Indexed Bit Map (IBM) for Mining Frequent Sequences

Lionel Savary, Karine Zeitouni

Indexed Bit Map (IBM) for Mining Frequent Sequences

Lionel Savary, Karine Zeitouni
PRiSM Laboratory, 45 Avenue des Etats-Unis, 78035 Versailles cedex
{Lionel.Savary, Karine.Zeitouni}@prism.uvsq.fr

Résumé

La recherche de séquences fréquentes est un des problèmes récemment étudié en fouille de données. Dans cet article, nous proposons une nouvelle méthodologie pour la découverte de séquences fréquentes. L'algorithme proposé recherche des motifs fréquents en respectant l'ordre des articles. Ses performances ont été optimisées grâce aux structures et aux index proposés. En effet, il n'effectue qu'une seule lecture dans la base de données tout en ayant une représentation des données peu gourmande en mémoire et performante lors de la recherche de sous-séquences fréquentes. Les résultats expérimentaux démontrent l'efficacité de notre algorithme par rapport aux algorithmes existants. Il a été testé dans le contexte d'analyse de séquences d'activités de la population dans le cadre d'une enquête sur la mobilité urbaine. Les résultats expérimentaux démontrent l'efficacité de notre méthode comparée aux algorithmes existant.

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. This algorithm has been applied to activity sequences belonging to a population time-use survey. The experimental results show the efficiency of our method compared to existing algorithms.

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 [4], [14], [11]. Other works have concerned mining frequent sequences in DNA [6] or text mining [3]. Moreover, some association rules algorithms use a bit map structure [5] that requires few space in main memory, and gives good performances.

The target application in this paper is related to population time-use analysis and more precisely their daily displacements [8]. The database describes the 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 [12], 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 characterize groups of the surveyed individuals. This allows analyzing the mobility of this urban population. Likewise, when considering transport mode, schedule 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, 3] 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 [14, 9, 7, 5] 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 show that our algorithm outperforms existing ones. The associated index structure directly accesses sequences over a given size n, without wasting time accessing sequences of size $k < n$. This also allows the user to specify such size constraint and avoids producing uninteresting patterns. Specifying size constraint is particularly useful in our application field where the size

of an activity sequence of an individual is characteristic of his profile.

The paper is organized as follows: section 2 presents related works, then, section 3 describes our data structure and the proposed algorithms. Section 4 presents their performance evaluation and experimental results: a cost analysis is given, the performances of the proposed algorithms are compared with those from related works and a discussion studies the extreme cases and highlights the advantages of our approaches. Finally section 5 gives a general conclusion summarizes our contribution and traces some perspectives.

2. Relate 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 Pk = |Lk|/|Ck|, where Lk stands for frequent sequences of size k, and Ck 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 [4] uses a bitmap representation of transaction sequences once the entire database has been loaded in a lexicographic tree. The disadvantage in this algorithm is that the entire database and all used data structures should completely fit into main memory. Indeed, this algorithm makes one scan of the database to load it in memory at the condition that there is enough space.

The *GSP* algorithm [11] 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. *GSP* makes multiple scans over the data because the source dataset is scanned to evaluate the support of candidates.

PrefixSpan [9] 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 as in the transaction sequences mentioned above. We believe this is the case of many applications related to tempral events or DNA sequences. Our application goal is to find frequent sequences in activity programs. This is performed by seeking chains of activities (or transport modes, or schedules, …) that characterize a group of individuals. Our algorithm will be compared to PrefixSpan and SPAM, the two most efficients 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, .., a_n) and s2 = (b_1, …, b_n): s1 $\subset$ s2 is defined by :
$\exists\ b_{i1} = a_1, \ldots, b_{in} = a_n$ such that $i1 < i2 < \ldots < in$.

3.1. Principle of the algorithm

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

The algorithm is based on four data structures:

1. A Bit Map is a binary matrix representing the distinct sequences of the database,
2. An SV vector encodes all the ordered combinations of sequences,
3. An index (INDEX) on the Bit Map allows a direct access to sequences according to their size,
4. An NB table associated to the Bit Map which informs about the frequency of each distinct sequences.

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 upper size stored in the IBM (figure 1 and 2).

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 three distinct sequences of size 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 3 distinct sequences: (HWH), (HSH), (HSMH).

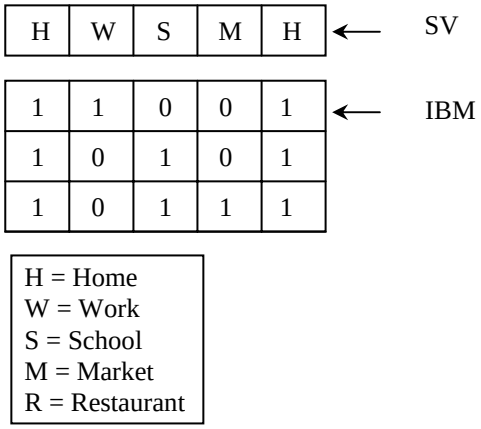

Figure 1. Indexed bit Map

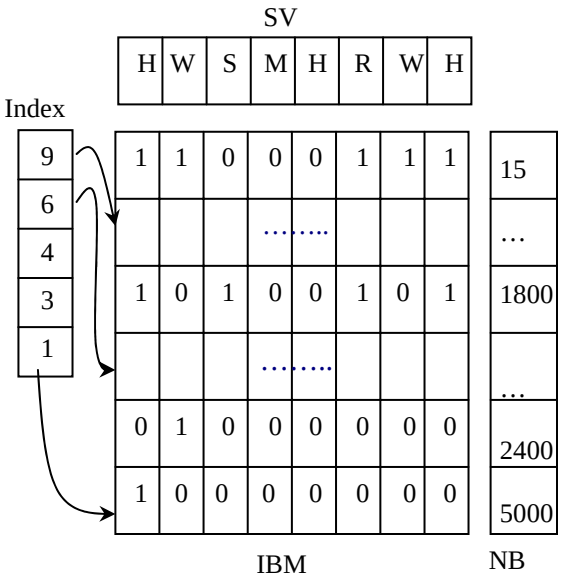

Figure 2. The data structure

In the example above (figure 2), IBM is composed of the whole distinct sequences of the database of size 1 to 5. 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 9) corresponds to the line number 9 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 (HWRWH) of size 5 has a frequency of 15 in the database.

In this algorithm, Index, SV, the NB table and IBM are built on the fly during one pass. At each insertion of a sequence, the IBM may increase in size, and a set of shifting operations are applied to the bit values stored in this table.

IBM (sequence database DB, threshold t**)**
00 **For** each sequence s in DB
01 - Gen-sequence-vector (s) //generates
// sequence vector
02 - Encode and Insert s in the IBM
03 - Update NB
04 - Update Index
05 **End For**
06 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**

Figure 3. IBM algorithm

Figure 3 shows the general IBM algorithm that takes as parameters: the batabase 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.

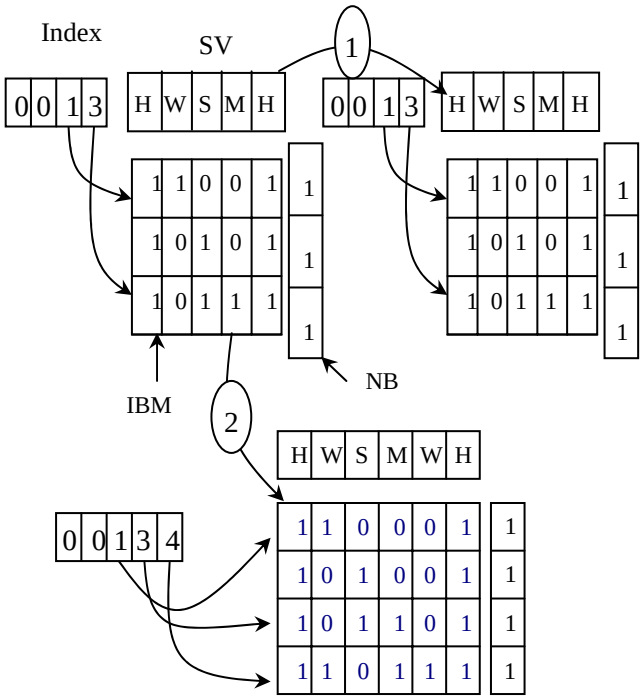

Figure 4. IBM generation

In the example of Figure 4, the original IBM is the same as the one in figure 1 with the NB and the Index tables in addition. Here, the sequences (HWH), (HSH) and (HSMH) have respectively the frequency 15, 10 and 12 in the database. The arrow with value 1 shows an insertion of a sequence (HSH). This operation does not change the values in SV and IBM since this sequence already exists, but it increments the frequency corresponding to this sequence in the NB table. The arrow with value 2 shows an insertion of the sequence (HWMWH). This operation modifies the SV vector (section 3.2) and a set of shifting operation is applied to the IBM in order to correspond to the values stored in the SV, to preserve the existing encoded sequences and a new line is added for the new sequence. Then the frequency of this new sequence is set to one.

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 5. Here, *s* stands for a sequence of the database read during the scan, and 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: (i) 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. (ii)Else, *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 - SV = ϕ //SV empty at // the beginning
01 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 position(*b*) > position(*a*) in s and position(b) > current_position in SV
05 - Insert *a* before *b*
06 - Else insert *a* at the end of SV
07 current_position = position(*a*) in SV

Figure 5. 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. Candidate generation is realized using the fusion process (joining phase) as in the GSP algorithm [11]:

Given a sequence s = (s1s2…sn) of size n and two candidate sequences c = (c1 c2 …cn-1) and c' = (c'1 c'2 … c'n-1) of size n-1, s is generated from c and c' if the following conditions hold:

$\forall\ i \in [2..n-1]$, $c_i = c'_{i-1}$. (n > 3). c and c' have a common contiguous subsequence of size n-2.

if n = 3, $c_2 = c'_1$. c and c' have only one item in common.

if n = 2, c = (c1) and c' = (c'1). Then s = (c1 c'1).

Then in case 1 and 2, s is generated as follow: $\forall\ i \in [1..n-1]$, $s_i = c_i$. And $\forall\ j \in [1..n-1]$, $s_{j+1} = c'_j$.

For example, consider the two candidates sequences c = (MMH) and c' = (MHM) of size 3. (MH) is a common contiguous subsequence of c and c', and of size 2. Thus the candidate s = (MMHM) is generated from c and c'.

3.4. Support counting

Once the candidates have been generated, their frequencies can be determined using the data structure. For a given candidate C of size S, the algorithm first looks in the cell number S of the Index where the first sequence of size S is encoded. Then, this line l is accessed. 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 (see figure 6).

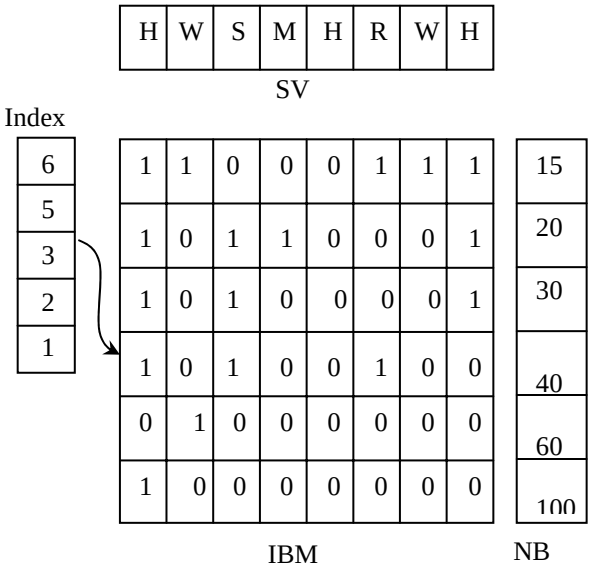

Figure 6. Example of candidate support counting

Suppose C = (HSH) of size S = 3. Then, the algorihtm will access to the cell number 3 of the Index which pin point to the line 3 of the IBM table, where the first sequence of size 3 starts. This sequence does not contain C. But sequences in line 4 to 6 contain C. So the frequency of C is computed as 30+20+15 = 65.

The support of C is equal to 65 / (100+60+40+30+20+15) = 0.245. If the support threshold is equal to 0.4, C candidate will not be retained as frequent pattern.

3.5. IBM2 proposal

The advantage of the data structure proposed in IBM is that it takes a few memory spaces. However, since the bit variable is not provided in programming languages like Java, C++, IBM algorithm need to decode the binary representation. Therefore, shifting operations are required to check each cell of the bit map for a given position in SV. This leads to decrease the performances of processing time.

In order to avoid these superfluous computations, we propose 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 SPAM and PrefixSpan.

4. Experimentations and performances analysis

This section first gives some results for our application, and then reports the performance analysis. An analytical evaluation is provided in section 4.2 followed by experimental tests. Finally, those results are discussed in section 4.4.

4.1. Analysis of population time survey

The IBM has been performed on real data related to daily activity programs of the population of Lille (a French town). In this application, the number of items is about 10, the number of sequences is 10800; while distinct sequences are about 3429; the sequence size varies between 2 and 34 with a mean of size equals to 6.
We have discovered some interesting patterns among which:

- 49% of the population do (Home, Leisure, Home).
- Only 37% of the population do (Home, Work, Home).
- 9% of the population do (Home, Work, Home, Work, Home).
- 11% of the population do (Home, School, Leisure, Home).
- 8% of the population do (Home, Shopping, Home, Leisure, Home).

These results allow better understanding the daily activity and mobility for a given population, which is useful in decision support. As an example, based on such results, policy makers may improve their transport policy

4.2. Performance evaluation

The cost analysis is based on the evaluation of the number of memory accesses, to determine the frequencies of the generated sub-sequences. Here, the cost is computed without taking into account the cost to generate candidates. Notations and parameters are listed in the table 1 below.

The cost to determine the frequency for one sub-sequence c is equal to:

Eq1: $Cost(c) = C_{index} + C_{NB} + C_{SV} + C_{IBM.}$

Where the cost to access a specified value in the Index is :

Eq.2: $C_{index} = 1$

This Index access will determine the starting position (pos) in the IBM for sequences to be compared with a candidate of size L_{SS} (Index[L_{SS}] = pos). The number of such lines is:

Eq.3: $C_{NB} = n - pos_c$.

4.2.1. Cost evaluation for IBM2

The cost to compare a sub-sequence of size L_{SS} with SV is equal to:

Eq.4: $$C_{SV} = L_{SS}(c) * C_{L_{SS}(c)}^{SV_E}$$

$$C_{SV} = L_{SS}(c) * \frac{SV_E!}{L_{SS}!(SV_E - L_{SS}(c))!}$$

Proof:

In order to compare c with SV, all sub-sequences of size $L_{SS}(c)$ in SV must be compared to c. The number of combinations of $M=L_{SS}(c)$ items among a list of $N=SV_E$ items is a common formula given by:

$$C_M^N = \frac{N!}{M!(N-M)!}$$

The number of comparisons for each item of a candidate c with a SV sub-sequence is equal to M. The cost to compare a candidate c with SV is equal to:

$$M * C_M^N = M * \frac{N!}{M!(N-M)!}$$

□

Example:

Suppose a sequence vector SV = (a, b, c d) of size $SV_E = 4$ and a candidate sub-sequence c of size $L_{SS}(c) = 3$. Then c can match with the following 4 sub-sequences of SV: (a,b,c), (a,b,d), (a,c,d), (b,c,d). The number of comparison to find c in SV is equal to 3*4=12 memory accesses is necessary to compare c with SV, which is equal to: $3 * \frac{4!}{3!(4-3)!}$

Finally to compare a candidate c in the bit map structure, $C_{IBM} = (n - pos_c) * C_{SV}$, since it is compared from position pos_c to the top.

Eq. 5: $$C_{IBM} = (n - pos_c) * L_{SS}(c) * C_{L_{SS}(c)}^{SV_E}$$

Based on the above equations, Eq. 1 to Eq.5, frequency calculation for one candidate c is:

$$Cost(c) = 1 + n - pos_c + L_{SS}(c) * C_{L_{SS}(c)}^{SV_E} + (n - pos_c) * L_{SS}(c) * C_{L_{SS}(c)}^{SV_E}.$$

$$Cost(c) = 1 + n - pos_c + (1 + n - pos_c) * (L_{SS}(c) * C_{L_{SS}(c)}^{SV_E}).$$

$$Cost(c) = (1 + n - pos_c) * \left\{1 + L_{SS}(c) * C_{L_{SS}(c)}^{SV_E}\right\}$$

Then for all of the N generated candidates (noted i for simplification), the algorithm cost C is equal to:

$$C = \sum_{i=1}^{N} Cost(i)$$

$$C = \sum_{i=1}^{N} (1 + n - pos_i) * \left\{1 + L_{SS}(i) * C_{L_{SS}(i)}^{SV_E}\right\}$$

4.2.2. Cost evaluation for IBM

Using IBM, shifting operations are required to get the value of a bit. For implementation, the Bit Map is in fact composed of a Byte Map, where each Byte encodes 8 bits. Then the maximum number of memory accesses to retrieve a specific bit value is equal to 8.

Then, the cost for IBM is equal to:

$$C = \sum_{i=1}^{N} (1 + n - pos_i) * \left\{1 + L_{SS}(i) * 8 * C_{L_{SS}(i)}^{SV_E}\right\}$$

Cost(k)	The cost of frequency calculation for k candidate sub-sequences.
C_{index}	The access cost to a specified value in the Index.
C_{NB}	The number of accessed lines in NB.
C_{SV}	The cost of comparison of a given sub-sequence with SV.
C_{IBM}	The cost of comparison of a given sub-sequence with sequences stored in IBM.
Tss	The number of items in a given sub-sequence.
SV_E	The number of elements in SV
n	The total number of lines in IBM
N	The total number of generated Candidates.
Pos_c	Position of the first sequence in the bit map of size equal to c size

Table 1. Notations and parameters

4.2.3. Remarks

According to the above formulas, the cost depends on:

1. The number m of lines to scan in the Bit Map ($m = n - pos_i$).
2. The length of SV (SV_E).
3. The length of a candidate sub-sequence ($L_{SS}(i)$).
4. The number of generated candidates (N).

When the threshold increases, the number of generated candidates N will decrease. Then the cost of retrieving all frequent patterns for great support will also decrease.

Inversely, when the threshold decreases, the number of generated candidates N increases. Thus the cost of retrieving all frequent patterns will increase. The number of lines to scan in the Bit or Boolean Map will be numerous. This has a direct repercussion in the processing time.

This is consistent with our experimental results illustrated from figures 7 to 10 in next section

4.3. Performance measurements

The experiments aimed to validate our approach and to compare it to other methods. This comparison focuses on processing performances, storage costs, and scalability. The tests were performed on a 2.5Ghz Pentium IV with 1 GB of memory running Microsoft Windows XP Professional, with three different sizes of datasets: 100,000; 300,000; 600,000; and 1,000,000 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 (from 0 to 9). For our experimentations, we have used the packages PrefixSpan-0.4.tar.gz[1] and Spam.1.3.1.tar.gz[2].

4.3.1. Processing time cost

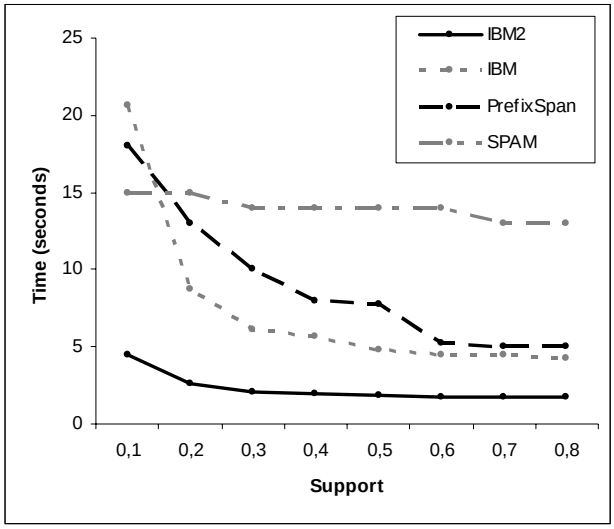

Figure. 7. Performances with 100,000 rows

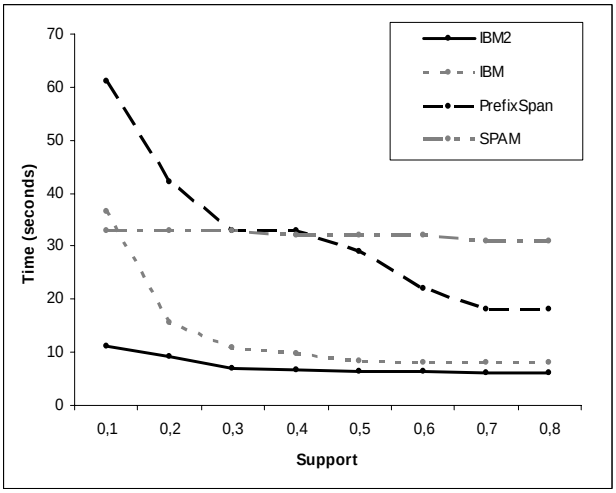

Figure. 8. Performances with 300,000 rows

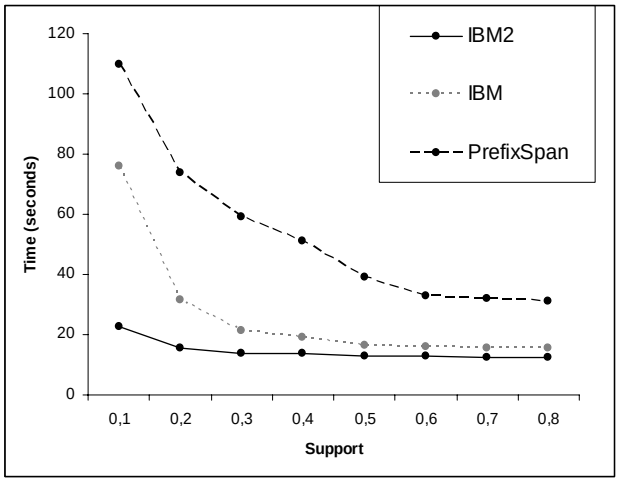

Figure. 9. Performances with 600,000 rows

[1] http://chasen.org/~taku/software/prefixspan/

[2] http://himalaya-tools.sourceforge.net/Spam/#download

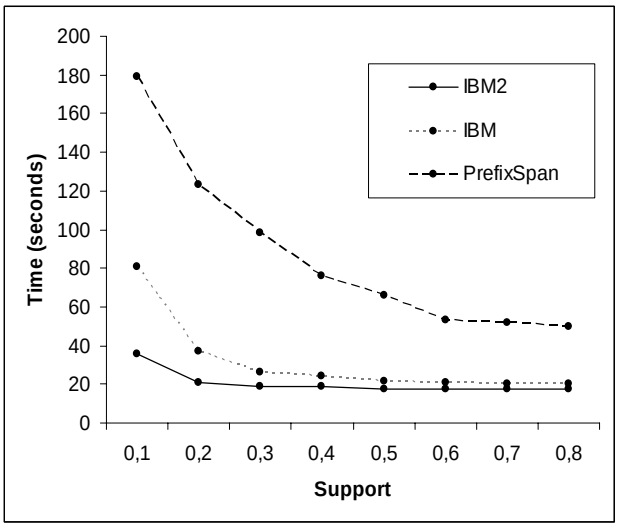

Figure. 10. Performances with 1,000,000 rows

Notice that IBM and IBM2 have been implemented in JAVA, and perform a database scan, whereas PrefixSpan and SPAM have been implemented in C++ and read the dataset from a file. Although SPAM and PrefixSpan development environment is theoretically more favorable, IBM and IBM2 outperform SPAM and PrefixSpan as shown in the figures above, especially in the context of small number of items. This arises in several applications as the one treated here, or for mining web traversal patterns (the number of distinct web pages is limited). Notice that beyond 700,000 rows with 512 MB of memory, PrefixSpan crashes because its data structure does not fit in the main memory, whereas IBM and IBM2 run efficiently and do not require much more memory resources than for smaller databases. We have pushed the experimentation to 1,000,000 sequences. These experiments show that IBM and IBM2 are more appropriate for large databases than SPAM and PrefixSpan.

The experimentations show that the larger is the database size, the more IBM and IBM2 outperform SPAM and PrefixSpan. This is because IBM and IBM2 make only one scan of the database, and have an efficient and compact structure allowing a fast retrieval of frequent sub-sequences. According to the analytical assessments (section 4.2), the number of memory accesses increase with low threshold. In consequence, the performances of IBM and IBM2 will decrease. This has been confirmed by the tests as seen in figures 7 to 10, where IBM and IBM2 curves goes up inversely with the threshold.

There is only one case where PrefixSpan outperforms IBM (see figure 7). This occurs for small size of dataset (here 100,000 rows) and small support threshold (here 0.2). This is because: (i) the number of candidates to compare increases when the support is low. This comparison in IBM requires many shifting operations, which affects the performances; (ii) scanning a small database in PrefixSpan will requires less I/O.

Compared to IBM, IBM2 and PrefixSpan, SPAM performs linearly and outperforms both PrefixSpan and IBM, when the support threshold becomes small: under 0.2 with 100,000 and 300,000 rows (figure 8). Beyond 300,000 rows, SPAM data structure requires too much memory space and may overflow depending on the used platform (see figure 11).

But whatever is the size of the database, IBM2 always outperforms SPAM, PrefixSpan, and IBM because no shifting operation is required.

4.3.2. Storage cost

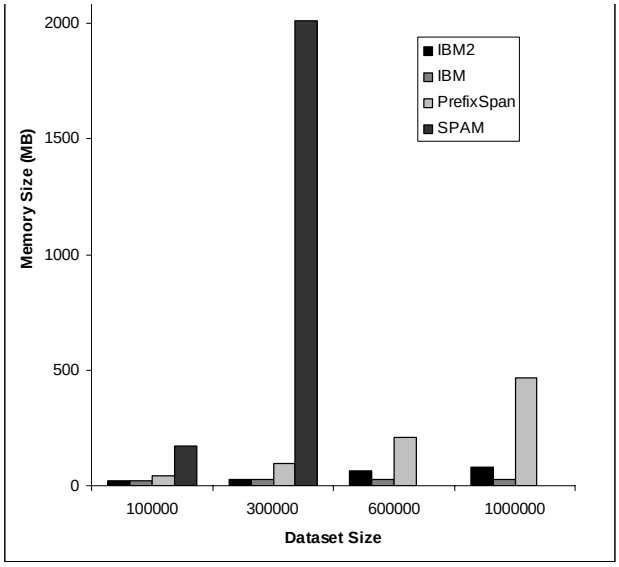

Figure. 11. Memory consumption

Figure 11 shows the total memory consumption (in Mega Bytes) used by IBM and IBM2, SPAM, and PrefixSpan.

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*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 1000,000 rows (figure 10), 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.

4.3. Discussion

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. As we can see in figure 11, memory consumption using IBM and IBM2 compared to SPAM and PrefixSpan becomes insignificant when the size of dataset becomes large. For example with 1,000,000 rows, the total memory consumption for IBM used by Java is equal to 28 MB (81 with IBM2) whereas for PrefixSpan, it is about 468 MB.

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. 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.

In order to prove the efficiency of IBM in extreme cases, we have performed a test with a sequence c_T of the following form: (H,W,H,W,H,W, … ,H,W) of size 200, composed of repeated series of H and W items. This type of sequence may increase the size of the data structure. This experiment aims first to demonstrate that this situation does not affect the processing costs. The second goal is to evaluate the loss of storage performance.

Tests have been done with datasets composed of 600,000 and 1,000,000 sequences. Indeed, no variation of processing costs has been detected. This is because, according to the sequence vector generation algorithm (section 3.2), the items (H or W) that are not located in other sequences are put at the end of SV. Therefore, unless repeated series are actually frequent in the database, the probability to have long repeated series of HW in the middle of SV is very low. Then, using the data structure for candidate generation and frequent patterns will not be affected, because the items H,W put at the end of the structure would never been accessed.

Concerning the storages costs, we have estimated and experimentally testes the loss of performances. For a dataset composed of 600,000 sequences, the size of SV varies from 265 to 328 values, with the new inserted test c_T sequence. Then the size of the Boolean Map is equal to: 328*90,000 = 29.52 Mega Bytes, and 3.69 MB for the Bit Map. This corresponds to about 19% more storage cost compared to the case without c_T.

For a dataset composed of 1,000,000 sequences, the size of SV varies from 370 to 415 values. The size of the Boolean Map is equal to: 415*160,000 = 66.40 Mega Bytes, and 8.30 Mega Bytes for the Bit Map. This represents 10% more storage cost.

The theoretical loss and the experimental memory measurements are given in figure 12 and 13.

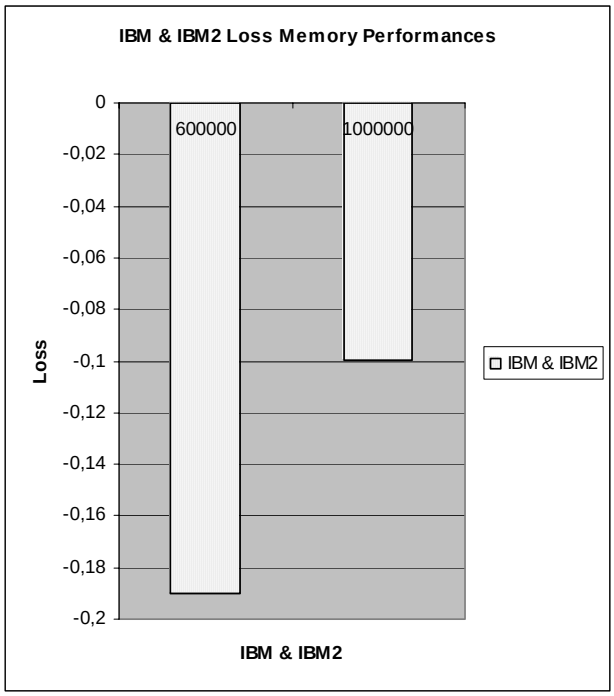

Figure. 12. Memory Loss Analytical Performances

In figure 12, the theoretical loss is estimated by:

$$loss = -\frac{New\_Map\_Size}{\Pr evious\_Map\_Size}$$

Previous_Map_Size stands for the Boolean or Bit Map without insertion of the test sequence, and New_Map_Size stands for the Boolean or Bit Map with insertion of the test sequence. Here we can notice that the size of the bit map is increased to roughly 19% with a dataset composed of 600,000 sequences, and 10% with 1,000,000 sequences. Here, the loss is less important for the dataset composed of 1,000,000 sequences than for 600,000 sequences (figure 12). This is because the more SV is long and the more the items composing a sequence have a higher probability to be found in SV. This also explains the gap performances between IBM and IBM2 with SPAM and PrefixSpan when the size of the dataset, and thus the size of SV, become greater.

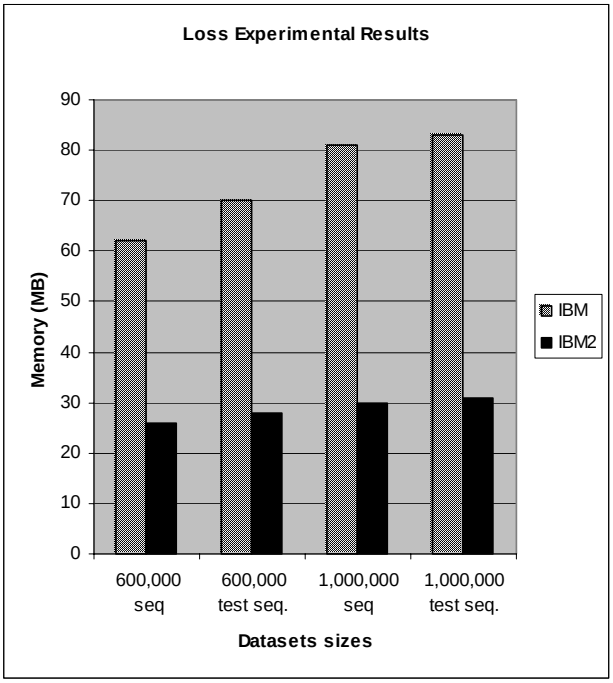

Figure. 13. Comparison of Storage Performances

Figure 13 gives the experimental measurements of memory consumption using datasets without c_T test sequence insertion (600,000 seq. and 1,000,000 seq.) and with c_T insertion (600,000 test seq. and 1,000,000 test seq.).

Here we can notice that the insertion of the c_T test sequence have a repercussion in the allocated size for IBM and IBM2, especially for a dataset composed of 600,000 sequences. But the variation of the memory allocation becomes insignificant with a higher SV size (with a dataset composed of 1,000,000 sequences).

The processing time for both datasets remains the same as in figure 9 and 10, since the items H,W located at the end of SV are never accessed.

5. Conclusion and perspectives

This paper presents a new algorithm IBM and a variation 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 database within an urban environment. IBM only makes one scan of the database and provides efficient data structure optimizing memory space occupancy, and access costs. The use of the specified index provides another optimization of comparisons during candidate counting.

The proposed algorithm allows complying with a new demand in the field of transport analysis. It is appropriate for large size databases with a low number of distinct items. The experiments have shown that IBM and IBM2 provide better performances than existing algorithms in most cases. Better response time is reached, while a very low memory is consumed.

Experimental results show that IBM2 outperforms IBM, which in turns outperforms SPAM and PrefixSpan for large and very large databases. But depending on the size of the dataset, IBM would be a better choice than IBM2 for very large database.

Notice that the proposed data structure for IBM and IBM2 algorithms and especially the SV vector could be used for other purposes as similarity search between sequences and sequence clustering. Therefore, we plan to use this data structure for efficient sequence clustering and similarity analysis. Another perspective is to apply it to different application contexts, as the analysis of traversal patterns in web usage mining [10], or DNA mining, and finally to extend the algorithm to customer sequence databases.

In the context of the activity-mobility survey, we will explore the mining of spatial sequences (such as trajectories) and the extension to multidimensional sequential patterns as in [13]. Indeed, each daily activity program is described using many attributes related to the individual like age, socio-professional category, etc. In addition, the sequence items may also have attributes as transport mode, duration, geographical location etc. This is still a challenging research issue.

6. 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] Ahonen H., “Finding all maximal frequent sequences in text”. ICML 1999, Machine Learning in text Data Analysis Workshop. Bled, Slovenia (1999).

[4] Jay A., Johannes .G, Tomi .Y, Jason F., “Sequential Pattern Mining Using a Bitmap Representation”. SIGMOD pp 429-435, July (2002), Edmonton, Alberta, Canada.

[5] Gardarin G., Pucheral P., Wu F., “Bitmap Based Algorithms for Mining Association Rules”, 14èmes Journées Bases de Données Avancées, BDA 1998, pp157-175, Hammamet, Tunisie, Octobre 1998.

[6] Han, J., Jamil, H. M., Lu, Y., Chen, L., Liao, Y. and Pei, J. DNA Miner, “A system prototype for mining DNA sequences”. In the proc. of the ACM SIGMOD, 2001, Santa Barbara, CA, USA.

[7] Masseglia F., Poncelet P., Teisseire M., “Incremental mining of sequential patterns in large databases”. Data Knowledge Engineering 46(1), pp 97-121 (2003).
[8] Ministère de l’Equipement, des Transports et du Logement. L’enquête ménages déplacements “méthode standard”. Collections du Certu. Octobre (1998). ISSN 1263-3313.
[9] Pei J., Han J., B. Mortazavi-Asl, and Pinto H., “Prefixspan: Mining sequential patterns efficiency by prefix-projected pat tern growth.” In Proc. of the International Conference on Data Engineering (ICDE), pp 215–224, 2001.
[10] Pei J., Han J., B. Mortazavi-Asl and H.Zhu, “Mining Access Patterns Efficiently from Web Logs”, Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining, pp. 396-407, 2000.
[11] Srikant R., Agrawal R., “Mining Sequential Patterns : Generalizations and Performance Improvements”. Proc. 5th EDBT, Mars 25-29, (1996). Avignon, France. pp 3-17.
[12] 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.
[13] Yu C.-C. and Y.-L. Chen. “Mining sequential patterns from multidimensional sequence data”. IEEE Transactions on Knowledge and Data Engineering, 17(1) pp 136-140, 2005.
[14] Zaki M. J., “Efficient Enumeration of Frequent Sequences”. Int. Conference on Information and Knowledge Management, November 1998, Washington DC.