

# Indexed Bit Map (IBM) for Mining Frequent Sequences

Lionel Savary<sup>1</sup>, Karine Zeitouni<sup>1</sup>

<sup>1</sup> PRiSM Laboratory, 45 Avenue des Etats-Unis,  
78035 Versailles, France  
{Lionel.Savary, Karine.Zeitouni}@prism.uvsq.fr

**Abstract.** Sequential pattern mining has been an emerging problem in data mining. In this paper, we propose a new algorithm for mining frequent sequences. It processes only one scan of the database thanks to an indexed structure associated to a bit map representation. Thus, it allows a fast data access and a compact storage in main memory. The experimental results show the efficiency of our method compared to existing algorithms. It has been tested on synthetic data and on real data containing sequences of activities of a urban population time-use survey.

## 1 Introduction

The problem of mining sequential patterns was first introduced in the context of customer transactions analysis [2]. It aims to retrieve frequent patterns in the sequences of products purchased by customers through time ordered transactions. Several algorithms have been proposed in order to improve the performances and to reduce required space in memory [5], [9], [6]. Other works have concerned mining frequent sequences in DNA [8] or Web Usage Mining [3]. Finally, notice the use of bit map structure in providing a compact representation and good performances.

The target application in this paper is related to population time-use analysis and more precisely their daily displacements [4]. Our data are related to daily activities carried out by each surveyed person at the scale of a whole urban area. Thus, for each person of a surveyed household, it captures the activity program [7], the transport mode used between two activities, the departure time, and the duration of the trip. For example, during a day, an individual can leave home, take children to school, go to work, pick children up from school, and come back home. Activity programs of most individuals may be the same or be similar. Each activity program could be seen as a sequence of single values, making it possible to discover frequent activity sequences that characterise groups of the surveyed individuals. This allows analyzing the mobility of this urban population. Likewise, when considering transport mode, schedules or duration sequences, it would be possible to determine a typology of used transport modes, schedules, and so on.

Existing algorithms are either inappropriate or not enough efficient to our specific case. Most works [1], [2], [6] make multiple scan of the database, which can be considered as the main bottleneck of algorithms of frequent sequence mining. Furthermore, unlike the analysis of sequential transactions where each transaction is an item set, our context only focuses on the analysis of sequences of items.

Although existing works [9], [10], [12] can be applied in this context, we propose here a new algorithm more appropriate to this particular case. This algorithm only makes one scan of the database. The indexed bit map structure needs few spaces in the main memory and allows a fast access to the data. The experimental results, using real or synthetic data, show that our algorithm outperforms existing ones.

The paper is organised as follows: section 2 presents related works, then, section 3 describes the proposed algorithm, section 4 proposes an optimisation, section 5 relates the experimentation and performance study, and finally, a general conclusion summarizes our contribution and traces some perspectives.

## 2 Related works

Most works related to mining frequent sequences are in the field of customer transaction analysis. Early work on frequent patterns -*Apriori* algorithm- only considered transactions, not sequence of transactions [1]. This algorithm is costly because it carries out multiple scans of the database to determine frequent subsets of items. Three algorithms dealing with sequence of transactions are presented and compared in [2]: *AprioriAll*, *AprioriSome* and *DynamicSome*. *AprioriAll* algorithm is an adaptation of *Apriori* to sequences where candidate generation and support are computed differently. *AprioriAll*, and *AprioriSome* only compute maximal frequent sequences. Their principle is to jump to candidates of size  $k+next(k)$  in the next scan, where  $next(k)>1$ . Maximum frequent sequences of lower size that have not been calculated are given in the backward phase. The value of  $next(k)$  increases with  $P_k = |L_k|/|C_k|$ , where  $L_k$  stands for frequent sequences of size  $k$ , and  $C_k$  the whole generated candidates of size  $k$ . *DynamicSome* algorithm is based on *AprioriSome* but uses a jump by a multiple of user defined *step*.

*SPAM* algorithm [5] uses a bitmap representation of transaction sequences once the entire database has been loaded in a lexicographic tree. But this algorithm considers that the entire database and all used data structures should completely fit into main memory, and then do not adapt for large datasets.

The *GSP* algorithm [6] exploits the property that all contiguous subsequences of a frequent sequence also have to be frequent. As *Apriori*, it generates frequent sequences, then candidate sequences by adding one or more items.

*PrefixSpan* [10] first finds the frequent items after scanning the database once. The sequence database is then projected, according to the frequent items, into several smaller databases. Finally, all sequential patterns are found by recursively growing subsequence fragments in each projected database. Employing a divide-and-conquer strategy with the *PatternGrowth* methodology, *PrefixSpan* efficiently mines the complete set of patterns.

### 3 IBM algorithm

We are now going to focus on the specific case where the considered sequences are basic since they are composed of single items, not of a set of items. This is the case in DNA [8], Web usage data [3] or activity program sequences [7]. Our algorithm will be compared to PrefixSpan, one of the most efficient among the above mentioned methods.

A sequence is said frequent if it is included in a number of sequences greater than a support given by the user. The inclusion between two sequences  $s1 = (a_1, \dots, a_n)$  and  $s2 = (b_1, \dots, b_n)$ :  $s1 \subset s2$  is defined by:  $\exists b_{i1} = a_1, \dots, b_{in} = a_n$  such that  $i1 < i2 < \dots < in$ .

#### 3.1 Principle of the algorithm

The proposed approach is two phases. The first stage is the data encoding into a memory resident data structures. The second one is the frequent generation that in turn is composed of candidate generation, and candidate support checking.

The data structure is based on four components: (i) a Bit Map is a binary matrix representing the distinct sequences of the database, (ii) an SV vector encodes all the ordered combinations of sequences, (iii) an index (INDEX) on the Bit Map allows a direct access to sequences according to their size, (iv) an NB table associated to the Bit Map which informs about the frequency of each distinct sequence (figure 1).

This algorithm only makes one scan of the database during which the total number of distinct sequences, the frequency of these sequences and the number of sequence by size are computed. This allows computing the support of each generated sequence. These sequences are classified by decreasing size in the IBM and only distinct sequences are stored in the Bit Map. An index by size allows a direct access to sequences according to their size. This structure provides an optimisation since a generated sequence  $s$  of size  $t$  will be directly compared with the sequences of the same or greater size stored in the IBM (figure 1).

In order to simplify the notations, we represent each activity by a specific character, e.g. HSWSH standing for (Home, School, Work, School, Home). In the figure 1, the sequence vector (SV) is made of 5 ordered activities (H,W,S,M,H). In this example one supposes that the database is composed of six distinct sequences of size 1 to 5 encoded in the IBM. The bit  $1$  indicates the items present in the sequence according to the SV and bit  $0$ , those that are not. Here, there are 6 distinct sequences: (H), (W), (HRS), (HSH), (HSMH) and (HSRWH). In the above example (figure 1), each cell of the INDEX indicates the first line where the corresponding size of sequence is stored. For example, the cell number 5 (with value 6) corresponds to the line number 6 of the first sequence of size 5 encoded in the IBM. The table NB associates to the IBM stores the frequency of each distinct sequence. Thus the sequence (HSMH) of size 4 occurs 20 times in the database. In this algorithm, INDEX, SV, NB and IBM are built on the fly during one pass. At each insertion of a sequence, the IBM may become larger, and a set of shifting operations are applied to the bit values stored in this table.

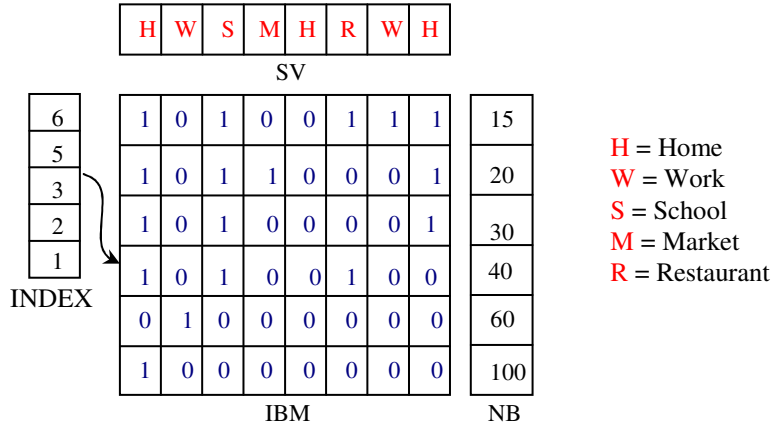


Fig. 1. The data structure

```

IBM (sequence database DB, threshold t)
00 For each sequence s in DB
01   Gen-sequence-vector (s)
02   Encode and Insert s in the IBM
03   Update NB
04   Update Index
05 End For
06 Integer k := 1;
07 While exists frequent sequence of size k
08   k := k+1;
09   Generate Ck
10   Gen-frequent-sequences (t)
11 End While

```

Fig. 2. IBM algorithm

Figure 2 shows the general IBM algorithm that takes as parameters: the database of sequences DB and a threshold t. This value (t) stands for the minimum frequency of the sequences which will be taken into account for the generation of the candidates. Then for each sequence, it reads from the database during the scan, the SV (line 01) is generated using a merging process (see section 3.2). If the sequence already exists in SV, only the NB table is updated (line 03): the line corresponding to this sequence in NB (and encoded in the IBM) is incremented. So, the frequency corresponding to this value is incremented. Else, if the sequence is not presented in SV, it is generated by the Gen-sequence-vector(s) function (section 3.2). The height of the IBM is increased to one line (line 02), the length is increased to the SV length, and the INDEX (line 04) is updated. Then, a set of shifting operations is applied to the IBM in order to preserve the initial values of existing sequences while encoding the new one.

Once all the data have been encoded in this structure (SV, IBM, NB, INDEX), new candidates (line 09) are generated (see section 3.3) and compared to the data stored in the IBM (line 10) with a fast access thanks to the index (INDEX).

### 3.2 Generation of the sequence vector

The sequence vector is generated during the unique scan of the database according to the algorithm of figure 3. Here,  $s$  stands for a sequence of the database read during the scan, and  $\text{position}(x)$  stands for the cell number of value  $x$  in the SV. If an item  $a$  of  $s$  already exists in SV, then there is nothing to do, otherwise, there are two possibilities: if there exists an item  $b$  such that the cell number of  $b$  is greater than the cell number of  $a$  and  $b$  is in SV (line 04 and 05), then  $a$  is inserted before the value  $b$  in SV; otherwise,  $a$  is inserted at the end of SV (line 06). Thus all the distinct sequences of the database are represented in the SV using a merging process.

```

Gen-sequence-vector( $s$ ):
00 var SV :=  $\emptyset$ ; {SV empty at the beginning};
01 Integer current_position := 0; {position mark in SV};
02 For each item  $a$  of  $s$ 
03   If  $a \notin$  SV
04     If  $\exists b \in s$  such that ( $b \in$  SV and  $\text{position}(b) >$ 
 $\text{position}(a)$  in  $s$  and  $\text{position}(b) >$  current_position
in SV)
05       Insert  $a$  before  $b$ 
06     Else insert  $a$  at the end of SV
07       current_position :=  $\text{position}(a)$  in SV;
08 End For

```

**Fig. 3.** Sequence Vector generation

### 3.3 Candidate generation

During the scan, the frequencies of all items are computed. Those whose support is underneath the one specified by the user are deleted. Then, candidates are generated from these frequent items, using the fusion process as in GSP algorithm [6].

### 3.4 Candidate support counting

For a given candidate  $C$  of size  $S$ , the algorithm first accesses the first sequence of size  $S$  encoded in IBM, which corresponds to the line  $l = \text{INDEX}(S)$ . For each line starting from the line  $l$  to the last line of IBM table, the algorithm determines using the SV vector if  $C$  is contained in each line of IBM. If so, the corresponding frequency of

this sequence stored in the NB table, is added to the frequency of the candidate. After the comparison with each line until the last one, the support of C is computed.

## 4 Implementation and optimization

The IBM algorithm proposed here takes few spaces in the main memory. But whereas the bit variable is not provided in programming languages like Java, C++, some shifting operations are required to access the target value stored in the bit map and corresponding to the value stored in SV. In order to avoid these superfluous computations, we have proposed the IBM2 algorithm, where the bit map is replaced by a Boolean matrix, i.e. where cells are declared of Boolean type, which takes 8 bits for each cell. Although this solution requires more space in memory, the access to the target value stored in the Boolean matrix is done directly without shifting computations. The result of their respective performances is detailed in the next section and compared with PrefixSpan.

## 5 Experimental results

The experiments were performed on a 2.5Ghz Pentium IV with 1.5 GB of memory running Microsoft Windows XP Professional, with three different sizes of datasets: 100000, 300000, 600000 and 1000000 rows. Items and the size of the sequences have been randomly generated for the experimentations. The size of sequences is randomly generated from 2 to 60, and the number of distinct items is about 10 for figures 4 to 7. Nevertheless, this number has been pushed to 35 distinct items in order to test the impact on the algorithm efficiency. For this experimentation, we have used the package PrefixSpan-0.4.tar.gz<sup>1</sup>. Compared to IBM and IBM2 that have been implemented in Java, PrefixSpan has been implemented in C++ - *a priori* more optimal than Java -. The experimentations show that the larger is the database size, the more IBM and IBM2 outperform PrefixSpan (Figures 4 to 7). This is because IBM and IBM2 make only one scan of the database and use an Indexed Bit Map structure which perform faster access to the sequences than the tree structures used in PrefixSpan. The size of the bit map also depends on the size of SV, which also increases with the number of distinct sequences. Notice that SV size does not depend on the size of the database itself. In fact, it only increases when the encountered sequence can not be encoded using the current SV. Moreover, not all the items of the inserted sequence are added in SV, but only those that are not present in the same order.

---

<sup>1</sup> <http://chasen.org/~taku/software/prefixspan/>

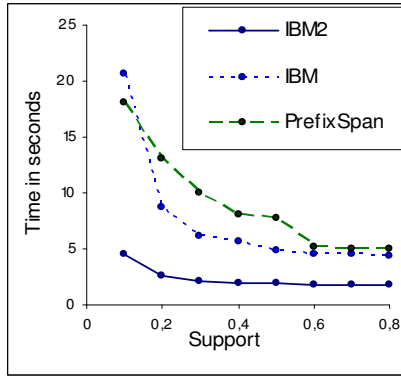


Fig. 4. Performances with 100,000 rows

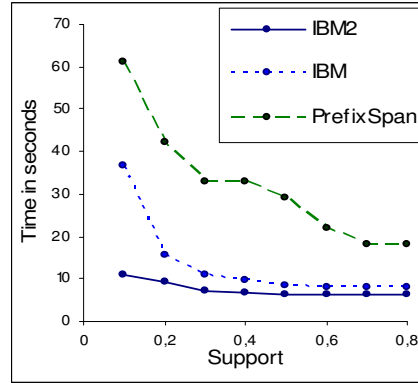


Fig. 5. Performances with 300,000 rows

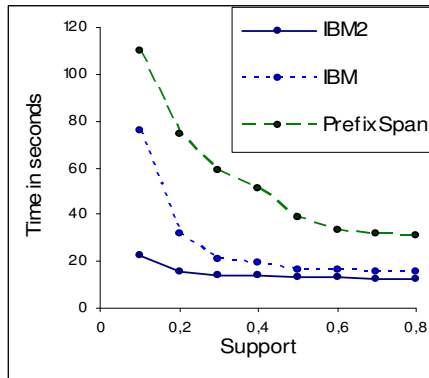


Fig. 6. Performances with 600,000 rows

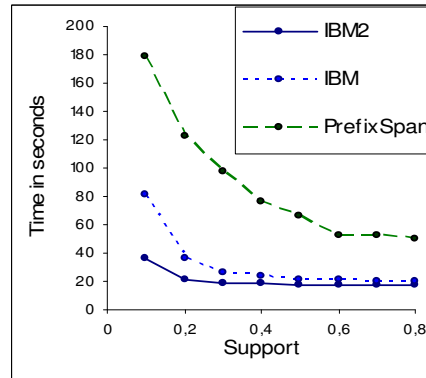


Fig. 7. Performances with 1,000,000 rows

Finally, since the probability to find common ordered items between SV and the current sequence becomes high as the building process advances, SV size becomes stable regardless of the size of the database. For instance, with a database composed of 600,000 rows, SV contains about 265 values for 90,000 distinct rows. The size of the Boolean Map is then equal to:  $265 \times 90,000 = 23.85$  Mega Bytes (a Boolean is encoded on 8 bits). As IBM is 8 times more compact, the size of the Bit Map is less than 3 MB. With 1,000,000 rows (figure 7), SV contains 370 elements for 160,000 distinct rows. Then, the size of the Boolean Map reaches 59.2 MB, whereas the size of the Bit Map fits in 7.5 MB. These results show that IBM is more appropriate than IBM2 for very large databases, due to data compression. However, IBM2 runs faster than IBM. This is due to the costs of shifting operations necessary to access target values, while IBM2 directly accesses the target sequences. For 100,000 rows until 20 distinct items, IBM and IBM2 perform better than PrefixSpan. Between 20 and 35 distinct items, IBM2 performs better than PrefixSpan, which becomes faster than IBM. But above 35 distinct items, PrefixSpan is faster than IBM and IBM2.

## 6 Conclusion and perspectives

This paper has presented a new algorithm IBM and its variant IBM2. The aim of this algorithm is to find all frequent sequences in item sequences. It has been applied to discover all frequent activity sequences in the time use mobility database within an urban environment. IBM only makes one scan of the database and provides an efficient data structure saving runtime and memory space. The use of the specified index provides another optimization of comparisons during candidate counting. Experimental results show that in most cases, IBM2 outperforms IBM, which in turn outperforms PrefixSpan for large and very large databases, with limited distinct items. Extensive experiments have been conducted that attest for the effectiveness and the efficiency of the proposed method, and are detailed in [11]. In perspective, IBM will be extended to multidimensional sequences (e.g. with attributes) and spatial sequences (such as trajectories). Other application fields will be explored, like pattern mining from DNA, Web Usage Mining or extension to customer transaction analysis. Finally, the proposed data structure adapts to similarity analysis of sequences and may be a good basis for efficient sequence clustering

## References

1. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules. In Proc. of the 20th Int. Conf. Very Large Data Bases (VLDB), Santiago, Chile, September (1994)
2. Agrawal, R., Srikant, R.: Mining sequential patterns. In Proc. of the 11th Int'l Conference on Data Engineering, Taipei, Taiwan, March (1995)
3. Spiliopoulou, M., Faulstich Lukas, C., Winkler, K.: A data miner analyzing the navigational behaviour of web users. In Proc. Of the Workshop on Machine Learning in User Modelling of the ACAI'99 Int. Conf., Creta, Greece, July (1999)
4. Ministère de l'Équipement, des Transports et du Logement. L'enquête ménages déplacements « méthode standard ». Collections du Certu. Octobre (1998), ISSN 1263-3313
5. Jay, A., Johannes, G., Tomi, Y., Jason F.: Sequential Pattern Mining using A Bitmap Representation. SIGMOD pp 429-435, July (2002), Edmonton, Alberta, Canada
6. Srikant, R., Agrawal, R.: Mining Sequential Patterns : Generalizations and Performance Improvements. Proc. 5th EDBT, Mars 25-29, (1996). Avignon, France. pp 3-17
7. Wang, D., Tao, C.: A spatio-temporal data model for activity-based transport demand modeling.. International Journal of Geographical Information Science, (2001), 15( 6), pp 561-585
8. Han, J., Jamil, H. M., Lu, Y., Chen, L., Liao, Y., Pei, J.: DNA Miner: A system prototype for mining DNA sequences. In the proc. Of the ACM SIGMOD International Conference on the management of data, Day 21-24,( 2001), Santa Barbara, CA, USA
9. Zaki, M. J.: Efficient Enumeration of Frequent Sequences. Int. Conference on Information and Knowledge Management, November( 1998), Washington DC
10. Pei, J., Han, J., Mortazavi-Asl, B., and H., Pinto.: Prefixspan: Mining sequential patterns efficiency by prefix-projected pattern growth. In Proc. of the International Conference on Data Engineering (ICDE), pp 215–224, (2001)
11. Savary, L., Zeitouni, K.: Indexed Bit Map (IBM) for Mining Frequent Sequences. Technical Report. Versailles University, France. PRISM Laboratory Report N° 2005/82, August (2005). <http://www.prism.uvsq.fr/rappports/bin/bibliography.php?id=300>