FAST Sequence Mining Based on Sparse Id-Lists

Eliana Salvemini[1], Fabio Fumarola[1], Donato Malerba[1], and Jiawei Han[2]

[1] Computer Science Dept., Univ. of Bari, E. Orabona, 4 - 70125 Bari, Italy
{esalvemini,ffumarola,malerba}@di.uniba.it
[2] Computer Science Dept., Univ. of Illinois at Urbana-Champaign, 201 N Goodwin Avenue Urbana, IL 61801, USA
hanj@cs.uiuc.edu

Abstract. Sequential pattern mining is an important data mining task with applications in basket analysis, world wide web, medicine and telecommunication. This task is challenging because sequence databases are usually large with many and long sequences and the number of possible sequential patterns to mine can be exponential. We proposed a new sequential pattern mining algorithm called FAST which employs a representation of the dataset with indexed sparse id-lists to fast counting the support of sequential patterns. We also use a lexicographic tree to improve the efficiency of candidates generation. FAST mines the complete set of patterns by greatly reducing the effort for support counting and candidate sequences generation. Experimental results on artificial and real data show that our method outperforms existing methods in literature up to an order of magnitude or two for large datasets.

Keywords: Data Mining, Sequential Pattern Discovery, Sparse Id-List.

1 Introduction

Finding sequential patterns in large transactional databases is an important data mining task which has applications in many different areas. Many sequential pattern mining algorithms have been proposed in literature [1,2,4,5,9,3,6]. The main limits of current algorithms are: 1) the need of multiple scans of the database; 2) the generation of a potentially huge set of candidate sequences; 3) the inefficiency in handling very long sequential patterns.

The most demanding phases of a sequence mining process are candidate generation and support counting. Candidate generation is usually based on the *Apriori property* [1] according to which super-patterns of an infrequent pattern cannot be frequent. This property helps to prune candidates and to reduce the search space. Basing on this heuristic GSP [2] and SPADE [4] methods have been proposed. Since the set of candidate sequences includes all the possible permutations of the frequent elements and repetition of items in a sequence, A-priori based methods still generate a really large set of candidate sequences. The support counting phase requires multiple database scans each time the candidate sequence length grows, which is prohibitively expensive for very long sequences.

M. Kryszkiewicz et al. (Eds.): ISMIS 2011, LNAI 6804, pp. 316–325, 2011.

To deal with these issues, Pei *et al.* developed [3] a new approach for efficient mining of sequential pattern in large sequence database. *Prefixspan* adopts a depth-first search and a pattern growth principle. The general idea is to consider only the sequences prefix and project their corresponding postfix into projected databases. This way the search space is reduced in each step because projected databases are smaller than the original database. Sequential patterns are grown by exploring only the items local to each projected database. The drawback of Prefixspan is that for long sequences it needs to do many database projections.

Lately in 2002 Jay Ayres *et al.* present a faster algorithm, SPAM [5] which outperforms the previous works up to an order of magnitude for large databases. SPAM combines an efficient support counting mechanism and a new search strategy to generate candidates. The support counting is based on a data vertical representation in bitmaps and it does not require database scans. The drawback of SPAM is that it uses too much memory to store the bitmaps.

Another algorithm based on bitmap representation is HVSM [6]. HVSM is slightly faster than SPAM because it uses a different search strategy. At the beginning it generates all the frequent items or itemsets which constitute the first tree level. On these nodes it performs only sequence-extensions to generate sequences, by taking brother nodes as child nodes. This way HSVM is faster than SPAM because it discovers all the frequent itemsets at the first step.

To mine large databases of long sequences, we propose a new method FAST (Fast sequence mining Algorithm based on Sparse id-lisTs), which prevents multiple database scans to compute pattern support by indexing the original set of sequences with specific data structures called *sparse id-lists*. The database is read once and loaded in the main memory in an indexed form, such that each item is associated with the lists of all the sequence ids in which it appears in the database, preserving the transaction ordering. The support is directly computed from the data structures associated to each sequence without requiring database scans. Our proposed method uses the same lexicographical tree of HSVM so it gains advantage from the idea to mine all the frequent itemsets as first step.

2 Problem Definition

The problem of mining sequential patterns can be stated as follow. Let $I = \{i_1, i_2, ..., i_n\}$ be a set of different **items**. An **itemset** is a non-empty unordered subset of items (or a single item) denoted as $(i_1, i_2, ..., i_k)$ where i_j is an item for $1 < j < k$. A **sequence** is an ordered list of itemsets denoted as $\langle s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_m \rangle$ where s_i is an itemset i.e. $s_i \subset I$ for $1 < j < m$. An item can occur at most once in an sequence itemset, but multiple times in different itemsets of a sequence. The number of items in a sequence is called the **length** of the sequence. A sequence of length k is called a **k-sequence**. A sequence $\alpha = \langle a_1 \rightarrow a_2 \rightarrow ... \rightarrow a_n \rangle$ is said a **subsequence** of another sequence $\beta = \langle b_1 \rightarrow b_2 \rightarrow ... \rightarrow b_m \rangle$ and β a **super sequence** of α, denoted as $\alpha \ll \beta$, if there exist integers $1 \leq j_1 \leq j_2 \leq ... \leq j_n \leq m$ such that $a_1 \subset b_{j1}$, $a_2 \subset b_{j2}$ and $a_n \subset b_{jn}$, *i.e.*, the sequence $A \rightarrow BC$ is a subsequence of the sequence $AB \rightarrow E \rightarrow BCD$,

while $AC \rightarrow AD$ is not a subsequence of $A \rightarrow C \rightarrow AD$. A **sequence database** D is a set of tuples $\langle sid, S\rangle$, where sid is a sequence id and S is a sequence. A **transaction** is formally an itemset. Each sequence has associated a list of transactions, each with a time stamp. We assume that no sequence has more than one transaction with the same time stamp, so that the transaction time can be the transaction id and we can also sort the transactions in the sequence according to the transaction time. A tuple $\langle sid, S\rangle$ is said to **contain** a sequence α if α is a subsequence of S, i.e. $\alpha \subset S$. The **support** or **frequency** of a sequence α in a database D, denoted as $\sigma(\alpha)$, is the number of tuples which contain α, i.e. $\sigma(\alpha) = \{\langle\ sid,\ S\ \rangle \mid (\langle\ sid,\ S\ \rangle\ \epsilon\ D) \wedge (\alpha \subset S)\}$. Given a user defined threshold δ called **minimum support**, denoted **min_sup**, a sequence α is said **frequent** in a database D if at least δ sequences in D contain α, i.e. $\sigma(\alpha) \geq \delta$. A frequent sequence is also called **sequential pattern**. A sequential pattern of length k is called an **k-pattern**. Given a sequence database D and a *min_sup* threshold δ, the problem of sequential pattern mining consist in finding the complete set of sequential patterns in the database D.

Example 1: Let D in Table 1 be a sequence database. Fixed $\delta = 2$. We omit the arrows in the sequences for brevity. The sequence $a(abc)(ac)d(cf)$ has 5 itemsets (transactions): a,(abc),(ac),d,(cf). Its length is 9, item a appears three times in the sequence so it contribute 3 to its length but the whole sequence contributes only 1 to the support of a. Sequence $\alpha = a(bc)$ is a subsequence of the sequences $a(abc)(ac)d(cf)$ and $(ad)c(bc)(ae)$, so $\sigma(\alpha) = 2$ and α is a sequential pattern.

Table 1. An example of sequence database D

Sequence_ID	Sequence
10	a (abc) (ac) d (cf)
20	(ad) c (bc) (ae)
30	(ef) (ab) (df) c b
40	e g (af) c b c

3 Algorithm

In this section we describe the algorithm, the lexicographic tree and the new data structure called *sparse id-list* upon which our algorithm is based.

Lexicographic Tree: Suppose we have a lexicographic ordering $\leq$ on the items I in the database. This ordering can be extended to sequences, by defining $a \leq b$ if sequence a is a subsequence of b. All sequences can be arranged in a lexicographic tree in which a node labeled with a is left brother of a node labeled with b if item a appears before item b in the lexicographic ordering. A node a is father of a node b if sequence in node a is a subsequence of sequence in node b.

A sequence can be extended with either a *sequence/itemset-extension*. A sequence-extended sequence is a sequence generated by adding a new transaction of a single item to the end of the sequence. An itemset-extended sequence is generated by adding an item to the last itemset of the sequence.

First Step: Itemset Extension. With a first database scan we compute the set I of all the frequent 1-items. Then we build a lexicographic *itemset tree* to compute all the frequent itemsets starting from I. The itemset tree has as root's children all the frequent 1-items and on these nodes we perform only *itemset-extensions*. Each node is iteratively expanded by taking its right brother nodes as child nodes. If the itemset obtained concatenating the labels of the two nodes is frequent, the candidate child node is added in the tree, otherwise it is pruned and that path no more expanded. After the *itemset-extension* all and only the frequent items and itemsets are contained in the lexicographic itemset tree. These are used as input to generate another lexicographic tree, the *sequence tree*, to mine all the frequent sequences.

Second Step: Sequence Extension. The first level of the *sequence tree* contains all the frequent items and itemsets generated in the first step. For sequences discovery, each node is iteratively expanded by taking all its brother nodes as child nodes, included itself. If the obtained sequence is frequent, then the new candidate child node is added in the tree, otherwise it is pruned and that path is not expanded. At the end of this process, the lexicographic *sequence tree* will contain all the sequential patterns.

4 Data Structure: Sparse Id-List

The support counting method of FAST is based on a novel data structure that we called *sparse id-list* (SIL). The concept of id-list is not new in literature, it was first introduced by SPADE [4]. SPADE id-list is a list of all the customer-id and transaction-id pair containing the element in the database. Its size is variable. Itemset/sequence-extensions are executed by joining the id-lists. This operation for long id-list is very expensive in time. Differently from SPADE, our *SIL* gives a horizontal representation of the database.

Definition 1. *Let D be a sequence database, $|D|$ the size of the D, $j \in (0,\ldots,|D|)$ be a sequence, as defined in section 2. For each frequent item/itemset i a* sparse id-lists *SIL_i can be defined as a vector of lists, where:*

- *the size of the vector of SIL_i is $|D|$,*
- *$SIL_i[j]$ is a list and contains the occurrences of i in the sequence j,*
- *if sequence j does not contains i, its list in position $SIL_i[j]$ has value null.*

The vector of SIL_i has as many rows as are the database sequences, its size is fixed and depends on $|D|$. Each position of the vector corresponds to a database sequence, and is a list which contains all the transaction ids of the sequence. We associate a *sparse id-list* to each item i in the sense that in the list of the *j-th* sequence there will be only the ids of transactions which contain item i in

sequence j, therefore the list size is not fixed. In the case an item appears in all the transactions of sequence j, the *j-th* list size will be equal to the *j-th* sequence length in the database, but in general it is shorter, that's why we named this data structure *sparse* id-lists. If an item does not appears in any transaction of a sequence, the list corresponding to that sequence in the vector will be *null*.

4.1 Support Counting for Item and Itemset Based on Sparse Id-List

A *SIL* is associated to each frequent item and itemset in the database and stored in memory. *SILs* for 1-items are incrementally built during the first database scan. *SILs* for itemsets are built during the itemset-estension from *SILs* of single items verifying the condition that the single items of itemset appear in the same transaction of the same sequence. The support is efficiently computed by just counting the number of lists in the *SIL* which are not *null*. This is done during the construction of the *SILs* itself without requiring a scan of the whole *SILs*.

Example 2: Consider the sequence database in Table 1 with $\sigma = 2$. With the first database scan FAST builds the *SILs* (Fig. 1(a) ... 2(a)) for all the 1-items, computes the supports and prunes all the *SILs* of unfrequent items. Item g is infrequent so its *SIL* has been removed. Now we perform an itemset-extension and build the *SIL* for itemset (ab) (Fig. 2(b)). To do this we have to scan all the rows in the *SILs* of a (Fig. 1(a)) and b (Fig. 1(b)) and for each row j we have to scan the list in it and report in the new *SIL* only the transaction ids which are equal between a and b. In this case at the first row the first transaction-id which is equal between a and b is 2, in the second row there isn't so we store *null*, at the third row it is 2, while in the fourth there isn't. *SIL* for itemset (ab) is now stored in memory along with *SILs* of frequent 1-items and it will be used in itemset-extension to compute the support for longer itemset ((abc) i.e.).

4.2 Support Counting for Sequence Based on Sparse Id-List

The support for sequences is computed from a derivative of *SIL* that we called *vertical-id-list* (*VIL*). It is a vector having the same size of the *SIL*. Each position of the vector refers to a particular sequence in the database, as for the *SIL*,

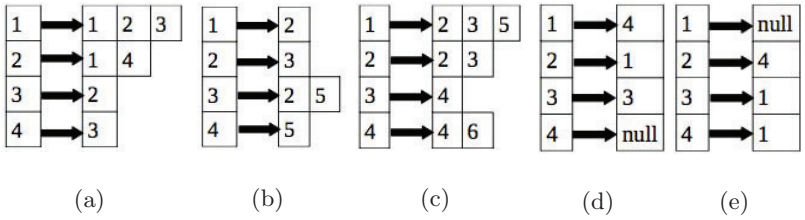


Fig. 1. From left to right, the *sparse id-lists* for items: a support $\sigma = 4$; b support $\sigma = 4$; c support $\sigma = 4$; d support $\sigma = 3$; e support $\sigma = 3$; f support $\sigma = 3$

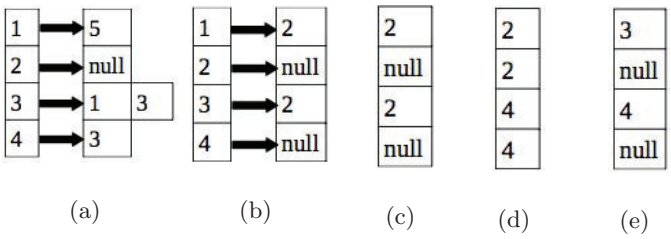


Fig. 2. From left to right: *sparse id-list* for itemset (ab) with support $\sigma(ab) = 2$; *vertical id-list* for itemset (ab), item c and sequence $(ab) \rightarrow c$

and stores only the first transaction id in which the element we are trying to concatenate in the sequence occurs, plus a reference to the *SIL* in which that transaction id is stored. During the sequence extension FAST stores in each node of the first level of the sequence tree a *VIL* for all the frequent items and itemsets computed in the first step. To generate the sequence $a \rightarrow b$ a new *VIL* is generated, verifying the condition that item b appears in the same sequence but in a transaction after items a. If this condition is not true we use the reference at the *SIL* stored in the *VIL* of b to move to the next transaction id greater than the transaction id of a. This transaction id will be stored in the new *VIL* of sequence $a \rightarrow b$ or *null* if it does not exist. The support of the sequence corresponds to the number of elements different than null in the *VIL* and, as for the *SIL*, it is computed during the construction of the *VIL* without requiring its scan. If the sequence $a \rightarrow b$ is frequent its *VIL* will be stored in the node $a \rightarrow b$ in the *sequence tree* and it will be used in the next sequence extension to compute the support of the super sequences of $a \rightarrow b$. The reader can notice that we only compute one scan of the database at the beginning of the algorithm to generate all the frequent 1-items and their corresponding *SILs*. Then the support for itemsets and sequences is directly computing by just counting the number of rows different than null in the *SIL* or *VIL*.

Example 3: Let us consider *SIL* for itemset (ab) in Fig. 2(b) and for item c in Fig. 1(c). Suppose we want to compute sequence $(ab) \rightarrow c$. We first have to get the *VILs* for elements (ab) and c from their *SILs* and stored in the corresponding nodes in the sequence tree. The *VILs* are showed in Figs. 2(c) and 2(d). For each position of the *VIL* of c, we search for a transaction id which is greater than the transaction id of the *VIL* of (ab) in the same position. If this condition is false we move to the next element using the reference at the *SIL* stored in the *VIL*. In this case the first transaction id in the *VIL* of c is 2 which is not greater than transaction id in the *VIL* of (ab), so we move to the next transaction id in the *SIL* of c (Fig. 1(c)) which is 3. The second transaction id is again 2 but since the transaction id in (ab) is null we have to skip to the following position. The next transaction id is 4 which is greater than the transaction id in (ab) 2 so it is stored in the new *VIL*. Finally the last transaction-id is 4 but since the

transaction id in (ab) is null the condition is not verified. The *VIL* for sequence $(ab) \rightarrow c$ is showed in Fig. 2(e).

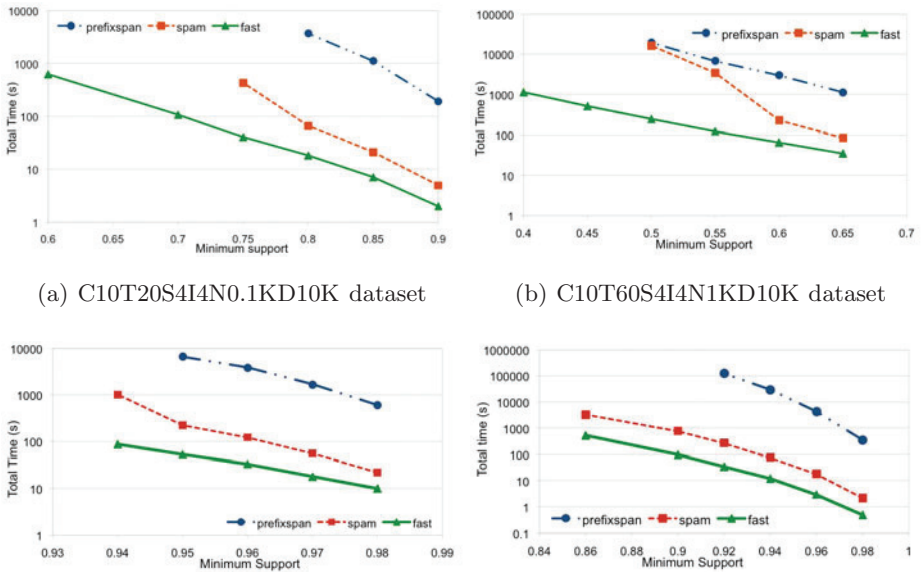
5 Experiments

We report the performances of FAST against PrefixSpan [3] and SPAM [5] by varying different database parameters. We did not compare FAST against SPADE [4] because authors in [5] demonstrate that SPAM always outperforms SPADE and against HSVM because the differences of performance between SPAM and HSVM are minimal as shown in [6]. We implemented FAST in java. The java codes/executables for PrefixSpan and SPAM were obtained from Philippe Fournier-Viger's [11] and Joshua Ho's websites [8]. The experiments were performed on a 2.4GHz Intel Xeon server with 10 GBs of RAM, running Linux.

Datasets: The synthetic datasets we used were generated using the IBM data generator [1], that mimics real-world transactions where people buy sets of items. We compared the performances by varying several factors which can be specified as input parameters to the data generator, with different minimum support values on small and large databases. These parameters are: D the number of customers in the dataset; C the average number of transactions per customer; T the average number of items (transaction size) per transaction; S the average size of maximal potentially large sequences; I the average size of itemset in maximal potentially large sequences and N the number of different items. We also compared the algorithms on a real dataset Pumsb-star, taken from the FIMI repository [10] which is dense and has 49046 long sequences of single items.

5.1 Performance Comparison

Figures 3(a)- 3(b) show performance comparison on small datasets. The datasets are identical, except that 3(a) has 100 distinct items (N=0.1k), whereas 3(b) has 1000 (N=1k). Figures 3(c), 3(d) and 4(a) show performance comparison on large datasets, with 100000 sequences (D=100k) with longer sequences (C=20, 35, 40). Experimental results shows that FAST has the best performance. It outperforms SPAM of one order of magnitude in the small and large datasets, and PrefixSpan till to two orders of magnitude on large datasets. For the support values where SPAM can run, it is generally in the second spot. It fails to run for lower support values because it needs huge memory to store the bitmaps. In all the experiments PrefixSpan turns out to be very slow compared to the other two algorithms. For long sequences it fails because it needs to do too many database projections. FAST instead can run faster also in presence of longer sequences with large datasets. Finally, Fig. 4(b) shows the performance comparison on Pumsb-star real dataset and on this dataset FAST is an order of magnitude faster than SPAM and two orders faster than PrefixSpan.



(a) C10T20S4I4N0.1KD10K dataset

(b) C10T60S4I4N1KD10K dataset

(c) C20T20S20I10N0.1KD100K dataset

(d) C40T40S20I20N1KD1K dataset

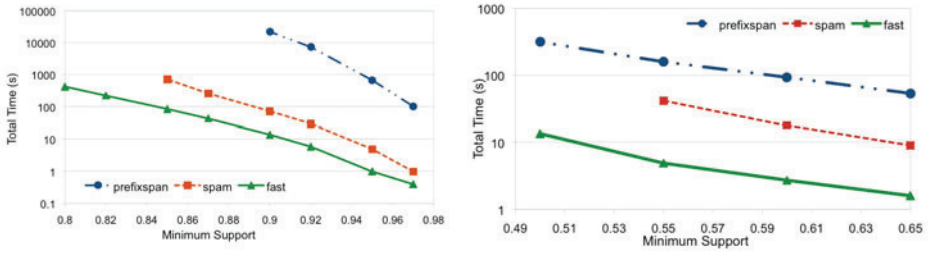
Fig. 3. Performance comparison varying minimum support on different datasets

To show that experimental results are not dependent on the language implementation, we compare our java version of FAST with the original C++ implementation of PrefixSpan [12]. Results in Fig. 5 show that FAST is still faster than PrefixSpan and starting from support 0.92 it is an order of magnitude faster than PrefixSpan.

5.2 Memory Usage

In Fig. 6 we presented graphics for the average memory consumption of the three algorithms on other synthetic datasets. We can quickly see as SPAM requires really much memory compared to FAST and PrefixSpan, up to one order of magnitude. SPAM is inefficient in space because even when an item is not present in a transaction it needs to store a zero in the bitmap to represent this fact. The trend of SPAM and FAST is comparable even if PrefixSpan requires less memory than FAST, because at each run it reduces the database size of one length avoiding to store sequence prefix in the database. FAST requires more memory than PrefixSpan because it needs to store in memory the *sparse id-lists* for all the frequent 1-items and itemsets. However, the memory usage of FAST is comparable to the one of PrefixSpan.

In Fig. 6(b) we present another kind of graphic which shows the total memory consumption of the three algorithms during the whole mining process, on the



(a) C35T40S20I20N1KD1K (b) pumsb-star

Fig. 4. Performance comparison on C35T40S20I20N1KD1K (a) and Pumsb-star (b)

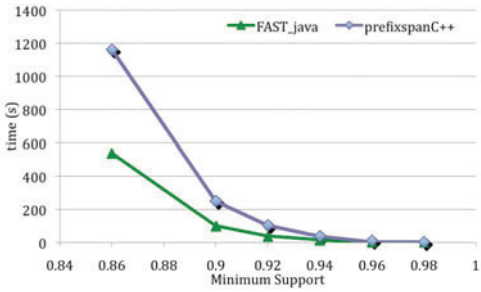
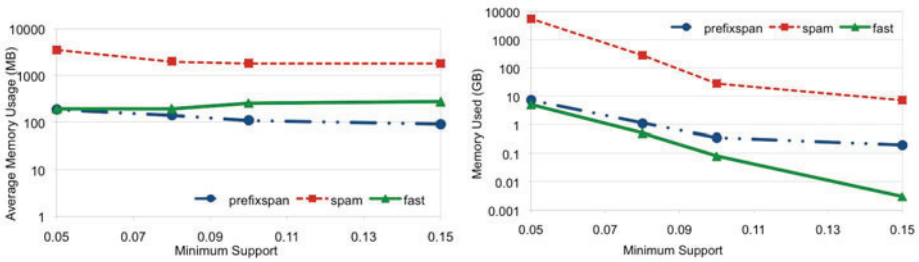


Fig. 5. Fast in Java vs. PrefixSpan in C++ on dataset C40T40S20I20N1KD1K



(a) D=5 C=15 T=10 S=10 I=10 (b) D=5 C=15 T=10 S=10 I=10

Fig. 6. Memory comparison on medium datasets

same datasets. It is interesting to notice that the total memory used by FAST is lower than the one used by SPAM and Prefixspan. This happens because FAST is faster than the other two algorithms in all the experiments that we performed and in this way it occupies less memory during the whole computation. We believe that even if the average memory consumption of FAST is slightly upper than PrefixSpan, the memory consumption of FAST is not a big issue as for

example in SPAM because the total memory that FAST uses is comparable and in some cases less than PrefixSpan.

6 Conclusions

We presented a new sequence mining algorithm FAST that quickly mines the complete set of patterns in a sequence database, greatly reducing the effort for support counting and candidate sequences generation phases. It employs a new data representation of the dataset based on sparse id-lists and indexed vertical id-lists, which allows to quickly access an element and count its support without database scans. Future work will consist of mining all the closed and maximal frequent sequences, as well as pushing constraints within the mining process to make the method suitable for domain specific sequence mining task.

References

1. Agrawal, R., Srikant, R.: Mining sequential patterns. In: International Conference on Data Engineering (ICDE 1995), Taipei,Taiwan, vol. 0, pp. 3–14 (1995)
2. Srikant, R., Agrawal, R.: Mining sequential patterns: Generalizations and performance improvements. In: Apers, P., Bouzeghoub, M., Gardarin, G. (eds.) Advances in Database Technology EDBT 1996, vol. 1057, pp. 1–17. Springer, Heidelberg (2006)
3. Pei, J., Han, J., Asl, M.B., Pinto, H., Chen, Q., Dayal, U., Hsu, M.C.: PrefixSpan Mining Sequential Patterns Efficiently by Prefix Projected Pattern Growth. In: Proc.17th Int'l Conf. on Data Eng., ICDE 2001, Heidelberg, Germany, pp. 215–226 (2001)
4. Zaki, M.J.: SPADE: An efficient algorithm for mining frequent sequences. Machine Learning 42, 31–60 (2001)
5. Ayres, J., Gehrke, J., Yiu, T., Flannick, J.: Sequential PAttern Mining using A Bitmap Representation, pp. 429–435. ACM Press, New York (2002)
6. Song, S., Hu, H., Jin, S.: HVSM: A New Sequential Pattern Mining Algorithm Using Bitmap Representation. In: Li, X., Wang, S., Dong, Z.Y. (eds.) Advanced Data Mining and Applications, vol. 3584, pp. 455–463. Springer, Heidelberg (2005)
7. Fournier-Viger, P., Nkambou, R., Nguifo, E.: A Knowledge Discovery Framework for Learning Task Models from User Interactions in Intelligent Tutoring Systems. In: Gelbukh, A., Morales, E. (eds.) MICAI 2008. LNCS (LNAI), vol. 5317, pp. 765–778. Springer, Heidelberg (2008)
8. Ho, J., Lukov, L., Chawla, S.: Sequential Pattern Mining with Constraints on Large Protein Databases (2008), `http://sydney.edu.au/engineering/it/~joshua/pexspam/`
9. Han, J., Pei, J., Asl, B.M., Chen, Q., Dayal, U., Hsu, M.C.: FreeSpan: frequent pattern-projected sequential pattern mining. In: KDD 2000: Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data mining, Boston, MA, pp. 355–359 (2000)
10. Workshop on frequent itemset mining implementations FIMI 2003 in conjunction with ICDM 2003, `http://fimi.cs.helsinki.fi`
11. SPMF, `http://www.philippe-fournier-viger.com/spmf/index.php`
12. Illimine, `http://illimine.cs.uiuc.edu/`