An Efficient Method for Mining Clickstream Patterns

Bang V. Bui[1], Bay Vo[2], Huy M. Huynh[3], Tu-Anh Nguyen-Hoang[1], and Bao Huynh[4(✉)]

[1] University of Information Technology, Vietnam National University HCMC, Ho Chi Minh City, Vietnam
vanbang0208@gmail.com, anhnht@uit.edu.vn
[2] Faculty of Information Technology, Ho Chi Minh City University of Technology (HUTECH), Ho Chi Minh City, Vietnam
vd.bay@hutech.edu.vn
[3] Faculty of Electrical Engineering and Computer Science, Technical University of Ostrava (VŠB), Ostrava-Poruba, Czech Republic
huy.minh.huynh.st@vsb.cz
[4] Faculty of Information Technology, Ton Duc Thang University, Ho Chi Minh City, Vietnam
huynhquocbao@tdt.edu.vn

Abstract. Recently, hybrid approaches, which combine an FP-tree-like data structure with an interaction-based approach, are efficient approaches for mining frequent itemsets. However, applying those approaches for sequential pattern mining arose some challenges. In this paper, we introduce a hybrid approach for a specific version of sequential pattern mining, clickstream pattern mining, with our proposed B-List structure and SMUB algorithm. The SMUB algorithm exploited the B-List structure that is generated from the SPPC tree and the B-List intersection are used to discover all sequential patterns in the given sequence database. Via our experiments on various databases, SMUB has been shown to be more efficient than the current state-of-the-art algorithm, CM-Spade, in terms of runtime, and scalability, especially on huge databases with very small thresholds.

Keywords: Data mining · Clickstream pattern · Sequence pattern

1 Introduction

The problem of sequential pattern mining was first brought up by Srikant and Agrawal in 1995 [2]. Since then, there have been quite a lot of approaches and algorithms proposed to solve this problem. However, finding an effective method is still challenging. Recently, hybrid approaches using DiffNodeSets [10], N-List [9] data structures are reported as very efficient for mining frequent itemsets. But can those approaches be applied for mining pattern with a sequential order? To the best of our knowledge, there have not any work that was based on the hybrid approaches using

H. S. Nguyen et al. (Eds.): IJCRS 2018, LNAI 11103, pp. 572–583, 2018.
https://doi.org/10.1007/978-3-319-99368-3_45

those data structures. Itemset patterns are easier to deal with because each item only appears once at most in each transaction of the database, and the order of items in the itemsets can be assigned by users. On the other hand, sequential patterns consist of multiple transactions in sequential or timely order. Thus, each item can appear more than one in a sequence, in various transactions, and in an order that users cannot predict.

In this paper, we propose the SMUB algorithm to tackle a part of sequential pattern mining problem by solving clickstream pattern mining, a special version of sequential pattern mining. SMUB is a hybrid-based approach algorithm, based on B-List, an extension of N-List data structure. B-Lists are generated from an SPPC tree. Via our experiments on various datasets have shown that SMUB was more efficient than the recent state-of-the-art algorithm, CM-Spade [11], with respect to runtime, especially on huge datasets with low minimum support thresholds.

We organized this paper as follows. In Sect. 2, we describe the basic concepts. In Sect. 3, we introduce related work. In Sect. 4, we introduce SPPC tree and definitions. In Sect. 5, we present our B-List and SMUB algorithm for clickstream pattern mining. In Sect. 6, we present our experiments. In Sect. 7, we conclude our study and present our future work.

2 Basic Concepts

Let $I = \{i_1, i_2, \ldots, i_j\}$ be a set of distinct elements, each element is called an item. A sequence is a list of items that are ordered. A clickstream sequence S is denoted as $\langle s_1, s_2, \ldots, s_q \rangle$, where $s_p \in S(1 \leq p \leq q)$ is an item. The number of items in clickstream is called the size or length of the clickstream. A clickstream sequence having length k is denoted as k-sequence. A clickstream sequence $S_a = \langle a_1, a_2, \ldots, a_n \rangle$ is a subsequence of another clickstream sequence $S_b = \langle b_1, b_2, \ldots, b_m \rangle$, denoted by $S_a \subseteq S_b$, if there exist integers $x_1 < x_2 < \cdots < x_y$ that $a_t = b_{x_y}$ with all of a_t. In other words, S_b is called a super sequence of S_a.

A clickstream sequence database *SDB* is a collection of clickstream and each sequence has a unique id (called sid). Support of a clickstream pattern P is defined as the number of clickstreams in *SDB* that are the super sequences of P. Given a threshold, a clickstream sequence is a frequent clickstream pattern if its support is more than or equal to the given threshold. The clickstream pattern mining task is discovering all frequent clickstream patterns in *SDB*.

Table 1. A clickstream sequence database

SID	Clickstream
100	<2,5,1>
200	<2,5,1,5,1>
300	<2,3>
400	<1,5,1,5,1>

3 Related Work

Several algorithms have been proposed for sequential pattern mining such as AproriAll [2], GSP [3] and SPADE [5]. All of them find all sequential patterns by using "generate and test candidate" approach which consumes a lot of time and memory. PrefixSpan [6], FUSP [7] and Sequential Pattern Tree [8] does not generate any candidate sequences, but the structure of the tree is complex; thus, they create lots of projected databases and in order to find new sequential patterns, they need to completely scan the projected databases.

SPADE algorithm [5] identifies all frequent items (viz., 1-sequences) at the beginning, converts the database to the vertical database format and identify the rest of sequential pattern by BFS or DFS based on lattice decomposition concept. Though experiments, it is more efficient than the GSP algorithm. However, SPADE needs to convert database from horizontal to the vertical format, so the memory usage for storing the databases increased and it is even bigger than the original databases.

In 2008, Lin et al. proposed FUSP-tree [7] data structure and its maintenance algorithm for mining sequential patterns in incremental databases. FUSP-tree consists of one root node and a set of prefix subtrees as the children branches of the root. Each node in the prefix subtrees contains three values: $item - name$ represents the node contains that item, *count* is the number of sequences represented by the section of the path reaching the node and $node - link$ links to the next node of the same item in another branch of the FUSP-tree. The FUSP-tree contains a Header-Table which stores frequent items, their count and the link to the first occurrence in the tree corresponding to the item. This table assists on finding appropriate items or sequences in the tree.

Fournier-Viger et al. proposed CM-Spade in 2014 [11]. In their work, they proposed the CMAP data structure to store co-occurrence information of items and used the CMAP to produce a candidate pruning mechanism. Basically, CM-Spade integrates CMAP data structure into the SPADE algorithm. It was reported to have better performance than previous algorithms, SPADE and SPAM. But CM-Spade still suffers from spending much time evaluating candidates that do not exist in the sequence database.

There have been quite a few several efficient algorithms recently for mining frequent itemset from transaction databases [1] such as FP-growth [4], N-List [9] and DiffNodeSets [10].

In 2012, Deng proposed PrePost [9] algorithms. PrePost was based on the N-List structure that was generated from PPC-tree, which was a new structure for representing transaction databases. This data structure saves all information of itemsets. By combining the approach of candidate-generation-and-test and the approach of mining sequence itemset directly without candidate generation, PrePost was reported as an efficient algorithm for mining frequent itemsets. PPC-tree structure includes a root node and a set of children nodes, the structure of each node includes five properties: *item-name*, *count*, *children-list*, *pre-order*, and *post-order*. *Item-name* registers which item this node represents, *count* registers the number of transactions presented by the portion of the path reaching this node, *children-list* registers all children of the node, *pre-order* is the pre-order rank of the node and *post-order* is the post-order rank of the node. PPC-tree structure is like an FP-tree [4].

4 SPPC-Tree Structure

Definition 1. SPPC-tree is a tree data structure. The tree consists of a root and a set of item prefix subtrees as the children of the root. Each node of the tree consists of eight fields: *item-name, count, first-child, first-father, right-sibling, label-sibling, pre-order, post-order*. *Item-name* is the item that the current node represents. *Count* is the number of sequences that have the same path reaching to the current node. *First-child* is a list that contains the first children of the node. *First-father* is the first previous node that is reached from the root node. *Right-sibling* is the first sibling node of the current node. *Label-sibling* is a list of nodes that have the same *item-name* even they may be in different branches of the tree. *Pre-order* is a list of pre-order ranks that were generated by pre-order traversal of the tree. *Post-order* is a list of post-order ranks that were generated by post-order traversal of the tree. SPPC-tree is derived from PPC-tree [9]. However, there are two differences between SPPC-tree and PPC-tree:

1. The support of frequent item is not the sum of all counts of nodes with same item name on SPPC-tree.
2. The *item-name* of an item can appear more than in one node in the same branch of SPPC tree.

Based on Definition 1, an SPPC-tree can be built by the following algorithm.

Algorithm 1 (Building an SPPC-tree)
Input: A sequence database *SDB* and a minimum support ξ.
Output: An SPPC-tree and the set of frequent items *F1*.
Procedure: *Construct-SPPC-tree* (*SDB*, ξ)
[Finding frequent items in the database]
1: Scan *SDB* once to find *F1*, the set of frequent items, with their supports $\geq \xi$.
[Start building SPPC-tree]
2: Create an SPPC node, called S_r, and assign it as a root node.
3: **for** each sequence *Seq* in *SDB* **do**
4: Remove the infrequent items from *Seq* and let p be the remnants of *Seq*. Thus, p is a sequence that only contains frequent items.
[Start inserting the sequence into the tree]
5: **for** each item in p **do**
6: **if** S_r has a child N such that N.*item-name* = p.*item-name* **then**
7: N.count++;
8: **else**
9: create a new node N with the default value;
10: **if** N.*right-sibling* == *null* **then**
11: add new node N to *first-child* list of S_r;
12: **else**
13: add new node N to *right-sibling* list of S_r;
14: **end if**
15: **end for**
16: **end for**
[Adding Pre-Post code after building the tree]
17: Traverse the SPPC-tree with pre-order and post-order traversals to generate the *pre-order* and the *post-order* values for each node.

For example, assuming that we use an example sequence database *SDB* in Table 1 with minimum support threshold ξ = 0.5. First, we convert the value of minimum support from a double value to an integer value: 4 * 0.5 = 2. Then, we scan *SDB* to find the frequent items with their support count greater than or equal to ξ. The final set is *SP*1 = {<1>,<2>,<5>} with their support counts. With all infrequent items eliminated, we have a newly transformed sequence database as in Table 2.

Table 2. The new sequence database with infrequent items already removed

SID	Clickstream
100	<2,5,1>
200	<2,5,1,5,1>
300	<2>
400	<1,5,1,5,1>

Based on the newly transformed database, we build an SPPC-tree by the following steps. First, we create an empty node and assign it as a root node, then we add sequence 100 to the tree. The adding process starts at the root node. From there, each item in the sequence will have a node created and appended to the tree in a sequential order. The first item of the sequence will be appended to the root, the second will be appended to the first node and so on. The tree will look like in Fig. 1a the sequence 100 is added. After which, we add sequence 200 to the tree. Because the subsequence <2,5,1> was previously added into the tree during adding sequence 100, so we increase the count of each same node, the process for the rest of the items the same as adding sequence 100. After the sequence 200 is added, the tree will be like Fig. 1b and so on. However, the sequence 400 does not start with the same start item with other previous sequences. Thus, we create a new branch and add each item in this sequence into the tree like what we did to 100. The tree then will be like in Fig. 1d. Considering the node 2:2, it means that this is the node of item 2 and its support count is 2.

After adding all sequences in *SDB* in Table 2 into the tree, we travel the tree using depth-first search (DFS) algorithm to add *pre-order* and *post-order* for each node. The tree looks like in Fig. 1e, which depicts the final result tree from *SDB* in Table 2 after executing the Algorithm 1. The node (0,4)2:3 mean this is the node of item 2, the *count* is 3, and the *pre-order* and *post-order* of the node is 0 and 4 respectively.

5 Sequential Pattern Mining Using B-Lists

In this section, we describe the idea and step by step of our proposed SMUB algorithm (sequential clickstream mining using B-List). SMUB is a hybrid approach for mining frequent sequences. Main steps of SMUB algorithm include: (1) build SPPC-tree and identify all frequent 1-sequences (2) based on SPPC-tree, conduct the B-List for each frequent 1-sequence (3) mine the remaining frequent k-sequences ($k > 1$). The details of the algorithm are presented in Sect. 5.2.

Definition 2 (SPP-code)**.** Given an SPPC-tree S_{tr} and a node $N \in S_{tr}$, an SPP-code of N is an element represented in the form of (*N.pre-order*, *N.post-order*):*count*.

Definition 3 (B-List of a frequent item; viz., frequent 1-sequence)**.** Given an SPPC-tree, the B-List of a specified frequent item is an ordered set of all the SPP-codes of nodes having the same *item-name* with respect to the frequent item. The SPP-codes are sorted in an ascending order based on their *pre-order* values and the B-List is represented in the form of $(x_1, y_1) : z_1 \rightarrow \cdots \rightarrow (x_n, y_n) : z_n$. For each SPP-code in a B-List, there should always be a node in SPPC-tree that is registered with the SPP-code.

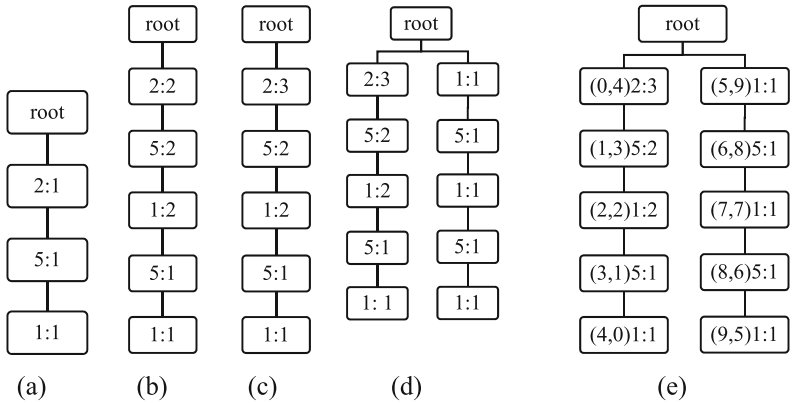

Fig. 1. Step by step SPPC-Tree construction: (a) after adding sequence 100 (b) after adding sequence 200 (c) after adding sequence 300 (d) after adding sequence 400 (e) after adding *pre-order* and *post-order*

Definition 4 (Support count of a B-List)**.** Given a B-List $BL = (x_1, y_1) : z_1 \rightarrow \cdots \rightarrow (x_n, y_n) : z_n$, and $BL_m = BL \backslash \{(x, y) : z \in BL | \exists (x_i, y_i) : z_i \in BL : x \rangle x_i \wedge y < y_i\}$. The support of BL can be calculated via BL_m by the sum of all z_k with $(x_k, y_k) : z_k \in BL_m$. For example, consider the B-List of the frequent 1-sequence <1> in Table 3, its BL_m is (2,2):2 → (5,9):1. So the support count would be 3.

Table 3. The B-Lists of frequent 1-sequences

Frequent 1-sequence	B-List
1	(2,2):2 → (4,0):1 → (5,9):1 → (7,7):1 → (9,5):1
2	(0,4):3
5	(1,3):2 → (3,1):1 → (6,8):1 → (8,6):1

5.1 B-List Generation for *k*-Sequences

Let $BL1$ and $BL2$ be the B-Lists of two k-frequent sequences $P_1 = \langle i_1, i_2, \ldots, i_{k-1}, x \rangle$ and $P_2 = \langle i_1, i_2, \ldots, i_{k-1}, y \rangle$, P_1 and P_2 share the same $(k - 1)$ prefix, the B-List of $(k + 1)$-sequence $P_3 = \langle i_1, \ldots, i_{k-1}, x, y \rangle$ is formed by following the procedure in Algorithm 2. In other words, BL_intersection only works between two frequent k-patterns that share $(k - 1)$ prefix. A special case is that frequent 1-sequences are considered sharing an empty prefix.

```
Algorithm 2 (BL_intersection)
Input: BL1 = (x11, y11): z11 → ⋯ → (x1m, y1m): z1m and
BL2 = (x21, y21): z21 → ⋯ → (x2n, y2n): z2n.
Output: BL3, the B-List of P3.
Procedure: BL_intersection (BL1, BL2)
1:  i ← 1; j ← 1;
2:  while i ≤ m && j ≤ n do
3:      if (x1i < x2j) then
4:          if (y1i > y2j) then
5:              Insert (x2j, y2j):z2j into BL3; j++;
6:          else
7:              j++;
8:          end if
9:      else
10:         i++;
11:     end if
12: end while
```

For example, assuming that we have frequent 1-sequence <5> and we want to generate the B-List of 2-sequence <5,5> . As shown in Table 3, the B-List of <5> is (1,3):2 → (3,1):1 → (6,8):1 → (8,6):1. The generation of the B-List of <5,5> is done by combining the B-List of <5> with itself. First, we check (1,3):2 with every element in the B-List of itself. However, the *pre-order* of the SPP-code (1,3):2 is 1, which is not greater than the *pre-order* of (1,3):2 itself. So we move to (3:1):1. The *pre-order* of (3:1):1 is 3, which is higher than *pre-order* of (1,3):2. The *post-order* of (3:1):1 is 1, which is less than *post-order* of (1,3):2. So (3:1):1 is added to the B-List of <5,5> . Finishing the BL_intersection, we have the B-List of <5,5> , which is (3,1):1 → (8,6):1.

5.2 Mining Clickstream Sequential Patterns

Based on previous definitions, Algorithm 3 illustrates the process of SMUB with high-level pseudocodes.

Algorithm 3 (Mining frequent clickstream patterns)
Input: the minimum support ξ, the sequential patterns 1-sequences SP_1 and set of all frequent 1-sequence B-List BL_1.
Output: The set of all sequential patterns SP.
1: Initialize SP and assign $SP = SP_1$
2: Call mining_L(SP_1, BL_1)
Procedure: mining_L(SP_k, BL_k)
3: Initialize $SP_{k+1} = \emptyset$ and $BL_{k+1} = \emptyset$
4: **for** each pattern P_a in SP_k **do**
5: **for** each pattern P_b in SP_k that share $(k-1)$ prefix with P_a **do**
6: Assuming $P_a =< i_1, i_2, \ldots, i_{k-1}, x >$ and $P_b =< i_1, i_2, \ldots, i_{k-1}, y >$, create $P_c =< i_1, i_2, \ldots, i_{k-1}, x, y >$
7: Create B-List of P_c by calling BL_intersection for B-Lists of P_a and P_b
8: **if** support count of B-List of $P_c \geq \xi$ **then**
9: Put P_c into SP and SP_{k+1}
10: Put B-List of P_c into BL_{k+1}
11: **end if**
12: **end for**
13: **end for**
14: Call mining_L(SP_{k+1}, BL_{k+1})

For example, considering the minimum support ξ = 3, we have SP_1 as the set of frequent 1-sequences <5>, <2> and <1> mined from the example database *SDB* and their respective B-List set BL_1. Running 3, we first join <5> with <5>, <2> and <1> to form 2-sequence candidates <5,5> , <5,1> and <5,2> . By generating B-Lists for aforementioned candidates, we can use them to check for support count of each candidate. Only <5,5> and <5,2> have their support counts higher than ξ, so they are frequent 2-sequences and are added into the set of frequent 2-patterns SP_2. In the same way, <2> is joined with <5>, <2> and <1>, and <1> is joined with <5>, <2> and <1>. The resultant frequent 2-patterns are added into SP_2 and their respective B-Lists are added into BL_2. Recursively, we re-run mining_L procedure with SP_2 and BL_2 and so on, until no candidate can be generated. Figure 2 illustrates the full set of frequent clickstream patterns.

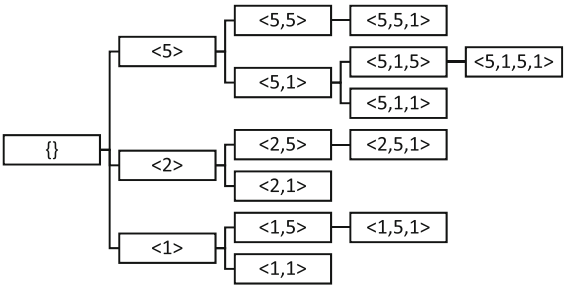

Fig. 2. The tree of frequent clickstream patterns

6 Experimental Evaluation

In this section, we performed experiments to assess the performance of the proposed algorithm. We performed experiments on a computer running Intel Core i7 2.2 GHz CPU, 16 GB memory, and macOS Sierra 10.12.6 operating system. We configured JVM with the flags of -Xmx10G -Xms10G (viz., the maximum memory allowed was 10 GB). The state-of-art algorithm, CM-Spade, for sequential pattern mining that was proved more efficient than previous algorithms, which were GSP, PrefixSpan and FUSP in [11]. So, in this paper we just compared the proposed algorithm, SMUB, with CM-Spade. We use Kosarak, FIFA, MSNBC, and BMS2 datasets (Table 4) for testing performance. We implemented the SMUB in Java 8. The experiments are conducted on each database by decreasing the minimum support thresholds until algorithm took too long time to execute (more than 2000s) or ran out of memory. The running time is the total execution time of the algorithm.

Table 4. Database description

Database	Sequences	Unique items	Average sequence length
Kosarak	990,002	41,270	8.1
FIFA	20,450	2,990	34.74
MSNBC	989,818	17	4.75
BMS2	77,512	3,340	4.62

Figure 3 shows the running time of SMUB and CM-Spade on Kosarak, FIFA, MSNBC, and BMS2 correspondingly. Generally, SMUB ran faster than CM-Spade and the gap kept getting bigger at smaller minimum support. Thus, we can see that SMUB is more efficient than CM-Spade at low minimum support threshold.

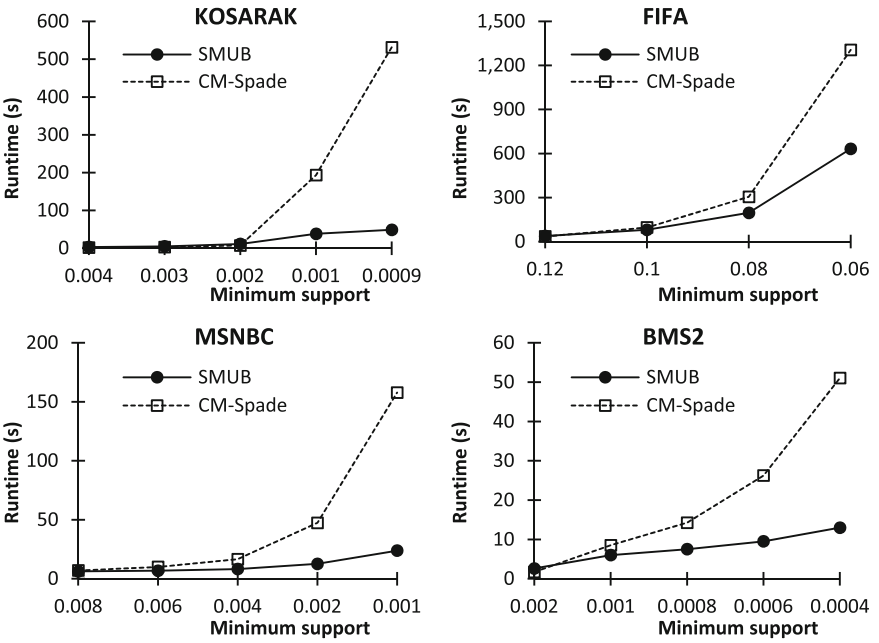

Fig. 3. Runtime of SMUB and CM-Spade

7 Conclusions and Future Work

In this paper, we proposed a novel data structure, B-List, for compressing and storing information for clickstream patterns. Based on B-Lists, we developed an algorithm, SMUB, for fast mining clickstream patterns in clickstream databases. The advantages of the SMUB algorithm compared to other previous algorithms are as follow: First, it uses a compact data structure, B-List, which is usually substantially smaller than the original databases, and thus avoids costly database scans in the subsequent mining processes. Second, counting the support of sequence is transformed into the intersection of B-Lists and it employs an efficient strategy with the complexity of $O(m + n)$ for intersecting two B-Lists, where m and n are the cardinalities of the two B-Lists respectively. We have implemented the SMUB algorithm and studied its performance in comparison with CM-Spade, a well-known sequential pattern mining algorithm, on a variety of real and synthetic datasets. Our performance study shows that the SMUB algorithm is more efficient than CM-Spade.

In future work, we will further explore our method to fully work with sequential pattern mining problem (viz., there is more than one element in itemsets). We also consider using the parallel approach for SMUB so that it can work even bigger databases.

References

1. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. ACM Sigmod Rec. **22**(2), 207–216 (1993)
2. Agrawal, R., Srikant, R.: Mining sequential patterns. In: The Eleventh International Conference on Data Engineering, pp. 3–14. IEEE (1995)
3. Srikant, R., Agrawal, R.: Mining sequential patterns: generalizations and performance improvements. In: Apers, P., Bouzeghoub, M., Gardarin, G. (eds.) EDBT 1996. LNCS, vol. 1057, pp. 1–17. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0014140
4. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. ACM Sigmod Rec. **29**(2), 1–2 (2000)
5. Zaki, M.J.: SPADE: an efficient algorithm for mining frequent sequences. Mach. Learn. **42** (1–2), 31–60 (2001)
6. Han, J., et al.: PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth. In: The 17th International Conference on Data Engineering, pp. 215–224 (2001)
7. Lin, C.-W., et al.: An incremental FUSP-tree maintenance algorithm. In: Proceedings of 2008 Eighth International Conference on Intelligent Systems Design and Applications, vol. 1, pp. 445–449. IEEE (2008)
8. Bithi, A.A., Ferdaus, A.A.: Sequential pattern tree mining. IOSR J. Comput. Eng. **5**(5), 79–89 (2013)
9. Deng, Z.-H., Wang, Z., Jiang, J.: A new algorithm for fast mining frequent itemsets using N-Lists. Sci. China Inf. Sci. **55**(9), 2008–2030 (2012)
10. Deng, Z.-H.: DiffNodesets: an efficient structure for fast mining frequent itemsets. Appl. Soft Comput. **41**, 214–223 (2016)
11. Fournier-Viger, P., Gomariz, A., Campos, M., Thomas, R.: Fast vertical mining of sequential patterns using co-occurrence information. In: Tseng, V.S., Ho, T.B., Zhou, Z.-H., Chen, A.L. P., Kao, H.-Y. (eds.) PAKDD 2014. LNCS (LNAI), vol. 8443, pp. 40–52. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06608-0_4