Efficient algorithms for mining clickstream patterns using pseudo-IDLists

Efficient algorithms for mining clickstream patterns using pseudo-IDLists

Huy M. Huynh[a], Loan T.T. Nguyen[b,g], Bay Vo[c,*], Unil Yun[d], Zuzana Komínková Oplatková[e], Tzung-Pei Hong[f]

[a]*Institute of Research and Development, Duy Tan University, Da Nang 550000, Viet Nam*

[b]*School of Computer Science and Engineering, International University, Ho Chi Minh City, Viet Nam*

[c]*Faculty of Information Technology, Ho Chi Minh City University of Technology (HUTECH), Ho Chi Minh City, Viet Nam*

[d]*Department of Computer Engineering, Sejong University, Seoul, Republic of Korea*

[e]*Faculty of Applied Informatics, Tomas Bata University in Zlín, Nám. T.G. Masaryka 5555, Zlín, Czech Republic*

[f]*Department of Computer Science and Information Engineering, National University of Kaohsiung, Kaohsiung, Taiwan*

[g]Vietnam National University, Ho Chi Minh City, Viet Nam

** Corresponding author E-mail addresses: huy.hm88@gmail.com (H.M. Huynh), nttloan@hcmiu.edu.vn (L.T.T. Nguyen), vd.bay@hutech.edu*

Abstract

Sequential pattern mining is an important task in data mining. Its subproblem, clickstream pattern mining, is starting to attract more research due to the growth of the Internet and the need to analyze online customer behaviors. To date, only few works are dedicately proposed for the problem of mining clickstream patterns. Although one approach is to use the general algorithms for sequential pattern mining, those algorithms' performance may suffer and the resources needed are more than would be necessary with a dedicated method for mining clickstreams. In this paper, we present pseudo-IDList, a novel data structure that is more suitable for clickstream pattern mining. Based on this structure, a vertical format algorithm named CUP (Clickstream pattern mining Using Pseudo-IDList) is proposed. Furthermore, we propose a pruning heuristic named DUB (Dynamic intersection Upper Bound) to improve our proposed algorithm. Four real-life clickstream databases are used for the experiments and the results show that our proposed methods are effective and efficient regarding runtimes and memory consumption.

Keywords: Sequential pattern mining, clickstream pattern mining, candidate pruning, vertical format

1. Introduction

Pattern mining is a practical problem, as it discovers useful and interesting patterns among chaotic data. Sequential pattern mining, first proposed in Agrawal and Srikant [**1**], is one of the most popular variants of pattern mining. Its task is to mine all frequent sequential patterns whose frequencies of appearance are equal to or larger than a minimum threshold in sequence databases. Many researchers have developed algorithms for this problem or its variants [**2-10**], and have been applying them to various domains, ranging from analyzing and mining patterns such as the purchases of customers [**11**], learner result predictions [**12**], and learning resource recommendations [**13**], to analyzing clickstream-type patterns [**14-20**] such as DNA sequences, event sequences or clickstream sequences on online stores. These algorithms can be categorized into two main groups based on the representative data structure they use, which are a horizontal or vertical data format. Some of the basic and well-known algorithms in the horizontal format group include GSP [**21**], FreeSpan [**22**] and PrefixSpan [**23**], of which PrefixSpan is considered the most effective. For the vertical format group, well-known examples include SPADE [**24**], SPAM [**25**] and the recently improved version CM-SPADE [**26**]. According to a number of earlier works [**24-26**], the vertical format group has advantages with regard to calculating supports without costly database scans and has very good overall performance. Moreover, CM-SPADE was proved [**26**] to be one of the most efficient of these algorithms regarding runtimes.

Clickstream analysis is of interest to website owners, as it can help them analyze the browsing behaviors of online customers. Although many vertical format algorithms have been proposed for pattern mining, most of them focus on mining (general) sequential patterns, in which a sequence contains many transactions and each transaction contains many items. However, each transaction in a clickstream sequence only contains one item. Hence, using vertical general algorithms (e.g. CM-SPADE) that are meant for sequential pattern mining to mine clickstream patterns would require more resources than necessary with a dedicated approach, and cause a potential decline in performance. The reason is that vertical format algorithms tend to repeatedly copy and store duplicate data. For example, CM-SPADE uses a bitmap to represent where a pattern appears in the database in a sequence. While the algorithm is processing, CM-SPADE tends to copy and store a part or the whole of this bitmap multiple times to represent data for other patterns. On small databases, the maximum memory consumption can be handled, as the duplicate data usually occur at a low rate. However, on large databases the maximum memory requirements to run vertical algorithms can grow out of hand, as the duplicate information occurs frequently and can take up a large amount of memory. The reason is that the search space is extremely large and the vertical format group uses a generate-candidate-and-test approach, which requires large numbers of pattern candidates to be generated. Before the algorithm can test and determine the candidates to discard, the algorithm first stores all of the candidates' information in the memory. The more candidates there are, then the more likely that duplicate information will be obtained, and this can greatly increase the memory requirements. Additionally, large numbers of the generated candidates do not exist in databases and many of them are discarded when the mining algorithms are used. The runtime for creating and storing the information needed to test the candidates can be quite long, so reducing the number of candidates can significantly reduce this. The challenges here are to find a way to reduce the duplicate information to ease the memory problem and to reduce the number of candidates generated to improve the runtime for the vertical algorithms that are used for clickstream pattern mining on large databases.

In this paper, we address the issues outlined above with our proposed data structures and pruning heuristic. Our contributions are as follows:

- We propose a novel data structure named pseudo-IDList that is more suitable for mining clickstream patterns. This avoids copying repetitive data and instead uses indices to obtain data for new patterns from existing data. Based on this data structure, we propose CUP (Clickstream pattern mining Using Pseudo-IDList), a vertical format algorithm, for mining clickstream patterns.
- We propose a pruning heuristic named DUB (Dynamic intersection Upper Bound constraint) to effectively prune candidates. Following this, we propose an implementation technique that uses bitmaps for the heuristic. The implementation dynamically shrinks the bitmaps based on the proposed pseudo-IDList.
- We perform an experimental evaluation of our proposed methods on four real-life clickstream databases and prove that CUP and DUB are effective.

The rest of the paper is organized as follows. We present some related works in Section **2**. Section **3** presents some definitions, existing concepts and the problem of clickstream pattern mining. Section **4** describes our proposed algorithm, data structures and pruning heuristic in detail. Section **5** deals with experiments and discussions, while the last section is devoted to our conclusions and future work.

2. Related works

The first variant of pattern mining was frequent itemset mining proposed by Agrawal et al. [**27**]. Following this, the authors extended the previous problem to introduce the problem of sequential pattern mining [**1**]. As sequential pattern mining started to gain more attention from researchers, many effective algorithms were developed. They can be categorized into two main kinds: the horizontal or the vertical data formats.

Table 1 An example of a horizontal clickstream database.

UCID	User clickstream
100	⟨b, c, a, a, b, c, d, b, c, b, a⟩
200	⟨b, e, f, f, c, f, b⟩
300	⟨b, a, d, a, b⟩
400	⟨a, e, f, c, b, c, b⟩
500	⟨d, a, a, b, d⟩

The first group uses a horizontal database (e.g. **Table 1**) in which each row is assigned information about a sequence id as well as a list of positions where the pattern appears in the sequence. The most popular algorithms in this family are GSP [**21**], FreeSpan [**22**] and PrefixSpan [**23**]. FreeSpan uses a frequent item matrix to keep track of frequent items and reduces the size of the databases gradually. With each time the databases are reduced (in what are known as projection processes), the frequent patterns grow in size, while the reduced databases become smaller and the database scans faster. PrefixSpan is an improved version of FreeSpan that uses a prefix projection method, and has been developed into more advanced algorithms. For example, Kessl [**28**] developed a parallel algorithm based on PrefixSpan and probabilistic load balancing.

The second group uses a vertical database format (e.g. **Fig. 1**) in which each item or pattern has an individual data structure to indicate the sequence in which the item or pattern appears and its position in the sequence. In general, this group has a better overall performance than the first group [**26**]. Some of the popular algorithms in this family include SPADE [**24**], SPAM [**25**], PRISM [**29**], and more recently CM-SPAM and CM-SPADE [**26**]. SPADE [**24**] is one of the early and efficient algorithms in the vertical family. It is based on the concept of equivalence class and the decomposition of sublattices to divide the entire lattice into separate pieces; each piece can then be fitted into computer memory to be calculated. Ayres et al. [**25**] proposed the SPAM algorithm, which introduced a depth-first search strategy to generate pattern candidates by extending a node with either an item or a new itemset. Additionally, SPAM uses bitmap data structure to store position information of patterns and encode a given vertical database as vertical bitmaps. PRISM [**29**] uses a special version of vertical databases based on primal block encoding. More recently, Fournier-Viger et al. [**26**] proposed a data structure called CMAP (i.e. a co-occurrence map), which stores co-occurrence information across a given database to help discard redundant candidates more efficiently. CMAP is integrated it into SPAM and SPADE to make two improved algorithms, CM-SPADE and CM-SPAM. The improved algorithms have been reported as offering significant performance improvements over previous state-of-the-art methods.

In some cases, some other requirements during mining are needed by users. Hence, researchers have begun to find more ways to include or alter the requirements of frequent patterns. One interesting approach is to apply multiple constraints or adding weights. For example, the approaches in Van, Vo and Le [**2**], among other works [**30-34**], include constraints such as inter-constraints, weight constraints or maximal constraints. Sometimes, users may want to discover a certain number of top patterns, thus instead of relying on minimum thresholds researchers proposed methods to select the top-k frequent patterns [**35,36**]. Alternatively, instead of mining normal patterns, users may want a more compact patterns, such as closed patterns [**37,38**], which may contain information about more than one normal pattern, or generators [**39,40**], which are the smallest subsequences that characterize groups of sequences in a sequence database.

Problems that are also considered related to clickstream pattern mining are frequent substring mining, n-gram extraction, and skip-gram extraction. Frequent substring mining and n-gram extraction can be seen as consecutive clickstream pattern mining, as the patterns do not allow skips or gaps, as in the original clickstream mining. For example, considering a genome sequence ABDABCABDA, a frequent substring pattern can be one of AB, ABD, or BDA but not ABA nor ADA. There are also distinct characteristics between frequent substring mining and n-gram extraction. First, the support count of frequent substring patterns is the number of occurrences of the patterns, unlike clickstream mining, which has its support calculated by the number of user clickstreams that contain the patterns. Similarly, the support of patterns in n-gram extraction is the number of documents that contain the n-gram patterns, in which if we consider a document a user clickstream then the support count formula is the same as in clickstream mining.

There are several notable works on substring mining [4147]. In Vilo [**41**], the author proposed a method that constructs suffix tries by scanning the string databases. The suffix tries contain information on substrings and their support. In De Raedt et al. [**42**] and Lee and De Raedt [**43**], two algorithms called VST and FAVST were proposed based on version space trees, which are a special version of suffix trees. Those earlier algorithms, however, are theoretically reported as space inefficient. Therefore, later works aim to reduce space usage. For example, the work in Fischer, Heun, and Kramer [**44**] used suffix arrays instead of suffix trees or version space trees, while Fischer, Makinen, and Valimaki [**45**] proposed multiple optimizations for suffix arrays to further

reduce the space use. Additionally, some authors extended the problem of substring mining to suit certain needs in real-world problems, such as finding approximate substring patterns instead of fully matching [**47**] or finding substrings in uncertain sequence databases [**46**]. For n-gram mining, there have been works such as Berberich and Bedathur [**48**], and Utama and Distiawan [**49**], in which the aim is mining n-gram in large databases. In the former [**48**], the authors proposed SUFFIX-s, which is an algorithm that is tuned for MapReduce. In the latter [**49**], the authors proposed Spark-gram, which is an algorithm based on the SUFFIX-s. Spark-gram is adapted to run on Spark for faster mining of short n-grams with a lower threshold.

Skip-grams are a special type of n-gram that allow skip elements, but these must be defined. Van Gompel and van den Bosch [**50**] used the term skip-gram for fixed-length n-grams with skip elements and flex-gram for dynamic length ones. For example, the pattern A*D*B is a skip-gram with two skip positions that are denoted by "*''. If * is strictly a single element, then it is called a skip-gram, otherwise, if * denotes a group of elements, then the skip-gram is a flex-gram. Skip-grams and flex-grams are important for skip-gram modeling, which is a form of machine learning for natural language processing. However, to the best of our knowledge, there has never been a real method to extract frequent skip-grams and flex-grams. The reason is that the models often do subsampling of the text for the skip-grams and cannot really take the frequency of the patterns into account due to the large size of training data and the iterative process of model training. However, if we somehow can extract those frequent skip-grams along with their frequencies and integrate them into the model training process, perhaps they can improve the models' performance.

Clickstream pattern mining has become important because of its wide range of real-life applications (e.g. web log analysis, intrusion detection); however, most previous works apply existing general sequential pattern algorithms to mine such patterns. For example, Cooley, Mobasher, and Srivastava [**14**], and Demiriz [**15**] used general sequential pattern algorithms to discover interesting clickstream patterns in user browsing behaviors, while Ting et al. [**16**] analyzed the unexpected clickstream patterns of users to support and improve website design. Setiawan and Yahya [**51**] sequential rules mined from event logs and used them to analyze human behaviors during the production process of software factories. Law et al. [**52**] incorporate sequential pattern mining in their proposed methods for recursive event sequence, which is a type of clickstream sequence, to support query analysis. In healthcare and clinic, there are some works such as [**53**] or [**54**] that used clickstream mining as a part of their proposed visual analytic techniques. Their propose methods enabled on-demand analytics, interactive visualization, and exploratory analysis of clinical events to help patients' medical conditions. In the security domain, Pramono [**18**], and Lee and Stolfo [**55**] built a detection classifier based on the association rules generated from frequent clickstream patterns in system calls . However, there are a few works that proposed algorithms targeting clickstream patterns. For example, Dalmas [**17**] proposed TWINCLE to mine patterns in rich event log databases to help optimize organizational processes. More recently, Van, Yoshitaka, and Le [**19**] proposed the MWAPC and EMWAPC algorithms to specifically mine web access patterns with a super-pattern constraint. In the current paper, we thus put more focus on exclusive approaches for mining clickstream patterns with the proposed data structures and pruning heuristics.

3. Problem definitions

In this section, we define the problem of clickstream pattern mining and present some related definitions.

Let I = {i_1, i_2,..., i_n} be a set of integer values and each value represents an event (e.g. a click on a website's URL), **a clickstream** (sequence) X = <x_1, x_2,..., x_m) is a sequence of events, where $x_i \in I$. An event can appear more than once in the same sequence at various positions. The event's location denotes the order in which the event happens. The number of events in a clickstream denotes the **length** of the clickstream, and the clickstream with a length k is called a k-clickstream.

Let X = <x_1, x_2,..., x_{k-1}, x_k) and Y = <y_1 y_2,..., y_{j-1}, y_j) be two clickstream, $X \sqsubseteq Y$ indicates that Y is a **super clickstream** of X or X is a **sub-clickstream** of Y if there exists integers $1 \leq t_1 < t_2 < \bullet \bullet \bullet < tj \leq j$ such that $x_1 = y_{t1}$, $x_2 = y_{t2}$,...,$x_j = y_{ti}$. A clickstream Z = <z_1, z_2,..., z_{k-1}) is called a (k — 1)-**prefix** of X if $x_1 = z_1$, $x_2 = z_2$,..., $x_{j-1} = z_{j-1}$. In other words, Z is also a sub-clickstream of X and X = <Z, x_k).

A user clickstream is a clickstream generated by a user via various actions such as browsing the internet or navigating folders on a computer. A **clickstream database** CDB is a collection of user clickstreams, each of which is assigned with a UCID (user clickstream id) (**Table 1**). A (**clickstream**) **pattern** is a subclickstream of one or more user clickstreams. In other words, we can say that ''the pattern **appears** in one or more user clickstreams'', or ''one or more user clickstreams contain the pattern''. The **support** (**count**) of a pattern X, denoted by supcount(X), is the number of user clickstreams that are the pattern's super clickstreams, i.e. supcount(X) = $|\{Y | \forall Y \in CDB : X \sqsubseteq Y\}|$. Given a **minimum support threshold** ϒ, a pattern X is **frequent** if its support is equal to or larger than ϒ, i.e. supcount(X) ≥ ϒ . A **pattern candidate** is a clickstream that is formed from two patterns that may or may not appear in the database that is being examined.

The downward closure property states that all of the subsequences of a frequent sequence are also frequent. A clickstream is one kind of sequence, and thus this property can be used. This property also implies that: (1) All sub-clickstream patterns of a frequent clickstream pattern have their supports equal to or higher than the frequent pattern's support. (2) Let X be a clickstream pattern and Y is an X's super clickstream pattern. If S_X and S_Y are two sets of user clickstreams that are super clickstreams of X and Y respectively, S_Y is a subset of S_X (denoted by $S_y \subseteq S_X$). In other words, Y only appears in user clickstreams that contain X. The downward closure property is a well-known heuristic for pruning candidates in previously existing algorithms.

Problem definition. The problem of clickstream pattern mining is to find an entire set of frequent clickstream patterns that exist in a given clickstream database CDL and satisfy a minimum support threshold y .

Example 1. Given the CDB in **Table 1** and a minimum support threshold ϒ = 3, the user clickstream (with UCID = 200) is (b, e, f, f, c, f, b>. Clickstreams (b, c, b> and (e, c, f) are both subclickstreams of user clickstream 200 and are 3-patterns, whereas (f, e, f) is not. Pattern (b, c, b) appears in user clickstreams 100, 200 and 400, thus its supcount((b, c, b>) = 3. Pattern (e, c, f) only appears in user clickstream 200, thus its supcount((e, c, f>) = 1. (b, c, b) is a frequent clickstream pattern as its support = ϒ = 3, whereas (e, c, f) is infrequent because its support = 1 < ϒ = 3.

4. The CUP algorithm

In this section, we describe our proposed algorithm named CUP (Clickstream pattern mining Using Pseudo-IDList) and its components. Like most of the general sequential mining algorithms, CUP also uses the idea of traversing a sequence lattice in a depth-first search manner to enumerate the entire set of frequent patterns. However, most of the general sequential mining algorithms must copy repetitive data multiple times during execution, while our algorithm avoids this issue.

Additionally, they only use a single type of IDList while we use two types of IDList, including the (semi-vertical) data IDList and pseudo-IDList. The former (whose variants are also used in CM-SPAM and CM-SPADE) directly stores the position information of a pattern while the latter stores indexed data of the former and indirectly express the data of the pattern. In other words, pseudo-IDList retrieves the actual data of a pattern via the data IDList. We also propose a tighter upper bound heuristic for pruning candidate patterns. Generally, the key mining processes of CUP (illustrated in pseudocode in Algorithm 2 in Section **4.5**) contain the following main steps.

- **Step 1**. Identifying frequent 1-patterns in a given horizontal database and transforming the horizontal database to a set of data IDLists (i.e. a vertical database as illustrated in **Fig. 1**). Each data IDList holds the information necessary for frequent 1-patterns (more details of this are given in Section **4.1**).
- **Step 2**. Generating pattern candidates with length (k + 1) from two given frequent k-patterns that share the same (k — 1)-prefix. Frequent 1-patterns are considered as sharing the 0-prefix (or an empty prefix). The proposed pruning heuristic (Section **4.3**) is used right after this to discard redundant candidates.
- **Step 3**. Creating data for the generated candidates and checking support values. Any candidate that does not satisfy the minimum support threshold y is discarded. We obtain their support via the proposed pseudo-IDList (Section **4.2**) instead of scanning through the horizontal database. Each candidate is assigned with one pseudo-IDList and the pseudo-IDList is created by joining data from two IDLists of its parents. Producing IDLists generally requires most of the processing time in the algorithm. The process then loops back at the generating candidate step (step 2) until no candidates can be found.

4.1. (Semi-vertical) data IDList

A (semi-vertical) data IDList is a data structure that records where a pattern appears in the horizontal database (i.e. which user clickstreams contain the pattern and in which positions the pattern appears). Following [**24**], the positions where the pattern appeared in a user clickstream are only recorded as the positions of the last item of the pattern. For example, given user clickstream 400 = (a, e, f, c, b, c, b) in **Table 1** and a pattern (e, f, c), its position list would be (**4, 6**). A set of data IDLists is called a vertical database (**Fig. 1**). A data IDList contains the following elements:

- P: a pattern which the data IDList records the position information in a database.
- M: a lookup matrix with three columns {Data id, UCID, Position list}. UCID is its corresponding user clickstream id in the horizontal database. The position list contains positions where the last item of the pattern occurs in the user clickstream and it is ordered in ascending order. A data id is a locally assigned id for the corresponding (user) clickstream in the IDList. Different IDLists may assign different data ids for the same user clickstream in the database. The use of data ids is explained in Section **4.3**.
- The support supcount(P) is the number of row elements in M.

If X is a super clickstream pattern of Y and the last event. in X is equal to the last event in Y, then for every clickstream that contains both X and Y, the position list of X, if it exists, is always a sub-list of the position list of Y. Additionally, the first element in X's position list is always equal to or larger than the first element's value in Y's position list. For example, with two patterns X = ⟨c, b, c⟩, Y = ⟨b, c⟩ and using

the example database in **Table 1**, we can see that the position list of X, which is ⟨6, 9⟩, is a sub-list of Y, which is ⟨2, 6, 9⟩. This property is thus exploited by our proposed pseudo-IDList in the next section.

4.2. Pseudo-IDList

A pseudo-IDList holds an index matrix. While the pseudo-IDList holds no actual position information of a pattern, it retrieves (actual) data and indirectly represents a data IDList for the pattern through a different data IDList with its index matrix. It includes the following elements.

- P: a pattern which the pseudo-IDList records the location information in a database.
- DIP (data IDList pointer): a pointer to a data IDList where it can retrieve the position data of P. The data IDList is always of that frequent 1-pattern whose value is equal to the last item in P. For example, if the pattern is (a, b, c) then DIP would point to the data IDList of 1-pattern (c). We only used frequent 1-patterns' data IDLists because every pseudo-IDList of k-pattern (k > 1) can use the data from frequent 1-patterns' data IDLists.
- M: an index matrix contains indices of the data IDList in three columns {Local id, Data id, Start index}. In a similar way to IDList, a local id is a locally assigned user clickstream id for the corresponding (user) sequence in this pseudo-IDList. A data id is a value that matches a corresponding data id in the data IDList. Given a position list PL corresponding to the local id in data IDList and a start index that is a location in the position list, the position list for the pattern P is a sub-list of PL that starts at the start index to the end of PL.
- The support supcount(P) is the number of row elements in M.

M

Pattern: <a>

Data id	UCID	Position list
1	100	3, 4, 11
2	300	2, 4
3	400	1
4	500	2, 3

Pattern: <b>

Data id	UCID	Position list
1	100	1, 5, 8, 10
2	200	1, 7
3	300	1, 5
4	400	5, 7
5	500	4

Pattern: <c>

Data id	UCID	Position list
1	100	2, 6, 9
2	200	5
3	400	4, 6

Fig. 1. Data IDLists of frequent 1-patterns (i.e. the vertical format database).

While data IDLists are only created once for frequent 1-patterns at the start of the algorithm, the pseudo-IDLists are created for all possible pattern candidates throughout the mining process. This also means 1-patterns have both data IDLists and pseudo-IDLists while k-patterns (k ≥ 2) only have pseudo-IDLists. The pseudo-IDList of frequent 1-patterns are created to unify the process of creating pseudo-IDLists for k-patterns (k ≥ 2), as Algorithm 1 requires an input of two pseudo-IDLists of two frequent patterns. All pseudo-IDLists (**Fig. 4**) of frequent 1-patterns have a start index value equal to one, the value of local id matches the value of data id in the same row, and they both increase by one starting from one. **Fig. 2** illustrates the pseudo-IDLists of two patterns and their corresponding data IDLists in

Fig. 3. **Fig. 5** illustrates how a pseudo-IDList retrieves data and represents the data IDList for pattern (a, c).

Pseudo-IDList creation. In the vertical format group, e.g. (CM-)SPADE, producing the data IDList can repeatedly copy a part of a position list from the parent data IDList. Our proposed pseudo-IDList avoids this by using the data of existing data IDLists. To do this, the pseudo-IDList contains start indices of the position list in an existing data IDList. The procedure for creating a pseudo-IDList is also faster, because it does not repeatedly copy redundant information to the new candidate IDList. It therefore also reduces the runtime and memory footprint. The creation of a pseudo-IDList is described in Algorithm 1.

Example of creating a pseudo-IDList. Assume that we have the patterns X = (a, a and Y = (a, c), their respective index matrices in pseudo-IDLists are denoted as M_X and M_Y (**Fig. 3**) and the matrices of data IDLists that DIP in X and Y's pseudo-IDLists points to are denoted as DIP_M_X and DIP_M_Y. DIP_M_X and DIP_M_Y are in fact the matrix M in data IDLists of patterns (a) and (c) in **Fig. 1**. In order to obtain the pseudo-IDList for pattern Z = (a, a, c), we do the following steps, as in Algorithm 1.

- We initialize the local id counter $local\_id_Z$ for M_Z to 1 then we start to iterate both matrix M_X and M_Y. The first row in M_X is $\{local\_id_X, data\_id_X, start\_index_X\} = \{1,1, 2\}$ and in M_Y is $\{local\_id_Y, data\_id_Y, start\_index_Y\} = \{1, 1, 2\}$.
- As $data\_id_X = 1$ and $data\_id_Y = 1$, we look for the row in DIP_M_X that has data id = 1, which is {Data id = 1, UCID = 100, Position list = (3, 4,11)}, and get the UCID = 100 value in that row to assign to UCIDX. Similarly, for DIP_M_Y, we have UCIDY = 100 as the row {Data id = 1, UCID = 100, Position list = (2, 6, 9)} has data id = 1.
- Because $UCID_X = UCID_Y = 100$, the rows contain duplicate information. We start searching for the start index of duplicate information.
- We have pos_list_X = (3, 4,11) and pos_list_Y = (2, 6, 9) as both of them are the position lists in the rows with data id = 1 in DIP_M_X and DIP_M_Y respectively. Because $start\_index_X = 2$, we start at the second element in pos_list_X, which is $e_X = 4$. We then go through pos_list_Y to see which element has values greater than e_X . Starting with the second element in pos_list_Y (because we have $start\_index_Y = 2$), we have $e_Y = 6$. Because $e_Y = 6 > e_X = 4$, the second element in pos_list_Y is the start index of the duplicate information of the first row in M_Z in the pseudo-IDList of Z. Thus, we have $start\_index_Z = 2$. We add a row $\{local\_id_Z, data\_id_Y, start\_index_Z\} = \{1,1, 2\}$ to M_Z and increase $local\_id_Z$ to 2.
- We move to the second row in both M_X and M_Y. Repeating the previous steps, we have $UCID_X = 300$ and $UCID_Y = 400$. Because $UCID_X = 300 < UCID_Y = 400$, we move to the third row in M_X while we are still in the second row in M_Y . We now have $UCID_X = 500$ and $UCID_Y = 400$, but $UCID_X = 500 > UCID_Y = 400$ and thus we should move to the third row in M_Y. However, there are no rows left in MY so the loop searching for duplicate information stops.
- After this, we have a pseudo-IDList of Z consisting of the following elements: DIP points to data IDList of (c) and M_z = ({1, 1, 2}.

4.3. Dynamic Intersection Upper Bound Constraint Pruning Heuristic (DUB)

Let X and Y be two frequent clickstream patterns, and $P_1 = (X, last_Y)$ and $P_2 = (Y, last_X)$ be two candidates generated from X and Y. Let S_X and S_Y be sets of user clickstreams that contain X and Y, respectively, and let $S_\cap = S_X \cap S_Y$.

Theorem 1 *(Dynamic Intersection Upper Bound Constraint (DUB)). The number of sequences in the intersection set $S_\cap$, denoted as supcount($S_\cap$), is greater or equal to the support count of either P_1 or P_2;*

i.e. supcount($S_\cap$) ≥ supcount(P_1) and supcount($S_\cap$) ≥ supcount(P_2). This is considered a tighter upper bound than the downward closure property where supcount(X) ≥ supcount($S_\cap$) ≥ supcount(P_1) and supcount(Y) ≥ supcount($S_\cap$) ≥ supcount(P_2).

Proof. Let Z be a (k — 1)-prefix that is shared by both X and Y. We can formulate that P_1 = (Z, $last_X$, $last_Y$) and P_2 = (Z, $last_Y$, $last_X$), X = (Z, $last_X$) and Y = (Z, $last_Y$). If a user clickstream contains P_1 or P_2, consequently X = (Z, $last_X$) and Y = (Z, $last_Y$) must obviously be included in the same user clickstream.

Pattern: <a, a>

Data id	UCID	Position list
1	100	4, 11
2	300	4
4	500	3

Pattern: <a, b>

Data id	UCID	Position list
1	100	5, 8, 10
3	300	5
4	400	5, 7
5	500	4

Pattern: <a,c>

Data id	UCID	Position list
1	100	6, 9
3	400	4, 6

Fig. 2. Supposed-to-be data IDLists of some frequent 2-patterns that share prefix (a).

M

Pattern: <a, a>
DIP: <a>

Local id	Data id	Start index
1	1	2
2	2	2
3	4	2

Pattern: <a, b>
DIP: <b>

Local id	Data id	Start index
1	1	2
2	3	2
3	4	1
4	5	1

Pattern: <a, c>
DIP: <c>

Local id	Data id	Start index
1	1	2
2	3	1

Fig. 3. The respective pseudo-IDLists of frequent 2-patterns in Fig. 2.

Pattern: <a>		
DIP: <a>		
Local id	Data id	Position list
1	1	1
2	2	1
3	3	1
4	4	1

Pattern: <b>		
DIP: <b>		
Local id	Data id	Position list
1	1	1
2	2	1
3	3	1
4	4	1
5	5	1

Pattern: <c>		
DIP: <c>		
Local id	Data id	Position list
1	1	1
2	2	1
3	3	1

Fig. 4. Pseudo-IDLists of respectively frequent 1-patterns in Fig. 1

Pattern: <a,c>		
DIP: <c>		
Local id	Data id	Start index
1	1	2
2	3	1

Pattern: <c>		
Data id	UCID	Position list
1	100	2, **6, 9**
2	200	5
3	400	**4, 6**

Pattern: <a,c>		
Local id	UCID	Position list
1	100	6, 9
2	400	4, 6

Fig. 5. Data retrieval of a pseudo-IDList of pattern (a, c). The first row of the pseudo-IDList of (a, c) will search in the first row of data IDList of pattern (a). The start index in the first row is 2, so it retrieves the position (6, 9) of the full list (2, 6, 9). Similarly, the second row in the pseudo-IDList has start index of 1, so it retrieves the full position list (4, 6).

Algorithm 1. Pseudo-IDList creation.

Input: pseudo-IDList of X and Y
Output: pseudo-IDList of Z

1: $M_X \leftarrow$ the matrix M in pseudo-IDList of X // M_X holds all the index data of the pseudo-IDLists of X
2: $M_Y \leftarrow$ the *matrix* M in pseudo-IDList of Y // M_Y holds all the index data of the pseudo-IDLists of Y
3: Initialize the pseudo-IDList of Z to initial values
4: $M_Z \leftarrow$ the matrix M in pseudo-IDList of Z // M_Z will hold all the upcoming index data of the pseudo-IDList of Z after the current algorithm finishes
5: $DIP\_M_X \leftarrow$ the matrix M in data IDList that DIP of X points to // DIP_M_X holds all the UCIDs, local ids and positions lists in the data IDLists of frequent 1-pattern that DIP in X's pseudo-IDList point to
6: $DIP\_M_Y \leftarrow$ the matrix M in data IDList that DIP of Y points to // DIP_M_Y holds all the UCIDs, local ids and positions lists in the data IDLists of frequent 1-pattern that DIP in Y's pseudo-IDList point to
7: $local\_id_Z \leftarrow 1$ // A counter variable that assigns a local id value to each row in matrix M of Z's pseudo-IDList
8: $row\_index_X \leftarrow 1$ // An index variable that helps iterate all rows in M_X
9: $row\_index_Y \leftarrow 1$ // An index variable that helps iterate all rows in M_Y
/* We start iterating both M_X and M_Y at the same time */
10: **while** row_index_X does not exceed the number of rows in M_X **do**
11: **while** row_index_Y does not exceed the number of rows in M_Y **do**
12: $\{local\_id_X, data\_id_X, start\_index_X\} \leftarrow$ the row at row_index_X in M_X
13: $\{local\_id_Y, data\_id_Y, start\_index_Y\} \leftarrow$ the row at row_index_Y in M_Y
14: $UCID_X \leftarrow$ UCID value corresponding to $data\_id_X$ in DIP_M_X // We look up the row that contains $data\_id_X$ in DIP_M_X, then get the UCID value in that row and assign the UCID value to $UCID_X$
15: $UCID_Y \leftarrow$ UCID value corresponding to $data\_id_Y$ in DIP_M_Y // We look up the row that contains $data\_id_X$ in DIP_M_X, then get the UCID value in that row and assign the UCID value to $UCID_Y$
/* If a pair of rows from DIP_M_X and DIP_M_Y contain the same UCID, then it may contain duplicate information. We can start searching the start index of the duplicate position list in those rows. Because the rows are sorted ascendingly based on UCID, if $UCID_X$ is smaller than $UCID_Y$ then we must keep iterating the rows from M_X and vice versa until we find two rows that match */
16: **if** $UCID_X < UCID_Y$ **then**
17: $row\_index_X \leftarrow row\_index_X + 1$
18: **else if** $UCID_X > UCID_Y$ **then**
19: $row\_index_Y \leftarrow row\_index_Y + 1$
20: **else if** $UCID_X = UCID_Y$ **then**
21: $pos\_list_X \leftarrow$ the position list regarding $data\_id_X$ in DIP_M_X
22: $pos\_list_Y \leftarrow$ the position list regarding $data\_id_Y$ in DIP_M_Y
23: $e_X \leftarrow$ the element at $start\_index_X$ in pos_list_X
/* After we find rows containing duplicate information, we start searching for the start index where the duplicate information occurs in the current rows*/
24: **for** each element e_Y starting at $start\_index_Y$ in pos_list_Y **do**
25: **if** $e_Y > e_X$ **then** // The start index of duplicate information is found
26: $start\_index_Z \leftarrow$ the index of e_Y in pos_list_Y
27: Add a row $\{local\_id_Z, data\_id_Y, start\_index_Z\}$ to M_Z // Add new data into the pseudo-IDList of Z
28: Break; // Stop searching in pos_list_Y
29: $row\_index_X \leftarrow row\_index_X + 1$
30: $row\_index_Y \leftarrow row\_index_Y + 1$
31: $local\_id_Z \leftarrow local\_id_Z + 1$
32: DIP of pseudo-IDList of $Z \leftarrow DIP$ of pseudo-IDList of Y // DIP of Z always points to the same data IDList that Y's points to
33: **return** pseudo-IDList of Z

A user clickstream in $S_{\cap}$ must contain both X and Y but it does not state whether $last_X$ appears after or before lastY. In a case VC e $S_{\cap}$, $last_X$ always appears before $last_Y$ (which means C always contains the candidate X), then supcount($S_{\cap}$) = supcount(X). However, C may not always contain X because it does not guarantee that $last_X$ would always appear before $last_Y$ for every C. Thus, supcount($S_{\cap}$) > supcount(X) and similarly, supcount($S_{\cap}$) ≥ supcount(Y).

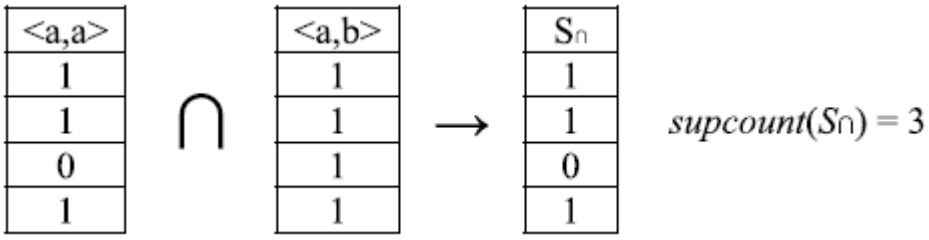

Fig. 6. An example of DUB, where we join DUB bitmaps of (a, a) and (a, b) to create an intersection bitmap. From this, we can find the supcount(Sn) for DUB.

Implementation. The basic idea here is to use a bitmap to represent a set of user clickstream ids in which a frequent pattern appears. Each user clickstream id corresponds to a true bit in the bitmap. As such, the intersection set $S_{\cap} = S_X$ n S_Y can be quickly obtained by the doing operator AND on two bitmaps representing S_X and S_Y.

Naturally, we can use bitmaps with fixed sizes that are proportionate to database sizes. However, while this may work well on small and normal size database, it would cause bad performance and a high memory footprint on large databases. For example, given a database with one million user clickstreams, every bitmap used for DUB must contain one million bits although only a part of the bitmap contains true bits, which is the pseudo-IDList for DUB. As mentioned in the downward closure property, a pattern appears in sequences that also contain the pattern's subsequences. Since prefixes are a special type of subsequence, patterns with the same prefix must also appear in the same sequences containing the prefix. With this assumption, we can use the bitmaps to represent the sets of sequence ids based on the local (user clickstream) ids of their parents (instead of using UCID).

Because pseudo-IDLists shrink over time, those bitmaps also shrink over time, thus reducing the memory needed and improving the heuristic performance. The bitmaps are created during the pseudo-IDList creation. For optimization, the local sequence id column of each data or pseudo-IDList can be removed when all the necessary bitmaps are created, thus saving memory.

Using DUB. We can use check if new candidates violate DUB and discard them during the candidate generation step. If the new candidates do not satisfy DUB, then they are not frequent patterns and thus the later steps (i.e. creating pseudo-IDLists and checking support) are unnecessary. (**Fig. 6**)

4.4. Candidate generation

To generate candidates, we use a similar method to that of (CM-)SPADE. Specifically, given two frequent k-patterns X and Y that share the same (k — 1)-prefix, let $last_X$ and lastY be the two events of X and Y respectively. If $last_X$ is not the same as lastY then a set of two candidates are generated, which

is {(X, lastY), (Y, last$_X$)}. Otherwise, the set only contains one candidate {(X, last$_Y$)}. This step also includes the pruning heuristic that we introduced earlier.

4.5. The CUP algorithm

In this section, we describe CUP in pseudocode in Algorithm 2 and give a running example using the database CDB in **Table 1** and a minimum support threshold y = 3. The entire lattice of patterns is illustrated in **Fig. 7**.

Step 1. We scan the horizontal database once to identify the frequent 1-pattern set, which is {(a), (b), (c)} with the respective support {4, 5, 3}. Data IDLists and pseudo-IDLists of these are also created during the database scan.

Table 2 Summary of databases for experiments.

Database	Database size	Distinct events
FIFA	20,450	2990
BMS2	77,512	3340
Kosarak	990,002	41,270
MSNBC	989,818	17

Step 2. We generate a pattern candidate set {(a, a), (a, b), (a, c)} that shares the 1-prefix (a) by combining (a) with (a), (a) with (b) and (a) with (c). Partial candidate sets {(b, a)} and {(c, a)} that respectively share 1-prefix (b), (c) are also created simultaneously during the combination. Those sets are completely built after we backtrack from traversing (a) and its children. After this, we apply DUB to filter the candidate set of (a). The intersection supports are respectively {3, 4, 2}. We discard candidate (a, c) of (a) because its intersection support = 2 < y = 3. Consequently, we also discard candidate (c, a) of (c) because it shares the same intersection support = 2 as (a,c).

Step 3. As the candidate set of (a), which is {(a, a), (a, b)} is passed to this step, we construct pseudo-IDLists for these candidates based on the pseudo-IDList of (a) and (b). After constructing this, the supports of the two candidates are identified as {3, 4}, both of which > y .As CUP operates in a depth-first search manner, we repeat back to step 2 and use those two frequent 2-patterns to generate 3-candidates. However, those 3-candidates are infrequent, and thus no more candidates can be generated on this branch. We backtrack to step 2 and complete the full candidate set of (b) then (c). The algorithm stops after no more candidates can be found in the (c) branch.

5. Experimental results

In this section, we describe the setups and environments for running experiments as well as report on the results of a performance comparison. The algorithms are CUP (our baseline proposed algorithm with pseudo-IDList and data IDList), CUP+DUB (in which we integrate our pruning heuristic DUB to CUP) and two existing algorithms, PrefixSpan [**23**], CM-SPADE [**26**] and SPAM [**25**]. Based on Fournier-Viger et al. [**26**], PrefixSpan and CM-SPADE are very effective for sequential pattern mining. We would like to use PrefixSpan to represent the horizontal format group and CM-SPADE to represent the vertical

format group, as they are the most effective algorithms among their groups. Additionally, CM-SPADE is the state-of-the-art algorithm and uses a similar concept to IDList, which can be used to compare with ours. Besides runtime (**Fig. 8**) and memory consumption (**Fig. 9**), we also report the effectiveness of our DUB heuristic based on the number of candidates it discards for each database (**Table 3**).

We implement the algorithms in Java language, and the tests are carried out on Windows 10 64 bit edition and JDK 8. The hardware specification includes 2.2 GHz Intel I7 8750H (with the Turbo Boost function deactivated for more stable runtimes) and 16 GB RAM. CM-SPADE and PrefixSpan are obtained via the SPMF package [**56**]. We use a version of PrefixSpan which was implemented in 2016 in the SPMF package, as it was optimized to be better than the previous version 2008 in the same package. We also collect four real-life clickstream databases via SPMF site for our performance benchmark. The description of each database is given in **Table 2**. For optimization reasons, we also integrate the CMAP data structure [**26**] to prune candidates in the baseline CUP algorithm. The way we execute the experiments is to run all algorithms on all test databases while decreasing the minimum support threshold y until one of the algorithms takes too long to execute.

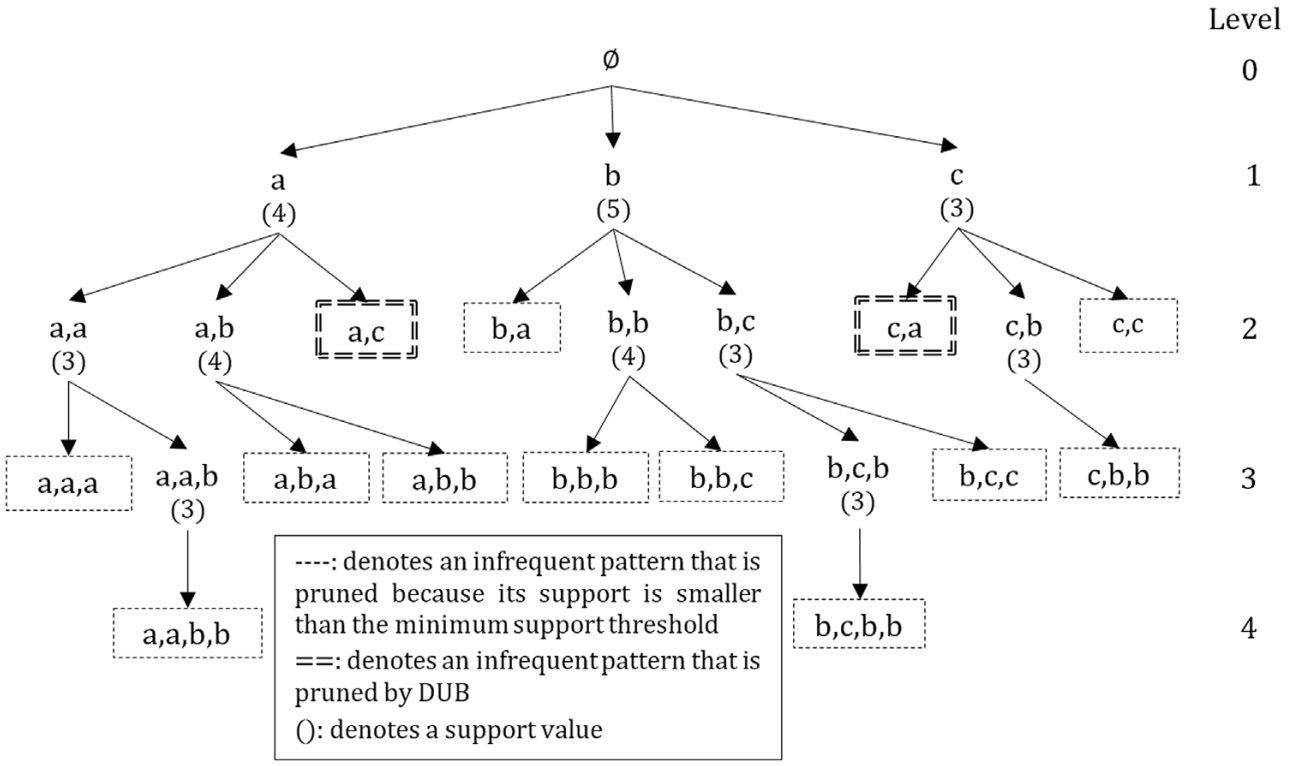

Fig. 7. The lattice of patterns corresponding to the example database.

Performance comparison. On all test databases, CUP(+DUB) outperform SPAM, PrefixSpan and CM-SPADE in terms of execution time, while CM-SPADE runs faster than PrefixSpan on FIFA, Kosarak, and MSNBC. SPAM has the highest runtime among the four algorithms, and it cannot run on Kosarak due to exceeding the allowed memory. On large databases (Kosarak and MSNBC), CUP(+DUB) runs noticeably faster than the other two. When minimum support thresholds are high, not many frequent patterns are produced; the differences in runtime among the four algorithms are small. However, as the minimum support threshold gets smaller, the gaps in runtimes get bigger. The runtimes of PrefixSpan and CM-SPADE also increase dramatically at a faster rate. The reason why CUP(+DUB) works

well is that pseudo-IDLists do not have to do copy operators for redundant data, and it instead this method only produces indices on existing data IDLists. Regarding the memory consumption, CM-SPADE has the highest because it stores repetitive data. CUP(+DUB) uses less memory than PrefixSpan on two databases, FIFA and Kosarak, and roughly the same amount on BMS2. On MSNBC, CUP(+DUB) uses more memory than PrefixSpan at high minimum support thresholds, but it gradually comes close to PrefixSpan at lower minimum support thresholds. This indicates the effectiveness and efficiency of memory consumption and runtime of pseudo-IDLists. CUP(+DUB) manages to achieve good runtimes while still being able to maintain low memory consumption.

Even though the mining process of CM-SPADE and CUP have several similar steps, different IDList data structures make CUP perform much better than CM-SPADE. For CM-SPADE, their IDLists are hashtables that containing tuples of {UCID, Bitmap}. The hash key is UCID and its value is a bitmap. The bitmap is an array of bits, in which each true bit represents a location of the pattern in the user clickstream. The required space is m + nbl for a single IDList in a perfect condition and with a perfect hash function, where m is the number of sequences and nbl is the sum of all the 32 bit blocks used in all the bitmaps. However, in practice, more than m + nbl space is required when we use the available hashtable data structure in Java. Additionally, collisions can occur when inserting new tuples into the CM-SPADE's IDLists, and when the hashtables (i.e. IDLists) are full they need to expand and rehash. When this happens, the runtime can increase. Using the bitmap representation of position lists also has its disadvantages. To represent a single position, we sometimes must use a whole bitmap with unnecessary empty bits. For example, considering a position list with a single position (122), we need a bitmap with 128 bits to represent that position, which is four times the amount of space of a single integer value. It also means that the bitmap iteration for the mentioned case may also take up four times longer to carry out. SPAM also suffers from this issue.

Table 3 Amount of candidate reduction and runtime reduction when DUB is integrated into CUP.

Database	Candidate reduction (%)	Runtime reduction (%)
FIFA	51 to 55	38 to 42
BMS2	68 to 70	34 to 40
Kosarak	42 to 52	13 to 24
MSNBC	40 to 51	21 to 33

Regarding CUP, given that there are two frequent patterns P_1 and P_2, of which the respective pseudo-IDList $Plist_1$ and $Plist_2$ contain m and n numbers of rows. In the algorithm, each row holds a start index to a position list of data IDList. Based on this start index, we can retrieve a part of the position list from the start index to the end of the list. Assume that the numbers of position elements that $Plist_1$ and $Plist_2$ retrieve are M and N, and let P_3 be a candidate pattern that is formed from P_1 and P_2. Creating a new pseudo-IDList $Plist_3$ for P_3 requires O(M + N) in runtime at most. For space complexity, each pseudo-IDList only takes 2 * k numbers of memory blocks constantly, where k is the number of rows in the pseudo-IDList and each block is assumed to be 32 bits. Whereas, if we use a data-IDList instead of pseudo-IDList, the space requirements would be higher than or at most equal to 2 * k. The reason is that each row in a data IDList stores every position of the patterns in a user clickstream and the UCID, while the pseudo-IDList only stores the start index to the data-IDLists of frequent 1-patterns and the UCID.

DUB effectiveness. The experimental results also indicate the effectiveness of our proposed pruning heuristic. CUP+DUB runs even faster than CUP as DUB reduces the workload by reducing the number of candidates generated. The details of candidate reduction and runtime improvement are provided in **Table 3**. The numbers of candidates generated are roughly cut down by 40% to 50% on three databases, FIFA, Kosarak, and MSNBC, depending on the minimum threshold values. The highest reduction is on BMS2, where it goes up to 70%. The runtime reduction using DUB varies, but it tends to increase as the minimum support threshold approaches lower values and becomes stable after certain support thresholds.

Algorithm 2. The CUP algorithm

Input: a clickstream database *CDB* and a minimum support threshold γ
Output: a set of frequent clickstream patterns in *CDB*

1: Identify all frequent 1-patterns by scanning the whole original user clickstream database. The frequent 1-patterns are put into the set *F*. Meanwhile, data IDLists and pseudo-IDLists for all frequent 1-patterns are also created in the scanning process. // Section 4.1 and 4.2
2: *sub_F* ← DFS_Traversing(*F*, γ) // We start traveling the tree by going through frequent 1- patterns (level 1) and one by one expanding them further. *Sub_F* will hold all the frequent *k*-patterns (*k* > 1) as the result set from the function.
3: *F* ← *F* ∪ *sub_F* // We have the complete set of frequent patterns in the clickstream database
4: **return** *F*

Function DFS_Traversing

Input: a set *F* of frequent *k*-pattern that share (*k*-1)-prefix and a minimum support eshold γ
Output: a subset *sub_F* of frequent clickstream patterns in *CDB*

5: *sub_F* ← ∅ // Will later store all frequent *k*-patterns that are super clickstreams of those patterns in *F*
6: *cand_set_list* ← ∅ // A set, in which each element is a set of candidates that share the same (*k*-1)-prefix
7: Sort *F* according to a given lexicographical order
8: **for** each frequent pattern *X* in *F* **do**
/* We slowly build an entire set of candidates that share *k*-prefix *X* */
9: **for** each frequent pattern *Y* > *X* in *F* **do** // *Y* > *X* regarding the lexicographical order
10: *cand_set* ← a set of candidate patterns generated using *X* and *Y* // Section 4.4
11: Apply the DUB heuristic to *cand_set* to prune candidates // Section 4.3
12: **for** each candidate pattern *Z* in *cand_set* **do**
13: Create pseudo-IDList for *Z* from pseudo-IDLists of *X* and *Y* // Section 4.2
14: **if** *supcount*(*Z*) < γ **then** // The candidate *Z* is infrequent
15: Continue;
16: *sub_F* ← *sub_F* ∪ *Z* // The candidate *Z* is frequent and is added into the result set
/* We put each candidate to the correct child candidate group based on whether *X* or *Y* will be the *k*-prefix of the new frequent candidate. If it is *X*, then the new candidate is put into the group that has *X* being their *k*-prefix. Otherwise, it is *Y*. */
17: **if** *X* is the *k*-prefix of *Z* **then**
18: *index_set*$_X$ ← index of a candidate group that share *k*-prefix *X*
19: *cand_set_list*[*index_set*$_X$] ← *cand_set_list*[*index_set*$_X$] ∪ *Z*
20: **else**
21: *index_set*$_Y$ ← index of a candidate group that share *k*-prefix *Y*
22: *cand_set_list*[*index_set*$_Y$] ← *cand_set_list*[*index_set*$_Y$] ∪ *Z*
23: *temp_F* ← DFS_Traversing(*cand_set_list*[*index_set*$_X$], γ) // We recursively traverse the children of *X*. The frequent (*k*+1)-patterns from those children will be put in *temp_F*.
24: *sub_F* ← *sub_F* ∪ *temp_F* // The result is merged into one
25: **return** *sub_F*

Specifically, it reduces from a minimum of 13% on Kosarak (at the highest value of minimum support threshold for the database) to a maximum of 39% on FIFA (at the lowest value of minimum support threshold for the database). The memory consumption does not really increase when adding DUB into CUP. On FIFA, it even reduces memory consumption. The reason is that even DUB requires a little extra memory allocation for bitmaps to execute DUB. The extra required memory is marginal, and it also gets smaller as the algorithm progresses due to the characteristics of DUB and shrinkable size of bitmaps. Meanwhile, DUB also discards many pseudo-IDLists of infrequent candidates from being created, and thus the memory saved from the discarded pseudo-IDLists makes up for the memory used for bitmap allocation.

Scalability. To date, there is no synthetic clickstream database generator for our problem. Thus, in order to study the algorithms' scalability (i.e. the way the algorithms react to changes in database size), we reuse two large databases from previous experiments, which are MSNBC and Kosarak. Specifically, we start with subsets of MSNBC and Kosarak with the size of 200,000 user clickstreams, then we increase the subset size to 400,000 and so on. During this process, we fix the minimum support threshold at 0.09% for Kosarak and 0.02% for MSNBC. The scalability experiments are presented in Figs. 10 and 11. The results show that our proposed algorithms together with CM-SPADE and PrefixSpan have steeper linear growth rates in runtimes when databases are slightly small (from a database size of 200,000 to 600,000). After that, the runtimes' growth rates get smaller. In terms of memory consumption, on Kosarak, our proposed algorithms have smaller and smoother growth rates, while PrefixSpan and CM-SPADE's growth rates are steeper at early changes of database sizes. On MSNBC, the memory consumptions of all algorithms have a small growth rate.

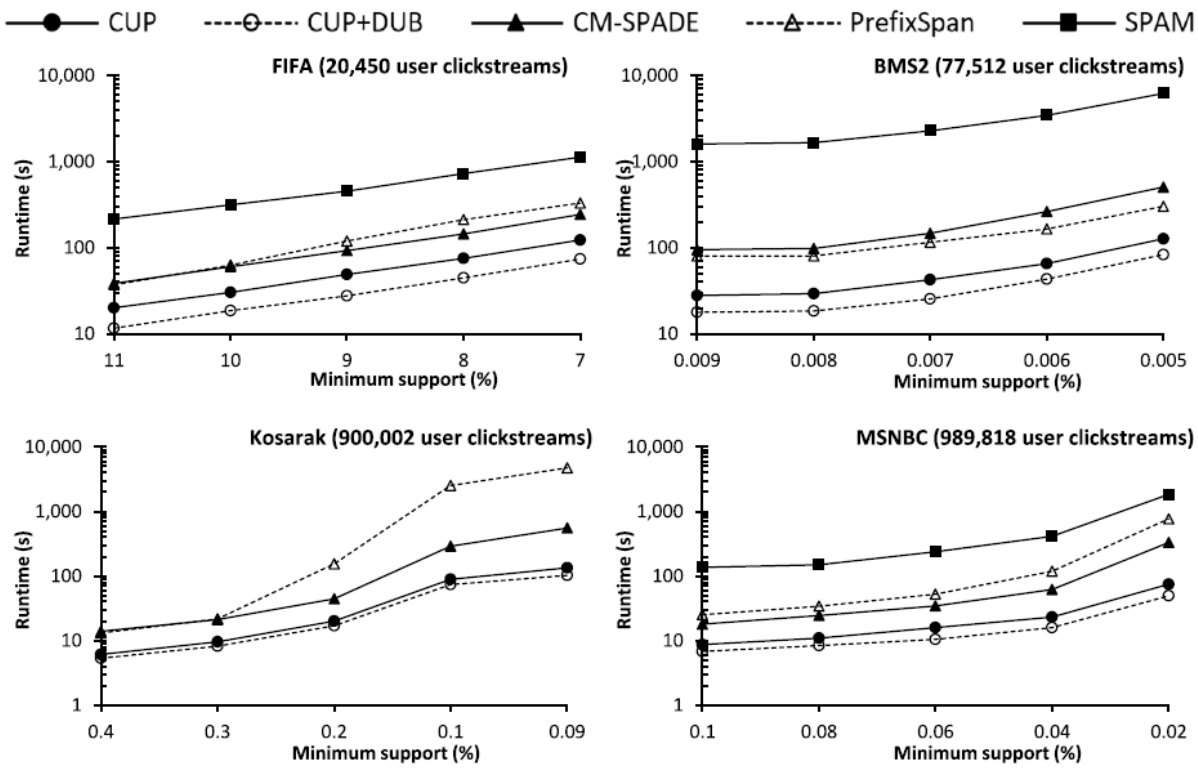

Fig. 8. Runtime for four databases.

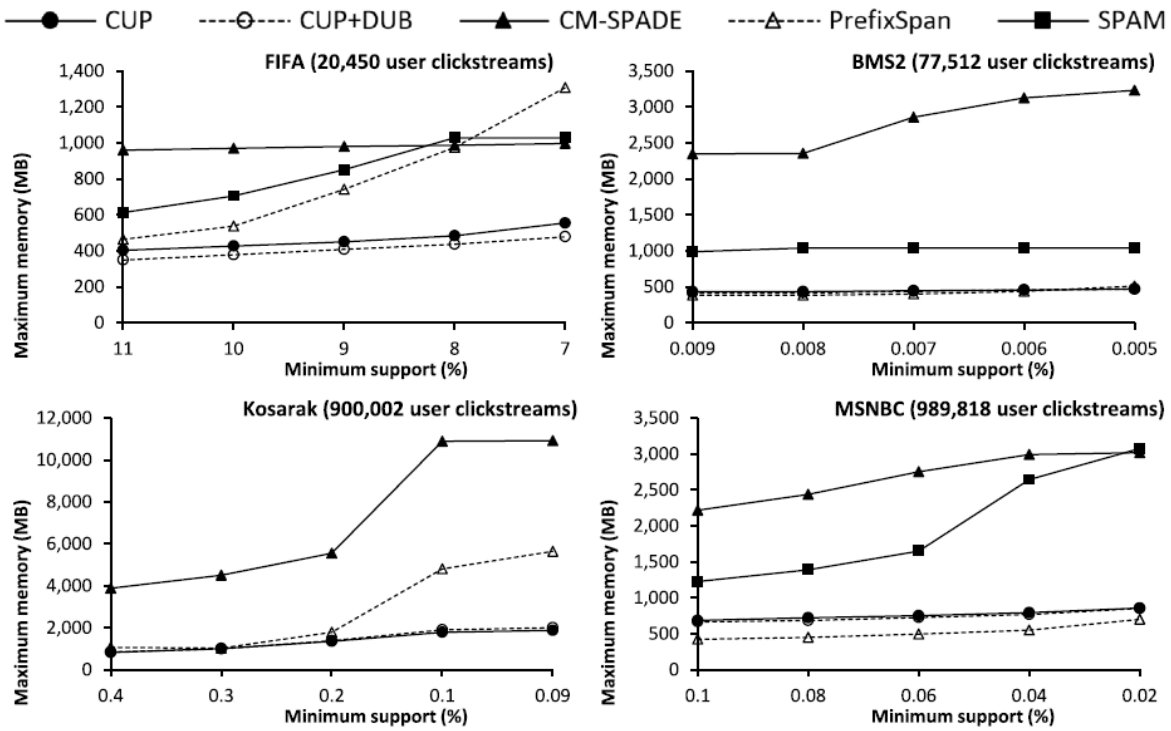

Fig. 9. Memory consumption for four databases.

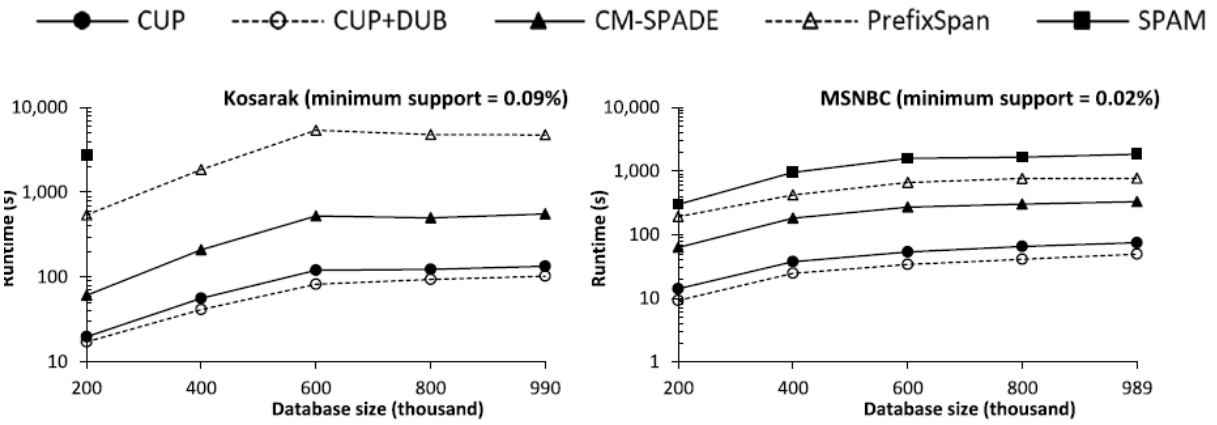

Fig. 10. Runtime when the size of the database changes.

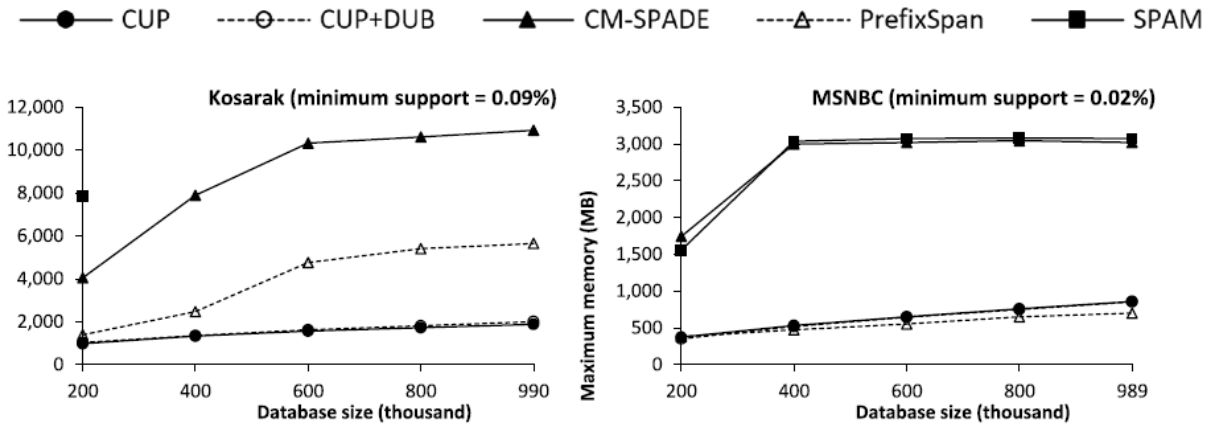

Fig. 11. Memory consumption when the size of the databases changes.

6. Conclusions and future work

Clickstream pattern mining has various potential applications, but there has been a lack of studies focusing exclusively on this issue. In this paper, we proposed an algorithm called CUP which uses pseudo-IDLists that exploit some properties of clickstreams to tackle this problem. Additionally, we also proposed a pruning heuristic named DUB to improve the performance of the CUP algorithm. Our experiments prove that CUP is generally faster than PrefixSpan and CM-SPADE, and is memory effective on large databases as it does not need to copy redundant data over to pseudo-IDLists of generated candidates. Furthermore, DUB improves the runtime further from 13 to 45% depending on the databases and minimum support thresholds used. In future work, we plan to parallelize CUP(+DUB) and adapt them to general sequential patterns as well as quantitative and incremental databases. We will also expand our work for mining closed sequential patterns and maximal sequential patterns.

References

[1] R. Agrawal, R. Srikant, Mining sequential patterns, in: Proceedings of the International Conference on Data Engineering, ICDE, 1995, pp. 3-14.

[2] T. Van, B. Vo, B. Le, Mining sequential patterns with itemset constraints, Knowl. Inf. Syst. 57 (2) (2018) 311-330.

[3] B. Huynh, C. Trinh, H. Huynh, T.T. Van, B. Vo, V. Snasel, An efficient approach for mining sequential patterns using multiple threads on very large databases, Eng. Appl. Artif. Intell. 74 (2018) 242-251.

[4] B. Le, H. Duong, T. Truong, P. Fournier-Viger, FCloSM, FGenSM: Two efficient algorithms for mining frequent closed and generator sequences using the local pruning strategy, Knowl. Inf. Syst. 53 (1) (2017) 71-107.

[5] P. Fournier-Viger, Z. Li, J.C.W. Lin, R.U. Kiran, F. Hamido, Efficient algorithms to identify periodic patterns in multiple sequences, Inf. Sci. 489 (2019) 205-226.

[6] J. Fowkes, C. Sutton, A subsequence interleaving model for sequential pattern mining, in: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 835-844.

[7] G. Lee, U. Yun, A new efficient approach for mining uncertain frequent patterns using minimum data structure without false positives, Future Gener. Comput. Syst. 68 (2017) 89-110.

[8] X. Ao, P. Luo, J. Wang, F. Zhuang, Q. He, Mining precise-positioning episode rules from event sequences, IEEE Trans. Knowl. Data Eng. 30 (3) (2018) 530-543.

[9] C. Boris, F. Len, G. Bart, Efficiently mining cohesion-based patterns and rules in event sequences, Data Min. Knowl. Discov. (2019) 1-58.

[10] U. Yun, D. Kim, Mining of high average-utility itemsets using novel list structure and pruning strategy, Future Gener. Comput. Syst. 68 (2017) 346-360.

[11] S.C. Hsueh, M.Y. Lin, C.L. Chen, Mining negative sequential patterns for e- commerce recommendations, in: Proceedings of the IEEE Asia-Pacific Services Computing Conference, APSCC, 2008, pp. 1213-1218.

[12] H.Q. Nguyen, T.T. Pham, V. Vo, B. Vo, T.T. Quan, The predictive modeling for learning student results based on sequential rules, Int. J. Innov. Comput. Inf. Control 14 (6) (2018) 2129-2140.

[13] J.K. Tarus, Z. Niu, A. Yousif, A hybrid knowledge-based recommender system for e- learning based on ontology and sequential pattern mining, Future Gener. Comput. Syst. 72 (2017) 37-48.

[14] R. Cooley, B. Mobasher, J. Srivastava, Web mining: information and pattern discovery on the World Wide Web, in: Proceedings of the IEEE International Conference on Tools with Artificial Intelligence, 1997, no. June 2014, pp. 558-567.

[15] A. Demiriz, webSPADE:a parallel sequence mining algorithm to analyze web log data, in: Proceedings of the International Conference on Data Mining, 2002, pp. 755-758.

[16] I.H. Ting, C. Kimble, D. Kudenko, UBB mining: Finding unexpected browsing behaviour in clickstream data to improve a web site's design, in: Proceedings of ACM International Conference on Web Intelligence, 2005, pp. 179-185.

[17] B. Dalmas, P. Fournier-Viger, S. Norre, TWINCLE: A constrained sequential rule mining algorithm for event logs, Procedia Comput. Sci. 112 (2017) 205-214.

[18] Y.W.T. Pramono, Anomaly-based intrusion detection and prevention system on website usage using rule-growth sequential pattern analysis: Case study: Statistics of Indonesia (BPS) website, in: Proceedings of the International Conference on Advanced Informatics: Concept, Theory and Application, 2015, pp. 203-208.

[19] T. Van, A. Yoshitaka, B. Le, Mining web access patterns with super-pattern constraint, Appl. Intell. 48 (11) (2018) 3902-3914.

[20] H.M. Huynh, L.T.T. Nguyen, B. Vo, A. Nguyen, V.S. Tseng, Efficient methods for mining weighted clickstream patterns, Expert Syst. Appl. 142 (2019) 112993.

[21] R. Srikant, R. Agrawal, Mining sequential patterns: Generalizations and performance improvements, in: Proceedings of the International Conference on Extending Database Technology, 1996, pp. 1-17.

[22] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, FreeSpan: Frequent pattern-projected sequential pattern mining, in: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2000, pp. 355-359.

[23] J. Pei, et al., PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth, in: Proceedings of the International Conference on Data Engineering, ICDE, 2001, pp. 215-224.

[24] M.J. Zaki, SPADE: An efficient algorithm for mining frequent sequences, Mach. Learn. 42 (1-2) (2001) 31-60.

[25] J. Ayres, J. Flannick, J. Gehrke, T. Yiu, Sequential pattern mining using a bitmap representation, in: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2002, pp. 429-435.

[26] P. Fournier-Viger, A. Gomariz, M. Campos, R. Thomas, Fast vertical mining of sequential patterns using co-occurrence information, in: Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining, 2014, pp. 40-52.

[27] R. Agrawal, T. Imieliński, A. Swami, Mining association rules between sets of items in large databases, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1993, pp. 207-216.

[28] R. Kessl, Probabilistic static load-balancing of parallel mining of frequent sequences, IEEE Trans. Knowl. Data Eng. 28 (5) (2016) 1299-1311.

[29] K. Gouda, M. Hassaan, M.J. Zaki, Prism: An effective approach for frequent sequence mining via prime-block encoding, J. Comput. System Sci. 76 (1) (2010) 88-102.

[30] U. Yun, K.H. Ryu, Discovering important sequential patterns with length-decreasing weighted support constraints, Int. J. Inf. Technol. Decis. Mak. 9 (4) (2010) 575-599.

[31] P. Fournier-Viger, C.W. Wu, V.S. Tseng, L. Cao, R. Nkambou, Mining partially- ordered sequential rules common to multiple sequences, IEEE Trans. Knowl. Data Eng. 27 (8) (2015) 2203-2216.

[32] T. Le, A. Nguyen, B. Huynh, B. Vo, W. Pedrycz, Mining constrained intersequence patterns?: a novel approach to cope with item constraints, Appl. Intell. 48 (5) (2018) 1327-1343.

[33] P. Fournier-Viger, C.W. Wu, A. Gomariz, V.S. Tseng, VMSP: Efficient vertical mining of maximal sequential patterns, in: Proceedings of the Canadian Conference on Artificial Intelligence, 2014, pp. 83-94.

[34] U. Yun, H. Nam, G. Lee, E. Yoon, Efficient approach for incremental high utility pattern mining with indexed list structure, Future Gener. Comput. Syst. 95 (2019) 221-239.

[35] T. Kieu, B. Vo, T. Le, Z.H. Deng, B. Le, Mining top-k co-occurrence items with sequential pattern, Expert Syst. Appl. 85 (2017) 123-133.

[36] F. Petitjean, T. Li, N. Tatti, G.I. Webb, Skopus: Mining top-k sequential patterns under leverage, Data Min. Knowl. Discov. 30 (5) (2016) 1086-1111.

[37] M.T. Tran, B. Le, B. Vo, Combination of dynamic bit vectors and transaction information for mining frequent closed sequences efficiently, Eng. Appl. Artif. Intell. 38 (2015) 183-189.

[38] F. Fumarola, P.F. Lanotte, M. Ceci, D. Malerba, CloFAST: Closed sequential pattern mining using sparse and vertical id-lists, Knowl. Inf. Syst. 48 (2) (2016) 429-463.

[39] P. Fournier-Viger, A. Gomariz, M. Šebek, M. Hlosta, VGEN: Fast vertical mining of sequential generator patterns, in: Proceedings of the International Conference on Data Warehousing and Knowledge Discovery, 2014, pp. 476-488.

[40] S. Yi, T. Zhao, Y. Zhang, S. Ma, Z. Che, An effective algorithm for mining sequential generators, Procedia Eng. 15 (2011) 3653-3657.

[41] J. Vilo, Pattern Discovery from Biosequences, 2002.

[42] L. De Raedt, M. Jaeger, Sau Dan Lee, H. Mannila, A theory of inductive query answering, in: Proceedings of the International Conference on Data Mining, 2002, pp. 123-130.

[43] S.D. Lee, L. De Raedt, An efficient algorithm for mining string databases under constraints, in: Proceedings of the International Workshop on Knowledge Discovery in Inductive Databases, 2005, pp. 108-129.

[44] J. Fischer, V. Heun, S. Kramer, Fast frequent string mining using suffix arrays, in: Proceedings of the International Conference on Data Mining, 2005, pp. 609-612.

[45] J. Fischer, V. Mákinen, N. Välimáki, Space efficient string mining under frequency constraints, in: Proceedings of the International Conference on Data Mining, 2008, pp. 193-202.

[46] Y. Li, J. Bailey, L. Kulik, J. Pei, Efficient matching of substrings in uncertain sequences, in: Proceedings of the SIAM International Conference on Data Mining, 2014, pp. 767-775.

[47] X. Ji, J. Bailey, An efficient technique for mining approximately frequent substring patterns, in: Proceedings of the International Conference on Data Mining Workshops, 2007, pp. 325-330.

[48] K. Berberich, S. Bedathur, Computing n-gram statistics in MapReduce, in: Proceedings of the International Conference on Extending Database Technology, 2013, pp. 101-112.

[49] P.A. Utama, B. Distiawan, Spark-gram: Mining frequent n-grams using parallel processing in Spark, in: Proceedings of the International Conference on Advanced Computer Science and Information Systems, 2015, pp. 129-136.

[50] M. van Gompel, A. van den Bosch, Efficient n-gram skipgram and flexgram modelling with Colibri core, J. Open Res. Softw. 4 (2016).

[51] F. Setiawan, B.N. Yahya, Improved behavior model based on sequential rule mining, Appl. Soft Comput. J. 68 (2018) 944-960.

[52] P.-M. Law, Z. Liu, S. Malik, R.C. Basole, MAQUI: Interweaving queries and pattern mining for recursive event sequence exploration, IEEE Trans. Vis. Comput. Graphics 25 (1) (2019) 396-406.

[53] T. Ledieu, G. Bouzillé, E. Polard, C. Plaisant, F. Thiessard, M. Cuggia, Clinical data analytics with time-related graphical user interfaces: Application to pharmacovigilance, Front. Pharmacol. 9 (2018).

[54] D. Gotz, F. Wang, A. Perer, A methodology for interactive mining and visual analysis of clinical event patterns using electronic health record data, J. Biomed. Inform. 48 (2014) 148-159.

[55] W. Lee, S.J. Stolfo, Data mining approaches for intrusion detection data mining approaches for intrusion detection, in: Proceedings of the Conference on USENIX Security, 1998.

[56] P. Fournier-Viger, et al., The SPMF open-source data mining library version 2, in: Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases, 2016, pp. 36-40.