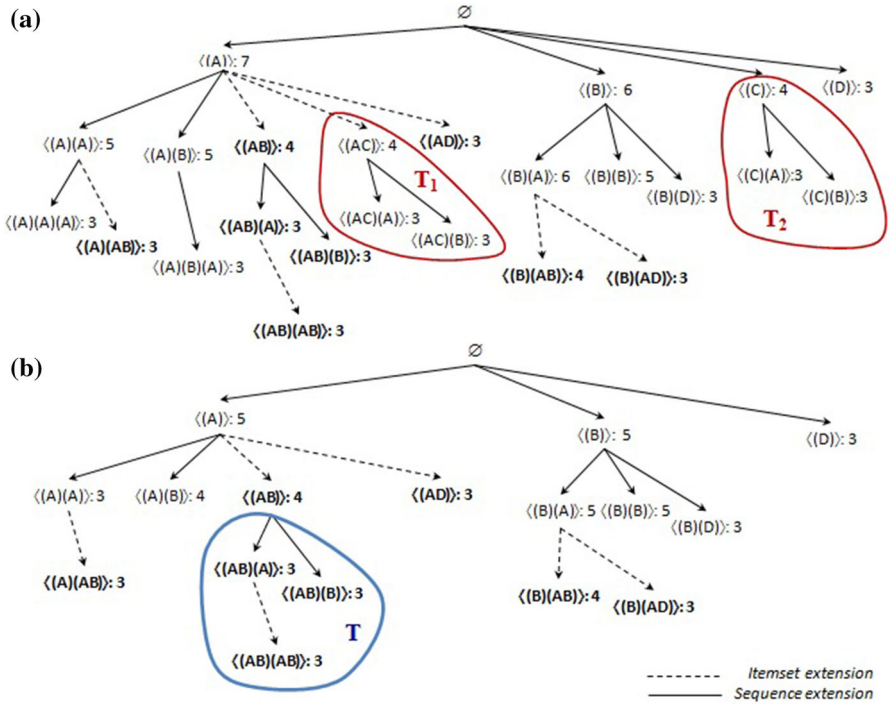

Fig. 2 **a** The prefix-tree contains all frequent candidates with their supports if we do not perform the transformation. **b** The prefix-tree if we perform the transformation

similar process for the remaining atoms, and the results are summarized in Table 6, we have $support(C) = 2 < minSup$, so $F_1 = \{A, B, D\}$.

If we do not perform the transformation, the search space is shown in Fig. 2a, otherwise it is reduced to that shown in Fig. 2b. Due to transformation, we have already pruned the patterns ⟨(A)(A)(A)⟩, ⟨(A)(B)(A)⟩, ⟨(AC)⟩, ⟨(AC)(A)⟩, ⟨(AC)(B)⟩, ⟨(C)⟩, ⟨(C)(A)⟩, and ⟨(C)(B)⟩ since their supports no longer satisfy the *minSup* .

Step 4 Check constraint and extend the patterns.

From step 2, $\mathbb{C} = \{(AB), (AD)\}$, because none of the atoms satisfy any constraint in set $\mathbb{C}$, we call the procedure *DBV-PREFIX-EXTENSION-CHECK()* for each atom in F_1. Consider the extension of atom A, we have ⟨(A)(A)⟩, ⟨(A)(B)⟩, ⟨(AB)⟩, and ⟨(AD)⟩ are frequent patterns. Since ⟨(A)(A)⟩ and ⟨(A)(B)⟩ do not satisfy any constraint, we recursively call procedure *DBV-PREFIX-EXTENSION-CHECK()*. But ⟨(AB)⟩ satisfies the constraint

Table 7 Parameters for the IBM data generator

C	Average itemsets per sequence
T	Average items per itemset
S	Average itemsets in maximal sequences
I	Average items in maximal sequences
N	Number of distinct items
D	Number of sequences

in set $\mathbb{C}$ then we just perform pattern extension without checking the constraint by calling procedure *DBV-PREFIX-EXTENSION()*. This means that all the sequences in T should not be checked. Fig. 2b shows the prefix-tree storing all the frequent candidates in the pattern growth and constraint check process, where the frequent satisfied-patterns are in bold. We have $FCP = \{\langle(A)(AB)\rangle : 3, \langle(AB)\rangle : 4, \langle(AB)(A)\rangle : 3, \langle(AB)(AB)\rangle : 3, \langle(AB)(B)\rangle : 3, \langle(AD)\rangle : 3, \langle(B)(AB)\rangle : 4, \langle(B)(AD)\rangle : 3\}$.

5 Experimental results

We perform a series of experiments to compare the performance of our proposed algorithms including MSPIC-Naive, MSPIC-DBV and PRISM algorithm, abbreviated as PRISM-IC, for mining sequential patterns with itemset constraints. Both MSPIC-Naive and MSPIC-DBV use the dynamic bit vector, but MSPIC-DBV applies two propositions, including the transformation for pruning and the constraint checking reduction. The experiments were implemented in Visual Studio 2008 C# and executed on a personal computer with an Intel Core i7 1.9-GHz CPU and 8GB of RAM running Windows 8.1. In order to record the true runtime of the algorithms, we turn off the output of frequent sequences during all the tests.

5.1 Experimental databases

We perform the experiments on two kinds of databases. The first kind is those databases for which itemsets are of size 1, such as web log databases and DNA sequence databases. The second kind is those databases for which the itemset size is greater than or equal to 1, such as databases of customer transactions.

We run tests on two real databases for the first kind, namely Gazelle and Kosarak, which were a part of the KDD Cup 2000 challenge dataset [13]. Gazelle contains 59,602 sequences (i.e., customers), 149,639 sessions, and 497 distinct page views. The average sequence length is 2.5, and the maximum sequence length is 267. Kosarak is a very large database, with 990,000 sequences of clickstream data from a news portal containing 41,270 distinct items. We thus use only a subset of 10,000 sequences with 10,000 distinct items.

For the second kind of databases, for which the itemset size is greater than or equal to 1, we use tests on two synthetic databases created using the synthetic data generator provided by IBM. These are C20T20S20I20N100D1k and C20T50S20I10N1kD100k [12]. The generator uses the parameters shown in Table 7.

5.2 Initialize constraint itemsets

To initialize the constraint itemsets, we define the selectivity of a constraint.

Table 8 Comparison of extracted pattern quantity on Gazelle

minSup	#FP	#*FCP*
1	510	268
0.9	640	400
0.8	807	468
0.7	1074	561
0.6	1485	815

For the databases in which all itemsets are of size 1, the selectivity of a constraint is defined as the ratio of the number of items (distinct values) which are selected to be the constraint against the total number of items N in the database. To ensure the selected ratio, the items selected to be the constraint are randomly obtained from F_1. For example, Gazelle has 497 distinct items, so 10% selectivity means ≈ 50 items are selected from F_1 to be the constraint. The number of constraint itemsets selected in Gazelle is also equal to the number of items selected, because every itemset in this database is only a single item.

For the databases in which the size of each itemset of a sequence is greater than or equal to 1, the selectivity of a constraint is defined as the ratio of the number of itemsets which are selected to be the constraint against the total number of itemsets ($D \times C$) in the database. The items in constraint itemsets are randomly obtained from F_1. For example, the C20T20N100D1k database has 100 distinct items and 1,000 sequences, with an average of 20 itemsets per sequence. The set $\mathbb{C}$ is randomly taken from F_1, with 10% selectivity meaning $|\mathbb{C}| = 10 \times 1000 \times 20/100 = 2000$, and the maximum size of the constraint itemset is 20.

5.3 Performance analysis

First, we perform the experiments for mining sequential patterns with and without itemset constraints. The results show that the number of patterns extracted in mining with constraint is lower than that seen when mining without constraint, because mining with constraint only returns those patterns which satisfy the constraint defined by the user. This subset is thus very interesting to the user. Table 8 shows a comparison of the extracted pattern quantity when mining without and with constraint on Gazelle with selectivity = 50%. We also obtain similar results for other databases.

We then carry out the experiments for mining with itemset constraints on each database by varying the value of the selectivity while fixing the *minSup*, and vice versa. In each case, all the three algorithms return the same quality and number of discovered patterns, but the runtimes are different. Since the difference between the approaches is very large, we show the results in two graphs for each database. Graph (*a*) presents a comparison of the execution times between PRISM-IC and MSPIC-Naive, and graph (*b*) shows that between MSPIC-Naive and MSPIC-DBV.

The experimental results show that both MSPIC-DBV and MSPIC-Naïve outperform PRISM-IC in execution time for both kinds of databases and that using a dynamic bit vector structure is more effective. Essentially, MPIC-DBV is faster than the other approaches.

As expected, we can see that the runtimes of the three algorithms increase significantly with decreasing *minSup*, and the difference between them becomes larger (Figs. 3, 4, 5, 6). The main reason is that, when *minSup* decreases there are many more frequent sequences, but MSPIC-DBV does not need to check the constraint for all candidates. Moreover, decreasing

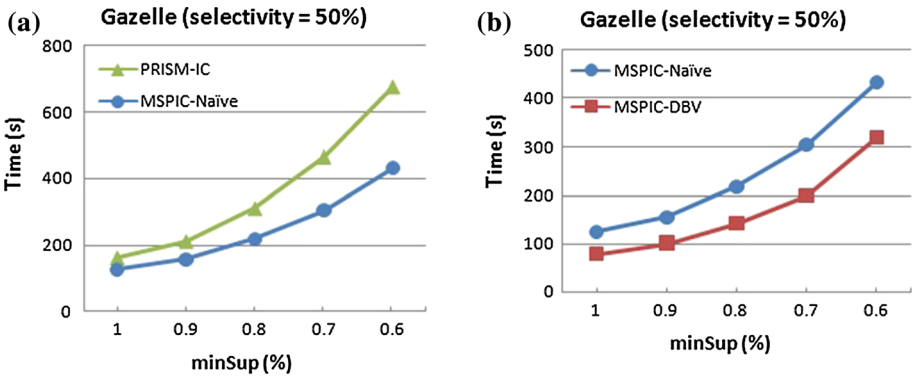

Fig. 3 Comparison of runtimes with various *minSup* values for Gazelle between **a** PRISM-IC and MSPIC-Naive, **b** MSPIC-Naïve and MSPIC-DBV

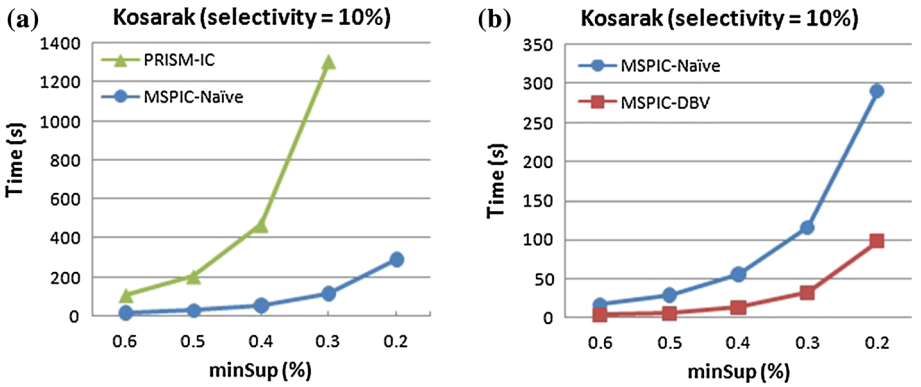

Fig. 4 Comparison of runtimes with various *minSup* values for Kosarak between **a** PRISM-IC and MSPIC-Naïve, **b** MSPIC-Naïve and MSPIC-DBV

minSup means that the candidates have to appear in fewer input sequences. As such, there are more opportunities for MSPIC-DBV to prune by using the transformation step.

Consider the Gazelle database with a selectivity of 50%, as shown in Fig. 3, with the *minSup* varying from 1 to 0.6%. At *minSup* 1%, the three algorithms are close in terms of runtime. However, the gaps between them become greater when decreasing *minSup*. At *minSup* of 0.6%, MSPIC-DBV is about 1.5 times faster than MSPIC-Naïve, and about 2.5 times faster than PRISM-IC. We also obtain similar results for the other databases in Fig. 4, 5, 6. The difference between the algorithms is very large at some *minSup* thresholds, and thus we do not show this in the chart. For an instance, in Fig. 5 for the database C20T20S20I20N100D1k, the runtime of PRIM-IC is over 7000*s* when *minSup* is less than 68%. At *minSup* 68%, MSPIC-DBV outperforms MSPIC-Naïve by about 1.5 times and PRISM-IC by up to 9 times.

On the other hand, when we fix the *minSup* and change the selectivity, the execution time increases along with the value of selectivity (Figs. 7, 8, 9, 10), as there more sequential patterns to be found. The gap between MSPIC-DBV and the other two algorithms is smaller, because the number of zero sequence blocks deleted by the transformation step is inversely proportional to the number of constraint itemsets. Hence, at a low selectivity value, MSPIC-DBV runs faster.

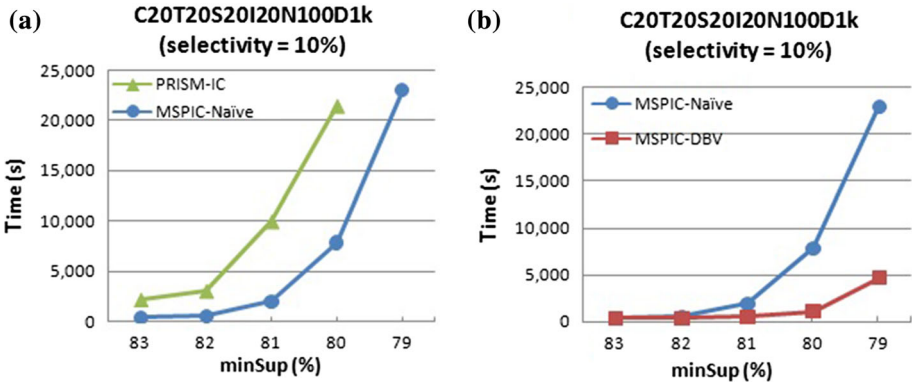

Fig. 5 Comparison of runtimes with various *minSup* values for C20T20S20I20N100D1k between **a** PRISM-IC and MSPIC-Naïve, **b** MSPIC-Naïve and MSPIC-DBV

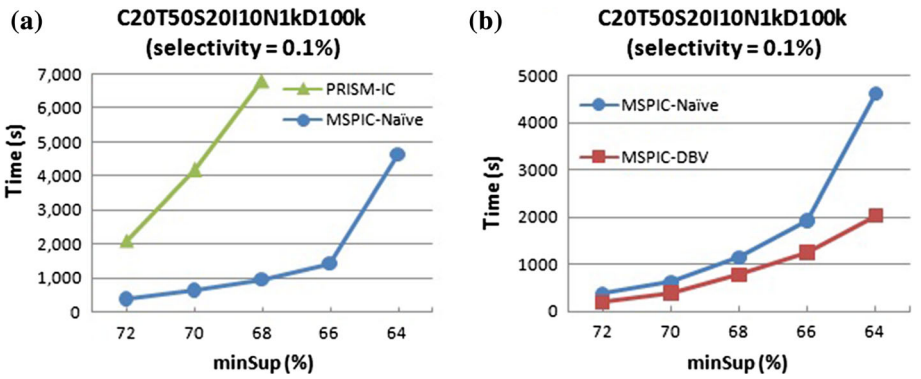

Fig. 6 Comparison of runtimes with various *minSup* values for C20T50S20I10N1kD100k between **a** PRISM-IC and MSPIC-Naïve, **b** MSPIC-Naïve and MSPIC-DBV

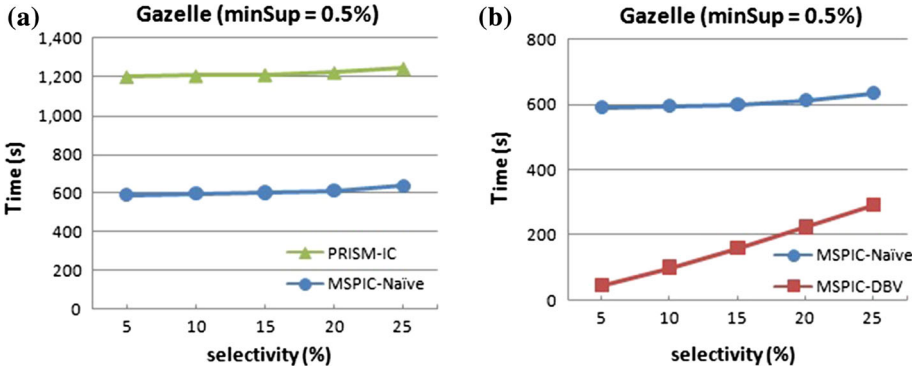

Fig. 7 Comparison of runtimes with various selectivity values for Gazelle between **a** PRISM-IC and MSPIC-Naïve, **b** MSPIC-Naïve and MSPIC-DBV

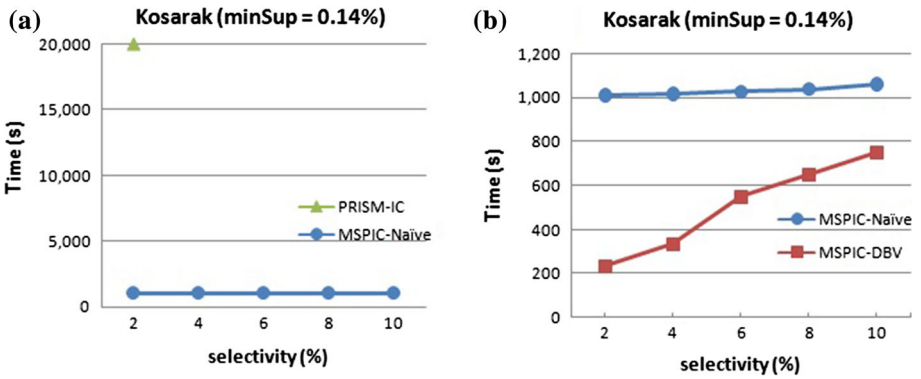

Fig. 8 Comparison of runtimes with various selectivity values for Kosarak between **a** PRISM-IC and MSPIC-Naïve, **b** MSPIC-Naïve and MSPIC-DBV

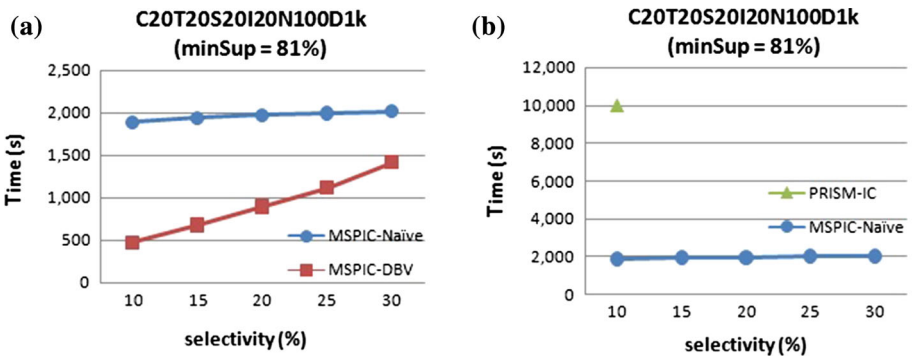

Fig. 9 Comparison of runtimes with various selectivity values for C20T20S20I20N100D1k between **a** PRISM-IC and MSPIC-Naïve, **b** MSPIC-Naïve and MSPIC-DBV

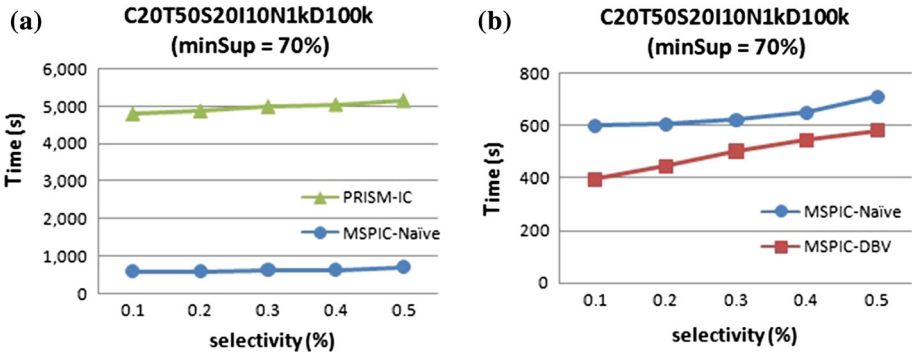

Fig. 10 Comparison of runtimes with various selectivity values for C20T50S20I10N1kD100k between **a** PRISM-IC and MSPIC-Naïve, **b** MSPIC-Naïve and MSPIC-DBV

Figure 7 shows how the runtime of the algorithms changes with various selectivities for Gazelle. We see that MSPIC-DBV outperforms the other two in most cases. At selectivity 25%, MSPIC-DBV is 2 times faster than MSPIC-Naïve and 4 times faster than PRISM-IC

but at selectivity 5% it is up to 13 times faster than MSPIC-Naïve and 26 times faster than PRISM-IC. We also have the same results with the Kosarak database and synthetic databases.

We can conclude that the primary reason why MSPIC-DBV performs well for all databases is due to avoiding checking the constraint for all candidates, early pruning search space and reducing cost of manipulations by the DBV transformation step.

6 Conclusions and future work

In this paper, we have proposed the MSPIC-DBV algorithm, an efficient algorithm for mining sequential patterns with itemset constraints. The MSPIC-DBV algorithm uses a new data structure, called dynamic bit vector to compress candidate information. In particular, it has the following two propositions: one for pruning the search space via the use of prefixes and subsequences both at the beginning and during the mining process, and reducing the calculations needed for pattern extension; and one for skipping the checking constraint step for some candidates. The experimental results show that MSPIC-DBV is more efficient than MSPIC-Naïve and PRISM-IC in all cases examined in this work.

For future work, we will apply this approach to mining sequential patterns with super-pattern constraints and with incorporating constraints. We will also apply our technique to mining frequent sequential closed patterns with constraints.

Acknowledgements This research is funded by Vietnam National Foundation for Science and Technology Development (NAFOSTED) under Grant Number 102.05-2015.07.

References

1. Agrawal R, Srikant R (1995) Mining sequential patterns. In: The 11th international conference on data engineering, pp 3–14
2. Ayres J, Gehrke JE, Yiu T, Flannick J (2002) Sequential pattern mining using a bitmap representation. In: The 8th ACM SIGKDD international conference on knowledge discovery and data mining, pp 429–435
3. Chen E, Cao H, Li Q, Qian T (2008) Efficient strategies for tough aggregate constraint-based sequential pattern mining. Inf Sci 176(1):1498–1518
4. Chen YL, Hu YH (2006) Constraint-based sequential pattern mining: the consideration of recency and compactness. Decis Support Syst 42(2):1203–1215
5. Chen J, Gu J, Yang, Qiao Z (2010) Efficient strategies for average constraint-based sequential pattern mining. In: The 2010 international conference on multimedia communications, pp 254–257
6. de Amo Sandra, Furtado DA (2007) First-order temporal pattern mining with regular expression constraints. Data Knowl Eng 62(3):401–420
7. Fumarola F Pasqua, Fabiana Lanotte PF, Ceci M, Malerba D (2016) CloFAST: closed sequential pattern mining using sparse and vertical id-lists. Knowl Inf Syst 48(2):429–463
8. Garofalakis MN, Rastogi R, Shim K (1999) SPIRIT: Sequential pattern mining with regular expression constraints. In: The 25th international conference on very large data bases, pp 7–10
9. Gouda K, Hassaan M, Zaki MJ (2010) Prism: a primal-encoding approach for frequent sequence mining. J Comput Syst Sci 76(1):88–102
10. Han J, Pei J, Mortazavi-Asl B, Chen Q, Dayal U, Hsu M-C Freespan (2000) Frequent pattern projected sequential pattern mining. In: The 6th ACM SIGKDD international conference on knowledge discovery and data mining, pp 355–359
11. Ho J, Lukov L, Chawla S (2005) Sequential pattern mining with constraints on large protein databases. In: The 12th international conference on management of data (COMAD 2005), pp 89–100
12. https://www.mediafire.com/folder/nebwi57vp4gjw/Synthetic_DB
13. Kohavi R, Brodley C, Frasca B, Mason L, Zheng Z (2000) KDD-Cup 2000 organizers' report: peeling the onion. SIGKDD Explor 2(2):86–98

14. Le B, Tran MT, Vo B (2015) Mining frequent closed inter-sequence patterns efficiently using dynamic bit vectors. Appl Intell 43(1):74–84
15. Liao VCC, Chen MS (2014) DFSP: a depth-first spelling algorithm for sequential pattern mining of biological sequences. Knowl Inf Syst 38(3):623–639
16. Lin MY, Lee SY (2005) Efficient mining of sequential patterns with time constraints by delimited pattern growth. Knowl Inf Syst 7(4):499–514
17. Lo D, Khoo SC, Li, J: Mining and ranking generators of sequential patterns. In: The 9th SIAM international conference on data mining, pp 553–564 (2008)
18. Mallick B, Garg D, Grover PS (2014) Constraint-based sequential pattern mining: a pattern growth algorithm incorporating compactness, length and monetary. Int Arab J Inf Technol 11(1):33–42
19. Masseglia F, Poncelet P, Teisseire M (2009) Efficient mining of sequential patterns with time constraints: reducing the combinations. Expert Syst Appl 36(2):2677–2690
20. Orlando S, Perego R, Silvestri C (2004) A new algorithm for gap constrained sequence mining. In: The 2004 ACM symposium on applied computing, pp 540–547
21. Orlando S, Perego R, Silvestri C (2004) A new algorithm for gap constrained sequence mining. In: The ACM symposium on applied computing (SAC), pp 540–547
22. Pei J et al (2004) Mining sequential patterns by pattern-growth: the PrefixSpan approach. IEEE Trans Knowl Eng 16(11):1424–1440
23. Pei J, Han J, Wang W (2007) Constraint-based sequential pattern mining: the pattern-growth methods. J Intell Inf Syst 28(2):133–160
24. Pokou JM, Fournier-Viger P, Moghrabi C (2016) Authorship attribution using small sets of frequent part-of-speech skip-grams. In: The international Florida artificial intelligence research society conference, pp 86–91
25. Senkul P, Salin S (2012) Improving pattern quality in web usage mining by using semantic information. Knowl Inf Syst 30(3):527–541
26. Srikant R, Agrawal R (1996) Mining sequential patterns: generalizations and performance improvements. In: The 5th international conference on extending database technology, pp 3–17
27. Tran MT, Le B, Vo B (2015) Combination of dynamic bit vectors and transaction information for mining frequent closed sequences efficiently. Eng Appl Artif Intell 38:183–189
28. Tran MT, Le B, Vo B, Hong TP (2016) Mining non-redundant sequential rules with dynamic bit vectors and pruning techniques. Appl Intell 45(2):333–342
29. Tsai CY, Lai BH (2015) A location-item-time sequential pattern mining algorithm for route recommendation. Knowl Based Syst 73:97–110
30. Van TT, Vo B, Le B (2014) IMSR_PreTree: an improved algorithm for mining sequential rules based on the prefix-tree. Vietnam J Comput Sci 1(2):97–105
31. Vo B, Hong TP, Le B (2012) DBV-miner: a dynamic bit-vector approach for fast mining frequent closed itemsets. Expert Syst Appl 39(8):7196–7206
32. Vo B, Tran MT, Nguyen H, Hong TP, Le B (2012) A dynamic bit-vector approach for efficiently mining inter-sequence patterns. In: 2012 third international conference on innovations in bio-inspired computing and applications (IBICA), pp 51–56
33. Yen SJ, Lee YS (2004) Mining sequential patterns with item constraints. In: Data warehousing and knowledge discovery, pp 381–390
34. Yun U, Ryu KH (2010) Discovering important sequential patterns with length-decreasing weighted support constraints. Int J Inf Technol Decis Mak 9(4):575–599
35. Zaki MJ (2000) Sequence mining in categorical domains: incorporating constraints. In: The 9th international conference on information and knowledge management. ACM, pp 422–429
36. Zaki MJ (2000) SPADE: an efficient algorithm for mining frequent sequences. Mach Learn J 42(1/2):31–60
37. Zhang J, Wang Y, Yang D (2015) CCSpan: mining closed contiguous sequential patterns. Knowl Based Syst 89:1–13
38. Zhang J, Wang Y, Zhang C, Shi Y (2016) Mining contiguous sequential generators in biological sequences. IEEE/ACM Trans Comput Biol Bioinform 13(5):855–867

Trang Van received her BSc in Information Technology from Qui Nhon University, Qui Nhon city, Vietnam, in 2007 and received her MSc degree in Information Systems from VNUHCM, University of Science, in 2011. She is currently a PhD Student at University of Information Technology, VNUHCM, Ho Chi Minh City, Vietnam. Her research interests include association rule, sequential rule and sequential pattern mining.

Bay Vo received his BSc, MSc and PhD degrees in Computer Science from the University of Science, VNUHCM, Ho Chi Minh City, Vietnam, in 2002, 2005 and 2011, respectively. He is currently an Associate Professor and Dean of Faculty of Information Technology, Ho Chi Minh City University of Technology, Vietnam. His research interests include association rules, classification, mining in incremental database, distributed databases and privacy preserving in data mining.

Bac Le is currently an Associate Professor and Vice-Dean of Faculty of Information Technology, and Head of Department of Computer Science, University of Science, VNUHCM, Ho Chi Minh City, Vietnam. His main research includes artificial intelligence, soft computing, data mining and data science.