A Taxonomy of Sequential Pattern Mining Algorithms

NIZAR R. MABROUKEH and C. I. EZEIFE

University of Windsor

Owing to important applications such as mining web page traversal sequences, many algorithms have been introduced in the area of sequential pattern mining over the last decade, most of which have also been modified to support concise representations like closed, maximal, incremental or hierarchical sequences. This article presents a taxonomy of sequential pattern-mining techniques in the literature with web usage mining as an application. This article investigates these algorithms by introducing a taxonomy for classifying sequential pattern-mining algorithms based on important key features supported by the techniques. This classification aims at enhancing understanding of sequential pattern-mining problems, current status of provided solutions, and direction of research in this area. This article also attempts to provide a comparative performance analysis of many of the key techniques and discusses theoretical aspects of the categories in the taxonomy.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Database Applications—*Data Mining*; J.1 [**Computer Applications**]: Administrative Data Processing—*Marketing*

General Terms: Algorithms, Management, Performance.

Additional Key Words and Phrases: Data mining, Web usage mining, sequential patterns, frequent patterns, web log, pattern growth, apriori property, prediction, early pruning, sequence mining, lattice theory, lexicographic order, tree projection, recommender systems, association rules

ACM Reference Format:

Mabroukeh, N. R. and Ezeife, C. I. 2010. A taxonomy of sequential pattern mining algorithms. ACM Comput. Surv. 43, 1, Article 3 (November 2010), 41 pages.
DOI = 10.1145/1824795.1824798 http://doi.acm.org/10.1145/1824795.1824798

1. INTRODUCTION

Sequential pattern mining discovers frequent subsequences as patterns in a sequence database. A sequence database stores a number of records, where all records are sequences of ordered events, with or without concrete notions of time. An example sequence database is retail customer transactions or purchase sequences in a grocery store showing, for each customer, the collection of store items they purchased every week for one month. These sequences of customer purchases can be represented

This research was supported by the Natural Science and Engineering Research Council (NSERC) of Canada under an operating grant (OGP-0194134) and a University of Windsor grant.

Authors' address: N. R. Mabroukeh and C. I. Ezeife, School of Computer Science, University of Windsor, ON N9B 3P4, Canada; email: {mabrouk, cezeife}@uwindsor.ca.

DOI 10.1145/1824795.1824798 http://doi.acm.org/10.1145/1824795.1824798

as records with a schema [Transaction/Customer ID, <Ordered Sequence Events>], where each sequence event is a set of store items like bread, sugar, tea, milk, and so on. For only two such customers, an example purchase sequential database is [T1, <(bread, milk), (bread, milk, sugar), (milk), (tea, sugar)>]; [T2, <(bread), (sugar, tea)>]. While the first customer, with a transaction ID shown as T1 in the example, made a purchase each of the four weeks, the second customer represented by T2 made purchases during only two of the weeks. Also, a customer can purchase one or more items during each market visit. Thus, records in a sequence database can have different lengths, and each event in a sequence can have one or more items in its set. Other examples of sequences are DNA sequences and web log data. Sequential pattern mining is an important problem with broad applications, including the analysis of customer purchase behavior, web access patterns, scientific experiments, disease treatment, natural disasters, and protein formations. A sequential pattern-mining algorithm mines the sequence database looking for repeating patterns (known as *frequent sequences*) that can be used later by end users or management to find associations between the different items or events in their data for purposes such as marketing campaigns, business reorganization, prediction and planning. With the increase in the use of the world wide web for e-commerce businesses, web services, and others, web usage mining is one of the most prevalent application areas of sequential pattern mining in the literature [Pei et al. 2000; Lu and Ezeife 2003; El-Sayed et al. 2004; Wang and Han 2004; Goethals 2005]. Typical applications of web usage mining fall into the area of user modeling, such as web content personalization, web site reorganization, prefetching and caching, e-commerce, and business intelligence [Facca and Lanzi 2005].

This article focuses on sequential pattern-mining techniques including those applicable to web usage mining. An introduction to web usage mining as an application of sequential pattern mining is presented in Section 1.1, and contributions and an outline of work appears in section 1.2.

1.1. Web Usage Mining as a Sequential Pattern-Mining Application

Web usage mining (also called *web log mining*) is an important application of sequential pattern mining concerned with finding user navigational patterns on the world wide web by extracting knowledge from web logs, where ordered sequences of events in the sequence database are composed of single items and not sets of items, with the assumption that a web user can physically access only one web page at any given point in time. If a time window for access is considered that may allow a web user to browse a collection of web pages over a specified period of time, it then reverts back to a general sequence database. Currently, most web usage-mining solutions consider web access by a user as one page at a time, giving rise to special sequence database with only one item in each sequence's ordered event list. Thus, given a set of events $E = \{a, b, c, d, e, f\}$, which may represent product web pages accessed by users in an e-commerce application, a web access sequence database for four users may have four records: [T1, <*abdac*>]; [T2, <*eaebcac*>]; [T3, <*babfaec*>]; [T4, <*abfac*>]. A web log pattern mining on this web sequence database can find a frequent sequence, *abac,* indicating that over 90% of users who visit product a's web page of http://www.company.com/producta.htm also immediately visit product b's web page of http://www.compay.com/productb.htm and then revisit product a's page, before visiting product c's page. Store managers may then place promotional prices on product a's web page, which is visited a number of times in sequence, to increase the sale of other products. The web log could be on the server-side, client-side, or on a proxy server, each with its own benefits and drawbacks in finding the users' relevant patterns and navigational sessions [Iváncsy and Vajk 2006]. While web log data recorded on the server side reflects the access of a web site by multiple users, and is good for mining multiple users' behavior and web recommender systems, server logs may not be entirely reliable due to caching, as cached page views are not recorded in a server log. Client-side data collection requires that a remote agent be implemented or a modified browser be used to collect single-user data, thus eliminating caching and session identification problems, and is useful for web content personalization applications. A proxy server can, on the other hand, reveal the actual HTTP requests from multiple clients to multiple web servers, thus characterizing the browsing behavior of a group of anonymous users sharing a common server [Srivastava et al. 2000]. Web user navigational patterns can be mined using sequential pattern mining of the preprocessed web log. Web usage mining works on data generated by observing web surf sessions or behaviors, which are stored in the web log from which all user behaviors on each web server can be extracted. An example of a line of data in a web log is: 137.207.76.120 - [30/Aug/2009:12:03:24 -0500] “GET /jdk1.3/docs/relnotes/deprecatedlist.html HTTP/1.0” 200 2781. This recorded information is in the format: host/ip user [date:time] ”request url” status bytes; it represents from left to right, the host IP address of the computer accessing the web page (137.207.76.120), the user identification number (-)(this dash stands for anonymous), the time of access (12:03:24 p.m. on Aug 30, 2009 at a location 5 hours behind Greenwich Mean Time (GMT)), request (GET/jdk1.3/docs/relnotes/deprecatedlist.html)(other types of request are POST and HEAD for http header information), the unified reference locator (url) of the web page being accessed (HTTP/1.0), status of the request (which can be either 200 series for success, 300 series for redirect, 400 series for failures and 500 series for server error), the number of bytes of data being requested (2781). The cached page views cannot be recorded in a server log, and the information passed through the POST method may not be available in server log. Thus, in order to gain a proper web usage data source, some extra techniques, such as packet sniffer and cookies, may be needed during web log preprocessing on the server side. In most cases, researchers assume that user web visit information is completely recorded in the web server log, which is preprocessed to obtain the transaction database to be mined for sequences. Several sequential pattern-mining techniques that can be applied to web usage mining have been introduced in the literature since mid 1990s. Previous surveys looked at different mining methods applicable to web logs [Srivastava et al. 2000; Facca and Lanzi 2003; Masseglia et al. 2005; Facca and Lanzi 2005; Iváncsy and Vajk 2006], but they lack three important things: they fail (i) to focus on sequential patterns as a complete solution; (ii) provide a taxonomy; and (iii) include a deep investigation of the techniques and theories used in mining sequential patterns.

1.2. Contributions and Outline

This article is directed towards the general case of sequential pattern mining, and does not look into algorithms specific to closed, maximal or incremental sequences, neither does it investigate special cases of constrained, approximate or near-match sequential pattern mining. Although all these special types of sequential mining techniques have not been classified in a taxonomy either, we keep our taxonomy neat and focused by excluding them. This taxonomy (the term *taxonomy* is used to mean classification of sequential pattern-mining techniques and algorithms in the literature) provides a deep discussion of features in each category of algorithms (i.e., taxonomy species), highlighting weaknesses and techniques that require further research. The goal of such a taxonomy is to answer questions like:

(1) *What are the important features that a reliable sequential pattern-mining algorithm should provide?* A reliable algorithm should have acceptable performance measures such as low CPU execution time and low memory utilization when mined with low minimum support values, and should be scalable.

(2) *What are the needs of sequential pattern-mining applications like web content and web log mining?* The algorithms should be able to handle varying types of data like low-quality data with noise, multielement-set event sequences and single-element-set event sequences, and sparse databases. The techniques should also be extendible to support mining sequence data in other types of domains such as distributed, constrained, stream, and object-oriented domains.

(3) *What are the different approaches used so far in general sequential pattern mining and which techniques are suitable for what domains*? The main techniques can be categorized into apriori-based, pattern-growth, early-pruning, and hybrids of these three techniques. The apriori-based algorithms are deemed too slow and have a large search space, while pattern-growth algorithms have been tested extensively on mining the web log and found to be fast, early-pruning algorithms have had success with protein sequences stored in dense databases.

(4) *What else can be contributed to sequential pattern mining*? Scalability and handling very long sequences is still a problem not addressed well by many existing techniques. Solutions for distributed and object-oriented domains remain open issues. The ability to mine periodic and time-sensitive frequent patterns, which enables a recommender/prediction system to be context- and time-aware, for example, a group of users may show frequent patterns in an e-commerce web site on holidays and special occasions (like shopping for Christmas) that are different from their regular frequent patterns. The recommender in a web usage-mining system should be able to tell which frequent patterns to mine and utilize based on different contexts. Also, the use of domain knowledge and inclusion of semantic information into the mining process itself requires further attention, as discussed in Section 5.

The proposed taxonomy contributes to research because

(1) It helps in-depth understanding of the different algorithms and their component features, and also techniques and methods used in research so far.

(2) It highlights the comparative advantages and drawbacks of the algorithms.

(3) It clearly shows research trends in time for each category of algorithms.

(4) It is the first taxonomy in the area of sequential pattern-mining research that presents a hierarchical, a tabular, and a chronological ordering of the algorithms along with their features.

(5) It also presents experimental performance features that include comparative analysis of CPU execution time and memory usage.

The rest of this article is organized as follows. Section 2 presents the formal definition of the problem of sequential pattern mining. Section 3 gives the proposed taxonomy with comparative experimental performance analysis of the reviewed algorithms, along with discussions of the features used in the classification of the algorithms. Section 4 presents surveys of sequential pattern-mining algorithms with examples. Algorithms belong to three main classification categories: apriori-based, pattern-growth and early-pruning, and some algorithms are hybrids of these techniques. Conclusions and future work are given in Section 5.

2. PROBLEM DEFINITION

This section presents the formal definition of the problem of sequential pattern mining and its application to mining the web log. Given (i) a set of sequential records (called *sequences*) representing a sequential database D; (ii) a minimum support threshold called *min_sup* ξ; and (iii) a set of k unique items or events $I = \{i_1, i_2, \ldots, i_k\}$. The problem of mining sequential patterns is that of finding the set of all frequent sequences S in the given sequence database D of items I at the given *min_sup* ξ.

For example, in a web usage-mining domain, the items in I can represent the list of web pages (e.g., pages *a, b, c, d, e, f*) or list of products (e.g., TV, radio) being sold at an e-commerce web site. An itemset is a nonempty, unordered collection of items (e.g., *(abe)* which are accessed at the same time). A sequence is a lexicographically ordered list of itemsets, for example, $S = < a(be)c(ad)>$. *Set Lexicographic Order* [Rymon 1992] is a total linear order, which can be defined as follows. Assume an itemset t of distinct items, $t = \{i_1, i_2, \ldots, i_k\}$, and another itemset of distinct items also $t' = \{j_1, j_2, \ldots, j_l\}$, where $i_1 \leq i_2 \leq \cdots \leq \imath_k$ and $j_1 \leq j_2 \leq \cdots \leq j_l$, such that $\leq$ indicates "occurs before" relationship. Then, for itemsets, $t < t'$ (t is lexicographically less than t') iff either of the following is true:

(1) for some integer h, $0 \leq h \leq min\{k, l\}$, we have $i_r = j_r$ for $r < h$, and $i_h < j_h$, or
(2) $k < l$, and $i_1 = j_1, i_2 = j_2, \ldots, i_k = j_k$.

An example of the first case is $(abc) < (abec)$ and $(af) < (bf)$; the second case is similar to a strict subset relationship, where t is a strict subset of t', for example, $(ab) < (abc)$.

An itemset is a set drawn from items in I, and denoted $(i_1, i_2, \ldots, i_k)$, where i_j is an item or event. A sequence S is denoted as a sequence of items or events $<e_1e_2e_3 \ldots e_q>$, where the sequence element e_j is an itemset (e.g., (be) in $<a(be)c(ad)>$) that might contain only one item (which is also referred to as 1-itemset). A sequence with k items is called a k-sequence. An item can occur only once in an itemset, but it can occur several times in different itemsets of a sequence. A sequence $\alpha = < e_{i_1}e_{i_2}e_{i_3}, \ldots, e_{i_m} >$ is a subsequence of another, sequence $\beta = < e_1e_2e_3 \ldots e_n >$, denoted $\alpha \preccurlyeq \beta$, if there exists integers $i_1 < i_2 < \ldots < i_m$ and all events $e_{i_j} \in \alpha$ and $e_i \in \beta$ and $i_1 \leq 1$ and $i_m \leq n$, such that $e_{i_j} \subseteq e_i$. A sequential pattern is *maximal* if it is not a subsequence of any other sequential pattern. Sequence lexicographical ordering can be defined as follows. Assume a lexicographical order $\leq$ of items I in the sequential access database, denoted $\leq_{\mathrm{I}}$. If an item i occurs before an item j, it is denoted $i \leq_{\mathrm{I}} j$; this order is also extended to sequences and subsequences by defining $S_a \leq S_b$ if S_a is a subsequence of S_b. Consider all sequences arranged in a sequence tree T (referred to as a Lexicographical Tree), as in Figure 1, as follows: The root of the tree is labeled $\{\}$. Recursively, if n is a node in the tree T, then n's children are all nodes n' such that $n \leq n'$ and $\forall m \in T : n' \leq m \Rightarrow n \leq m$, each sequence in the tree can be extended by adding a 1-sequence to its end or adding an itemset to its end. In the former case it is called a *sequence-extended sequence* and in the later case it is called an *itemset-extended sequence*, which is not applicable to the case of web log mining.

The frequency or support of a sequence (or subsequence) S, denoted $\sigma(S)$ is the total number of sequences of which S is a subsequence divided by the total number of sequences in the database D, whereas the *absolute support* (or *support count*) of a sequence (or subsequence) S is the total number of sequences in D of which S is a subsequence. A sequence is called *frequent* if its frequency is not less than a user-specified threshold, called minimum support, denoted *min_sup* or the greek letter ξ. A frequent sequenc S_α is called a frequent *closed sequence* if there exists no proper supersequence of S_α with the same support, that is, $\nexists\, S_\beta$ such that $S_\alpha \preccurlyeq S_\beta$ and $\sigma(S_\alpha) = \sigma(S_\beta)$; otherwise it is said that sequence S_α is absorbed by S_β [Wang and Han 2004]. For example,

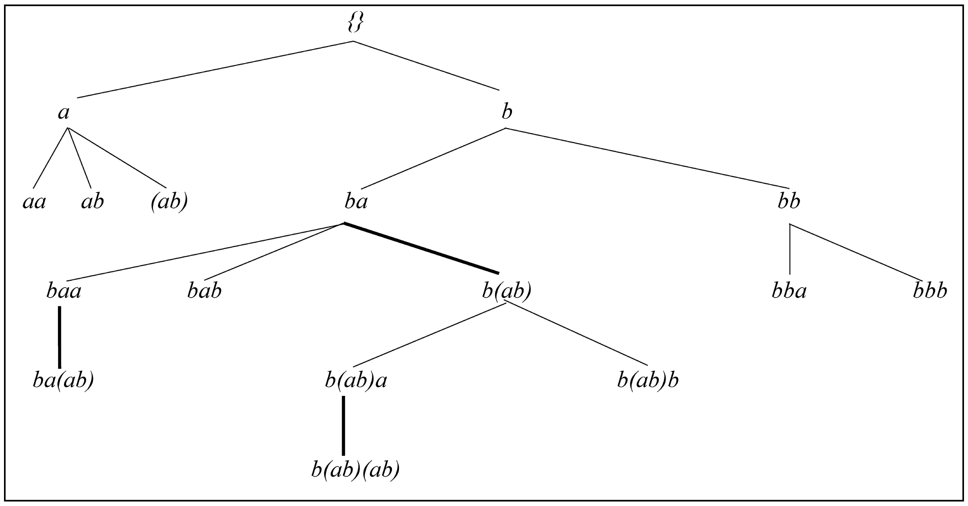


Fig. 1. Lexicographic sequence subtree for items *a* and *b* only. Light lines mean sequence-extended sequence (S-Step), bold lines mean item-set extended sequence (I-Step).

assume the frequent sequence $S_\beta =< beadc >$ is the only superset of the frequent sequence $S_\alpha =< bea >$, if, $\sigma(S_\alpha) = \sigma(S_\beta)$, then S_α is not a frequent closed sequence; on the other hand, if $\sigma(S_\alpha) > \sigma S_\beta$, then S_α is a frequent closed sequence. Notice that $\sigma(S_\beta)$ cannot be greater than $\sigma(S_\alpha)$, because $S_\alpha \preccurlyeq S_\beta$.

In web usage mining, D is a sequence database of transactions representing web accesses, where each transaction T has a unique identifier (Transaction id (TID) or Session id (SID)) and a sequence of single-element set events as in <*bcabd*> and <*bcabdac*>. The problem of sequential pattern mining is restricted to sequential web log mining with a D, *min_sup* ξ, and set of events $E = \{a, b, c, d\ldots\}$ representing web page addresses, with the following characteristics:

(1) Patterns in a web log consist of contiguous page views (items in the sequences). No two pages can be accessed by the same user at the same time, and so sequences contain only 1-itemsets (i.e., singleton itemsets). An example sequence is <*bcabdac*>, which is different from a general sequence like <(*ba*)(*ab*)*d*(*ac*)>.
(2) Order is also important in web usage mining, as the order of page references in a transaction sequence is important. Also, each event or item can be repeated, and represents page refreshes and backward traversals (e.g., <*aba*>, <*aab*>, where event *a* is repeated in these sequences).
(3) Web logs are sparse datasets, that is, usually there are many unique items with only a few repetitions in the sequences of one user, which makes algorithms targeting sparse datasets (e.g., LAPIN_WEB [Yang et al. 2006]) perform better than those for dense datasets (e.g., PrefixSpan [Pei et al. 2001]), as claimed in Yang et al. [2006]. Our experimental results, however, show that LAPIN_Suffix [Yang et al. 2005], outperforms PrefixSpan only on large datasets and at a higher support level when many frequent patterns are not likely to occur.

3. A TAXONOMY OF SEQUENTIAL PATTERN-MINING ALGORITHMS

A taxonomy of existing sequential pattern-mining algorithms is provided in Figure 4 in this section, which lists the algorithms, showing a comparative analysis of their different important features. This proposed taxonomy is composed of three main categories of sequential pattern-mining algorithms, namely, apriori-based, pattern-growth and early-pruning algorithms. First, the thirteen features used in classifying the reviewed algorithms are discussed in Sections 3.1 to 3.3 before presentation of the taxonomy in Section 3.4. These features are a result of careful investigation of the surveyed algorithms and represent a superset of features usually discussed in the literature. Some features are combined to make the list shorter, for example, *projected databases*, *conditional search* on projected trees, and *search space partitioning* are all combined under the *search space-partitioning* feature. A review of algorithms in each category is presented with detailed examples in Section 4.

Frequent sequential pattern discovery can essentially be thought of as association rule discovery over a temporal database. While association rule discovery [Agrawal et al. 1993] covers only intratransaction patterns (itemsets), sequential pattern mining also discovers intertransaction patterns (sequences), where ordering of items and itemsets is very important, such that the presence of a set of items is followed by another item in a time-ordered set of sessions or transactions. The set of all frequent sequences is a superset of the set of frequent itemsets. Due to this similarity, the earlier sequential pattern-mining algorithms were derived from association rule mining techniques. The first of such sequential pattern-mining algorithms is the AprioriAll algorithm [Agrawal and Srikant 1995], derived from the Apriori algorithm [Agrawal et al. 1993; Agrawal and Srikant 1994]. An algorithm can fall into one or more (hybrid algorithms) of the categories in the proposed taxonomy. Algorithms mainly differ in two ways:

(1) The way in which candidate sequences are generated and stored. The main goal here is to minimize the number of candidate sequences generated so as to minimize I/O cost.
(2) The way in which support is counted and how candidate sequences are tested for frequency. The key strategy here is to eliminate any database or data structure that has to be maintained all the time for support of counting purposes only.

The data structures used to store candidate sequences have also been a research topic and an important heuristic for memory utilization. Usually, a proposed algorithm also has a proposed data structure to accompany it, such as SPADE [Zaki 1998] with vertical databases; PrefixSpan [Pei at al. 2001] with projected databases; WAP-Mine [Pei et al. 2000] with WAP-tree; and PLWAP [Lu and Ezeife 2003] with its PLWAP-tree.

3.1. Three Features in the Taxonomy for Apriori-Based Algorithms

The Apriori [Agrawal and Srikant 1994] and AprioriAll [Agrawal and Srikant 1995] set the basis for a breed of algorithms that depend largely on the apriori property and use the *Apriori-generate* join procedure to generate candidate sequences. The apriori property states that *"All nonempty subsets of a frequent itemset must also be frequent"*. It is also described as antimonotonic (or downward-closed), in that if a sequence cannot pass the minimum support test, all of its supersequences will also fail the test. Given a database of web access sequences $\mathcal{D}$ over $\mathcal{J}$ items, and two sets $X, Y \subseteq \mathcal{J}$, then $X \subseteq Y \Rightarrow support(Y) \leq support(X)$. Hence, if a sequence is infrequent, all of its supersets must be infrequent; and vice versa, if a sequence is frequent, all its subsets must be frequent too. This antimonotonicity is used for pruning candidate sequences in the search space, and is exploited further for the benefit of most pattern-growth algorithms, as discussed in Section 4.2. Algorithms that depend *mainly* on the apriori property, without taking further actions to narrow the search space have the disadvantage of maintaining the support count for each subsequence being mined and testing this property during each iteration of the algorithm, which makes them computationally expensive. To overcome this problem, algorithms have to find a way to calculate support and prune candidate sequences without counting support and maintaining the count in each iteration. Most of the solutions provided so far for reducing the computational cost resulting from the apriori property use a bitmap vertical representation of the access sequence database [Zaki 1998; Ayers et al. 2002; Yang and Kitsuregawa 2005; Song et al. 2005] and employ bitwise operations to calculate support at each iteration. The transformed vertical databases, in their turn, introduce overheads that lower the performance of the proposed algorithm, but not necessarily worse than that of pattern-growth algorithms. Chiu et al. [2004] propose the DISC-all algorithm along with the *Direct Sequence Comparison DISC* technique, to avoid support for counting by pruning nonfrequent sequences according to other sequences of the same length (see Section 4.3.3). There is still no variation of the DISC-all for web log mining. Breadth-first search, generate-and-test, and multiple scans of the database, which are discussed below, are all *key* features of apriori-based methods that pose challenging problems, hinder the performance of the algorithms.

(1) *Breadth-first search:* Apriori-based algorithms are described as breath-first (level-wise) search algorithms because they construct all *k*-sequences together in each *k*th iteration of the algorithm as they traverse the search space.

(2) *Generate-and-test:* This feature is introduced by the Apriori algorithm [Agrawal et al. 1993] and is used by the very early algorithms in sequential pattern mining. In BIDE (an algorithm for mining frequent closed sequences [Wang and Han 2004]), it is referred to as "maintenance-and-test". It entails using exhaustive join operators (e.g., *Apriori-generate, GSP-join*), in which the pattern is simply grown one item at a time and tested against the minimum support. Algorithms that depend on this feature *only* display an inefficient pruning method and generate an explosive number of candidate sequences, consuming a lot of memory in the early stages of mining. To illustrate this, given n frequent 1-sequnces and min_sup=1, the apriori-based algorithms will generate $n^2 + \binom{n}{2}$ candidate 2-seuqneces and $\binom{n}{3}$ candidate 3-sequences, and so on. Eventually, the total number of candidate sequences generated will be greater than $\Sigma_{k=1}^{n} \binom{n}{k}$. For example, if there are 1000 frequent 1-sequences, apriori-based algorithms will generate $1000 \times 1000 + \frac{1000\times999}{2} =$ 1,499,500 candidate 2-sequences and $\frac{1000\times999\times998}{6} =$ 166,167,000 candidate 3-sequences. This causes the proposed methods to utilize secondary storage, and so introduce I/O delay into performance (e.g., GSP [Srikant and Agrawal 1996], ECLAT [Zaki 2000]). Solutions proposed include: (i) *Database partitioning*: The sequence database is partitioned into several disjoint parts that can fit into main memory, each part is mined separately, and finally all frequent sequences found from all parts are merged, Savasere et al. [1995]. (ii) *Search-space partitioning*: Methods to partition the search space include vertical database layout, as introduced in SPADE [Zaki 1998], which also partitions the lattice structure into equivalence classes that can be loaded and mined separately in main memory. Further discussion of search-space partitioning is provided in Section 3.2.

The generate-and-test feature also appears in SPAM [Ayres et al. 2002], which is considered one of the efficient algorithms for mining sequences. It is used in some stages of mining, usually along with search-space partitioning and projected databases without *Apriori-generate* join. In this case, memory consumption is reduced by an order of magnitude. Database projection (discussed later) is another example of search-space partitioning, and is also an important feature of pattern-growth algorithms.

(3) *Multiple scans of the database:* This feature entails scanning the original database to ascertain whether a long list of generated candidate sequences is frequent or

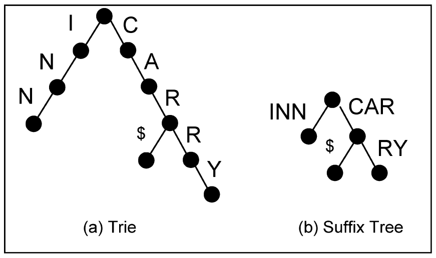


Fig. 2. A sample trie and a suffix tree resulting from its telescoping.

not. It is a very undesirable characteristic of most apriori-based algorithms and requires a lot of processing time and I/O cost. A solution to this limitation is to scan the database only once or twice to create a temporary data structure, which holds support information used during mining. PSP [Masseglia et al. 1999] uses a prefix tree to hold candidate sequences along with the support count for each sequence at the end of each branch that represents it. This method (along with GSP, from which it is adopted) becomes very inefficient when the support threshold is very low, making it a poor choice for web log mining. Apriori-GST [Tanasa 2005], on the other hand, employs a generalized suffix tree as a hash index to calculate the support count of stored subsequences. An attentive investigation of Apriori-GST makes one wonder why the author did not use the suffix tree for mining instead of just holding the support information and depending on a variation of apriori for mining. Other algorithms that do *not* depend on multiple scans of the database are mostly either pattern-growth algorithms (e.g., PrefixSpan [Pei et al. 2001]; WAP-mine [Pei et al. 2000]); or a hybrid of apriori-based and pattern-growth techniques (e.g., SPARSE [Antunes and Oliveira 2004]). These algorithms usually scan the database at least twice to create a different data structure that is loaded into main memory and used in the mining phase rather than perform candidate generate-and-test. This helps reduce I/O access costs or extra storage requirements.

3.2. Seven More Features in the Taxonomy for Pattern-Growth Algorithms

(1) *Sampling and/or compression:* Compression is mainly used in pattern-growth tree projection methods, such as FS-Miner [El-Sayed et al. 2004]; WAP-tree [Pei et al. 2000]; PLWAP [Lu and Ezeife 2003]; and PS-tree (a progressive sequential-mining algorithm [Huang et al. 2006]). In the apriori-based category, compression is used to solve the problem of fast memory consumption due to the explosive growth of candidate sequences by some apriori-based algorithms, like Apriori-GST [Tanasa 2005]. Compression is used in the data structure that holds the candidate sequences, usually a tree. Shared prefixes in sequences are represented in the tree by one branch, each node represents an item in the sequence alongside with its support count. Another compressed representation of candidate sequences is the Suffix tree, a variation of a *trie*. A trie [Dunham 2003] is a rooted tree, where root-leaf paths represent sequences. Tries are used to store strings in pattern-matching applications. Each item in the sequence (or character in the string) is stored on the edge to the node, which makes common prefixes of sequences shared, as illustrated in Figure 2(a).

Tries can grow to be very large and consume a lot of memory; as a solution, compression is introduced into a trie by using *telescoping*, resulting in a suffix tree (Figure 2(b)). This process involves collapsing any single-child node (which has no siblings) into its parent node. A problem remains on what to do with support counts stored in

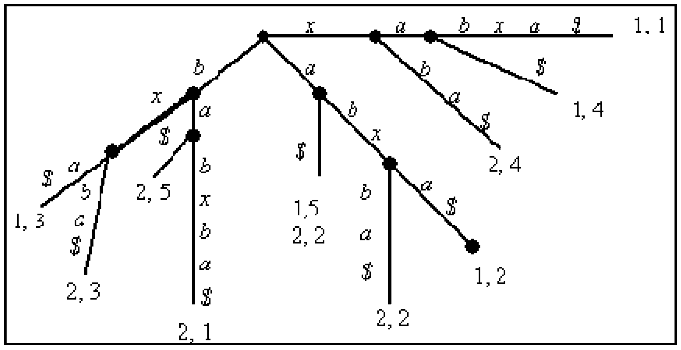


Fig. 3. General suffix tree used in Apriori-GST for the strings S1 = “xabxa” and S2 = “babxba”. (from Tanasa [2005]).

the collapsed edges if both parent and child have different counts? One solution proposed by Nandi and Jagadish [2007], working in the field of phrase prediction, is to collapse only those nodes that have similar counts. In FS-Miner, El-Sayed et al. [2004] do mention telescoping of the FS-tree, but do not provide any explanation or illustration of how it was implemented and its effect on performance. A general suffix tree is used in Apriori-GST (Figure 3), inspired by FP-tree [Han et al. 2000]. Here, items are represented by the edge, and each branch ends with an “$” representing a sequence along with its support count.

We notice that telescoping in a trie is not sufficiently addressed in sequential pattern mining, which leaves another research door open. It can be used to further compress tree structures in tree projection pattern-growth algorithms. Instead of storing one item per node, a node (or an edge) can store a complete prefix.

Sampling [Toivonen 1996], has been proposed as an improvement over AprioriAll. The problem with sampling is that the support threshold must be kept small, which causes a combinatorial explosion in the number of candidate patterns [Goethals 2005]. We think that sampling should be given more attention in pattern-growth and early-pruning algorithms as a way to reduce search space and processing time for mining. An open research area is in investigating ways to sample sequences and partition the search space based on the Fibonacci Sequence [Zill 1998] as a guide for sampling or partitioning of sequences. Another way to reduce the search space while mining sequential patterns is to focus on concise representations of sequences, such as mining maximal or closed sequences.

(2) *Candidate sequence pruning:* This feature in the early pruning of candidate sequences at a certain mining phase is unavoidable. Pattern-growth algorithms, and later early-pruning algorithms, try to utilize a data structure that allows them to prune candidate sequences early in the mining process (e.g., WAP-mine [Pei et al. 2000]; PLWAP [Lu and Ezeife 2003]; and PrefixSpan [Pei et al. 2001]), while early-pruning algorithms try to predict candidate sequences in an effort to *look-ahead* in order to avoid generating them (e.g., LAPIN [Yang et al. 2005]; DISC-all [Chiu et al. 2004]), as discussed in Section 3.3.

Pattern-growth algorithms that can prune candidate sequences early display a smaller search space and maintain a more directed and narrower search procedure. They use depth-first search along with antimonotonic application of the apriori property. Actually, none of the investigated algorithms use all of the aforementioned techniques. We see that WAP-mine, FS-Miner, and PLWAP use conditional search to constrain the mining procedure to smaller search spaces (WAP-mine claims to be a memory-only algorithm while PLWAP is not). Ezeife and Lu [2005] were able to find a way to use depth-first traversal in PLWAP during mining by linking nodes of similar frequent 1-sequence items in a queue in a preordered fashion when building the PLWAP-tree, while WAP-mine itself uses a way similar to breadth-first by linking nodes of similar frequent 1-sequence items in a queue based on first occurrence. PLWAP also has a position-coded feature that enables it to identify locations of nodes relevant to each other as a look-ahead capability and to prune candidate sequences early in the mining process. PrefixSpan, on the other hand, uses a direct antimonotonic application of the apriori property to prune candidate sequences alongside projected databases. In general, a reliable sequential pattern-mining algorithm must avoid generating candidate sequences or at least have a look-ahead feature that allows it to prune candidate sequences earlier in the mining process. We believe that the literature should mature in a way that allows it to avoid generating candidate sequences. Focus should be directed to finding mathematical models to represent sequences and mine them, and to generate formulas for predicting future ones.

(3) *Search space partitioning:* This feature is available in some apriori-based algorithms like SPADE and SPARSE [Antunes and Oliveira 2005], but it is a characteristic feature of pattern-growth algorithms. It allows partitioning of the generated search space of large candidate sequences for efficient memory management. There are different ways to partition the search space. ISM [Parthasarathy et al. 1999] and SPADE represent the search space as a *lattice* structure and use the notion of equivalence classes to partition it, although there is no physical implementation of the lattice. WAP-mine and PLWAP handle subtrees of a tree-projection structure recursively. Once the search space is partitioned, smaller partitions can be mined in parallel. Advanced techniques for search space partitioning include projected databases and conditional search, referred to as split-and-project techniques. The sequence database is divided into smaller databases projected on the frequent 1-sequences once they are mined, then those projected databases are allowed to grow on the prefix or the suffix, which in turn is used as another projection. FreeSpan [Han et al. 2000], on the other hand, uses projected databases to generate database annotations that guide the mining process to find frequent patterns faster; its projected database has a shrinking factor much less than that of PrefixSpan [Pei et al. 2001]. In PrefixSpan, projected databases are created simultaneously during one scan of the original sequence database. Lexicographic ordering is maintained and is important. Although split-and-project is a good feature to have, it consumes a lot of memory to store all the projections, especially when recursion is used. To solve this problem, Pei et al. [2001] propose the idea of *pseudo-projections*, used later by other algorithms like (LAPIN_Suffix [Yang et al. 2005] and SPARSE [Antunes and Oliveira 2004]). In this case, the projected database is not physically stored in memory, rather a pointer and an offset are used to specify locations of different projections in the sequence database in memory. Once the projected database is determined, another faster way to mine it is to use *bilevel projection*, represented in FreeSpan and PrefixSpan by the *S-Matrix* [Han et al. 2000; Pei at al. 2001]. Bilevel projection is not encouraged when the sequence database fits in main memory because it induces a lot of I/O. The S-Matrix stores support count information for projected databases in a lower triangular matrix holding information on the support count of different occurrences of the items in the mined prefix. One modification we suggest is to virtually split the S-Matrix into three different matrices that hold information for the three different occurrences of a frequent prefix. This allows the user to provide a choice of order to the algorithm, targeting web log-mining scenarios.

Tree projection algorithms simulate split-and-project by employing conditional search on the search space represented by the tree. In this case, a header linkage table is maintained at all times [Pei et al. 2000; Ezeife and Lu 2005]; it contains heads of linked lists that connect different occurrences of frequent 1-sequences (a linked list per frequent 1-sequence). The mining algorithm uses the frequent 1-sequences as projected databases, called conditional trees, and follows the links from the header linkage table through all the candidate *k*-sequences represented by tree branches, thus restricting the search space to only that where this frequent 1-sequence is a prefix/suffix; this 1-sequence is then grown with what is mined, and conditional search runs recursively over its, now, smaller search space.

(4) *Tree projection:* Tree projection usually accompanies pattern-growth algorithms. Here, algorithms implement a physical tree data structure representation of the search space, which is then traversed breadth-first or depth-first in search of frequent sequences, and pruning is based on the apriori property. WAP-mine uses the WAP-tree [Pei et al. 2000], which is generated in only two scans of the sequence database and is used instead of the database to store candidate sequences. Each node in the tree represents an item in a sequence with its support count, each branch represents a complete candidate sequence; and candidate sequences that share a path actually share a common prefix in the sequence database. FS-Miner [El-Sayed et al. 2004] uses the FS-tree, which allows the mining process to start with 2-sequences immediately from the second scan of the database—this is made possible by utilizing header linkage table of edges rather than frequent 1-sequences as in a WAP-tree. Also, any input sequence with infrequent links is pruned before being inserted into the tree using a direct application of the apriori-property. One problem we noticed with the FS-tree is that it can get very sparse fast, the reason being that while creating the tree, infrequent links within a given sequence are skipped and the sequence is split and stored in two branches of the tree instead of being compressed into one branch like the WAP-tree. The FS-tree is mined using conditional search with depth-first traversal and backward prefix extraction [El-Sayed et al. 2004]. While the FS-tree stores a lot of sequences and information in its node (frequent sequences along with *potential sequences*), the PLWAP-tree does not. It is built in a way similar to a WAP-tree, except that the header linkage table links the nodes in preorder fashion, and each node is assigned a binary position code to identify its descendants during mining and to avoid counting their support, thus skipping mining the whole infrequent suffix tree (early candidate sequence pruning). The position code stored in each node is a simple 4-byte integer, compared to the WAP-tree and FS-Miner which store much more data (intermediate WAP-trees and additional potential sequences with edge support counts, respectively).

(5) *Depth-first traversal:* We include this as a feature on its own because it is very important to have in any algorithm that uses a tree model. It has been stressed a lot and made very clear in several works [Ayres et al. 2002; Wang and Han 2004; Antunes and Oliveira 2004; Song et al. 2005; Ezeife and Lu 2005] that depth-first search of the search space makes a big difference in performance, and also helps in the early pruning of candidate sequences as well as mining of closed sequences [Wang and Han 2004]. The main reason for this performance is the fact that depth-first traversal utilizes far less memory, more directed search space, and thus less candidate sequence generation than breadth-first or post-order which are used by some early algorithms. Depth-first traversal must be used in any reliable sequential pattern-mining algorithm that uses tree projection or hierarchical mining methods. Algorithms that use depth-first traversal as indicated in the taxonomy of Figure 4 are SPADE, SPAM,

	Apriori-Based		Pattern-Growth								Early-Pruning		
Algorithm	**Generate-and-test**	**multiple scans of the database**	**sampling and/or compress-ion**	**candidate sequence pruning**	**search space partition-ing**	**tree project-ion**	**depth-first traversal**	**suffix growth**	**prefix growth**	**memory-only**	**support counting avoidance**	**vertical projection of the DB**	**position coded**
AprioriAll	X	X	X		X								
GSP	X	X		X									
SPADE	X			X	X		X					X	
FreeSpan					X					X			
WAP-mine*			X		X	X		X		X			
PrefixSpan				X	X				X	X			
SPAM	X						X			X		X	
FS-Miner*			X	X	X	X	X	X					
DISC-all				X	X		X		X		X	X	X
Apriori-GST*	X		X	X									
PLWAP*			X			X	X		X				X
HVSM	X			X							X	X	X
LAPIN*				X	X		X		X	X	X	X	X

*algorithm is specifically for web log mining or single itemset sequence, or a variation for web log mining is provided by the authors.

Fig. 4. Tabular taxonomy of sequential-pattern mining algorithms.

FS-Miner, DISC-all, PLWAP, and LAPIN, while GSP, WAP-mine, and HVSM use breadth-first traversal. In general, pattern growth methods can be seen as depth-first traversal algorithms, since they construct each pattern separately, in a recursive way.

(6) *Suffix/prefix growth:* Algorithms that depend on projected databases and conditional search in trees first find the frequent *k*-sequences, usually 1-sequences, and hold each frequent item as either prefix or suffix, then start building candidate sequences around these items and mine them recursively. The idea behind this is that frequent subsequences can always be found by growing a frequent prefix/suffix; since it is usually shared among a good number of these sequences. This greatly reduces the amount of memory required to store all the different candidate sequences that share the same prefix/suffix. The difference in performance between depth-first traversal with prefix growth, as opposed to breadth-first with suffix growth, can be seen by comparing the suffix-growing WAP-mine with the prefix-growing PLWAP. These two methods can be considered the best models of pattern-growth tree projection sequential pattern-mining techniques, although PLWAP outperforms WAP-mine by cutting off recursive building of intermediate WAP-trees during mining. FS-Miner, on the other hand, builds intermediate conditional base trees in a backward fashion during mining as it grows on the suffix.

(7) *Memory-only:* This feature targets algorithms that do not spawn an explosive number of candidate sequences, which enables them to have minimum I/O cost. Such algorithms usually employ early pruning of candidate sequences and search space partitioning. Other ways to reduce I/O cost and maintain sequences in main memory only include mining a compressed representative structure of the sequence database or samples of it, instead of the original complete database.

3.3. Three More Features in the Taxonomy of Early-Pruning Algorithms

(1) *Support counting avoidance:* Several recent algorithms have found a way to compute support of candidate sequences without carrying a count throughout the mining process, and without scanning the sequence database iteratively. Some methods include bitwise or logical operations (e.g., SPADE, SPAM, LAPIN-SPAM [Yang and Kitsuregawa 2005]); some use positional tables (e.g., LAPIN, HVSM, DISC-all); and others store support along with each item in a tree structure (e.g., FS-Miner, WAP-tree, PLWAP). It is very important for an efficient algorithm not to scan the sequence database each time to compute support. A big advantage of having this feature is that a sequence database can be removed from memory and no longer be used once the algorithm finds a way to store candidate sequences along with support counts in a tree structure, or any other representation for that matter.

(2) *Vertical projection of the database:* Here, the algorithm visits the sequence database only once or twice, a bitmap or a position indication table is constructed for each frequent item, thus, obtaining a vertical layout of the database rather than the usual horizontal form. The idea is not new, it has been used in SPADE [Zaki 1998], where it was called the *id-list*. The mining process uses only the vertical layout tables to generate candidate sequences and counts support in different ways which have been discussed previously (logical intersection, bitwise ANDing, simple add up of counts, etc.). This way, an algorithm utilizes less memory during the mining process, in comparison with tree projections and projected databases. Its uses less I/O, because the sequence database is no longer required during mining. Early-pruning algorithms use this approach along with bitmaps or positional tables. Clearly, this might be the

best way to reduce memory consumption. One disadvantage of bitmap representations is the amount of computation incurred by bitwise (usually AND) operations used to count the support for each candidate sequence.

(3) *Position-coded:* This feature (also called position induction) is the key idea in early-pruning algorithms, as discussed above. It enables an algorithm to look-ahead to avoid generating infrequent candidate sequences. If a certain item (or sequence) is infrequent, then there is no point in growing the suffix or the prefix of the sequence (due to the downward closure of the apriori property). This feature also plays a major role in PLWAP, making it a hybrid pattern-growth\early-pruning algorithm that outperforms WAP-mine and PrefixSpan with low minimum support when there is large amount of mined frequent patterns. In this case, a binary coded position tree is virtually constructed from any given WAP-tree-like data structure, say W, using a coding scheme initially derived from the binary tree equivalent of W. This scheme assigns unique binary position codes to nodes of W. This position code is used later to determine which node falls in the suffix of the conditional prefix node being mined so that it does not participate in calculating its support, and thus early-prune the infrequent candidate sequence that might arise from mining the suffix tree.

3.4. Taxonomy with Performance Analysis

In this section, we present the proposed taxonomy, illustrated in Figure 4. Figure 5 summarizes the supporting theory which each algorithm in the taxonomy is based on. Figure 6 provides a top-down hierarchical view of the taxonomy, while Figure 7 shows a comparative performance analysis of algorithms from each of the taxonomy categories.

Experimentation was performed on a 1.87GHz Intel Core Duo computer with 2 gigabytes of memory, running Windows Vista 32-bit, with the same implementations of the programs on synthetic data sets generated using the IBM resource data generator code [Agrawal et al. 1993]; also tested on a real data set from the School of Computer Science web log. The following parameters are used to generate the data sets as described in Agrawal and Srikant [1995]: | D | represents the number of sequences in the database; | C | is the average length of the sequences; | S | is the average length of a maximal potentially frequent sequence; | N | is the number of events; and | T | is the average number of items per transaction. Two data sets were used, a medium-size data set described as C5T3S5N50D200K and a large-size data set described as C15T8S8N120D800K. These were run at different minimum support values: low minimum supports of between 0.1% and 0.9% and regular minimum supports of 1% to 10%. CPU execution time is reported by the program of each algorithm in seconds, while physical memory usage was measured using the *Windows Task Manager* in Megabytes (MB) by pressing CTRL-ALT-DEL before the program started on the running operating system, and by reading the memory usage at the highest observed value. GSP, PLWAP,[1] and WAP-mine were initially implemented with the C++ language running under the Inprise C++ Builder environment and compiled on the command line of MS Visual Studio 9 running on the operating system described above, while the code for SPAM,[2] PrefixSpan,[3] and LAPIN[4] were downloaded from their respective authors' websites and used as provided. We ran each algorithm with dedicated system resources alone.

[1] GSP, PLWAP, and WAP are available at http://cs.uwindsor.ca/∼cezeife/codes.html.
[2] http://himalaya-tools.sourceforge.net/Spam/.
[3] http://illimine.cs.uiuc.edu/.
[4] http://www.tkl.iis.u-tokyo.ac.jp/∼yangzl/soft/LAPIN/index.html.

Year	Algorithm	Theoretical basis
1995	AprioriAll [Agrawal and Srikant 1995]	Apriori property and Apriori-generate join.
1996	GSP [Srikant and Agrawal 1996]	GSP-join.
1998/2001	SPADE [Zaki 2001]	Temporal joins over vertical database projections. Lattice structure.
2000	FreeSpan [Han et al. 2000]	S-Matrix and database annotations for search space reduction.
2000	WAP-mine [Pei et al. 2000]	Conditional search on suffix-projected WAP-tree.
2001	PrefixSpan [Pei et al. 2001]	Prefix heuristic.
2002	SPAM [Ayers et al. 2002]	Apriori-based candidate generation and pruning using depth-first traversal.
2003	PLWAP [Lu and Ezeife 2003]	Conditional search on prefix-projected PLWAP-tree using binary position code assignment .
2004	DISC-all [Chiu 2004]	Direct sequence comparison (DISC) and search space partitioning with counting arrays.
2004	FS-Miner [El-Sayed et al. 2004]	Conditional search on suffix tree.
2005	Apriori-GST [Goethals 2005]	Using suffix tree to index candidate sequences for support counting.
2005	HVSM [Song et al. 2005]	Using location information for support counting avoidance.
2007	LAPIN [Yang et al. 2007]	Last position induction.

Fig. 5. Theoretical basis for sequential pattern-mining algorithms.

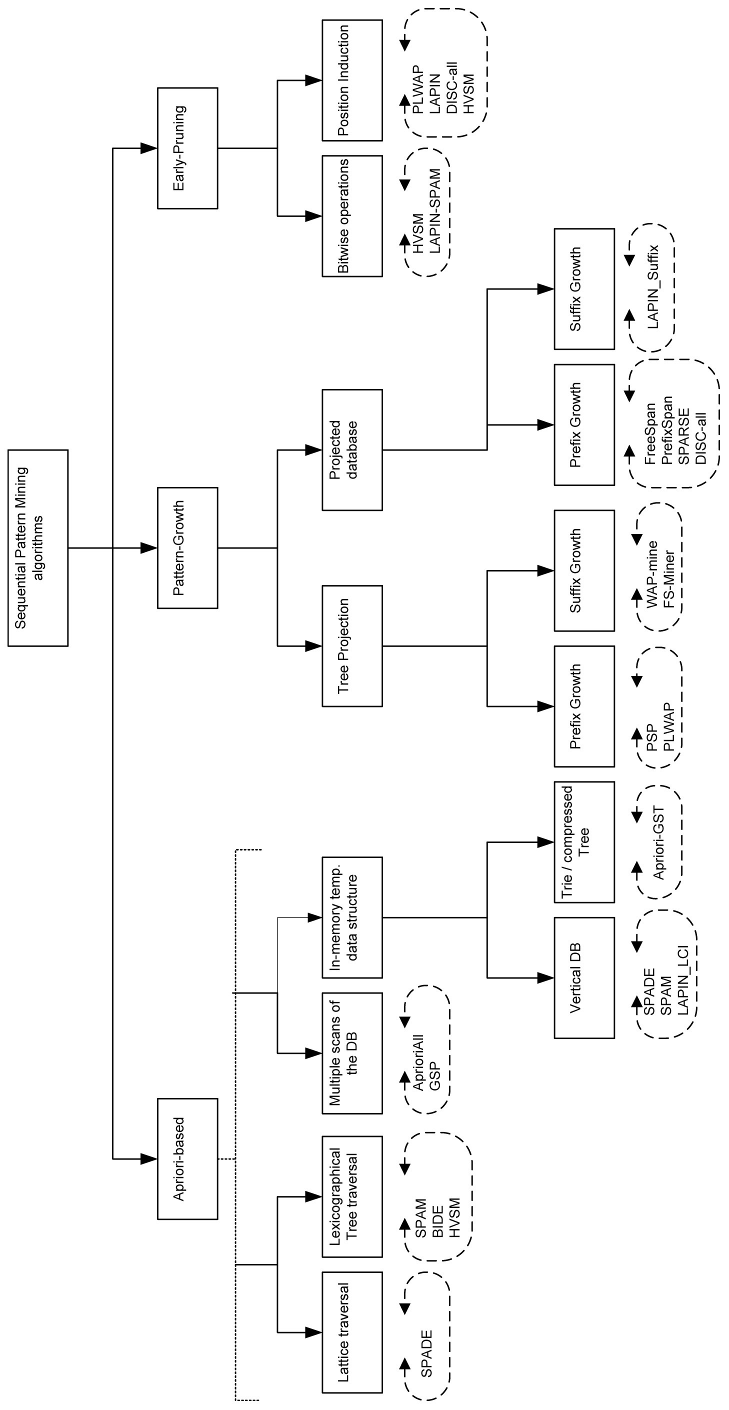


Fig. 6. A hierarchical taxonomy of significant sequential pattern-mining algorithms.

Algorithm	Data set size	Minimum Support	Execution Time (sec)	Memory Usage (MB)
GSP *Apriori-based*	Medium (| D |=200K)	Low (0.1%)	>3600	800
		Medium (1%)	2126	687
	Large (| D |=800K)	Low (0.1%)	-	-
		Medium (1%)	-	-
SPAM *Apriori-based*	Medium (| D |=200K)	Low (0.1%)	-	-
		Medium (1%)	136	574
	Large (| D |=800K)	Low (0.1%)	-	-
		Medium (1%)	674	1052
PrefixSpan *Pattern-Growth*	Medium (| D |=200K)	Low (0.1%)	31	13
		Medium (1%)	5	10
	Large (| D |=800K)	Low (0.1%)	1958	525
		Medium (1%)	798	320
WAP-mine *Pattern-Growth*	Medium (| D |=200K)	Low (0.1%)	-	-
		Medium (1%)	27	0.556
	Large (| D |=800K)	Low (0.1%)	-	-
		Medium (1%)	50	5
LAPIN_Suffix *Early Pruning*	Medium (| D |=200K)	Low (0.1%)	>3600	-
		Medium (1%)	7	8
	Large (| D |=800K)	Low (0.1%)	-	-
		Medium (1%)	201	300
PLWAP *Hybrid*	Medium (| D |=200K)	Low (0.1%)	23	5
		Medium (1%)	10	0.556
	Large (| D |=800K)	Low (0.1%)	32	9
		Medium (1%)	21	2

Fig. 7. Comparative analysis of algorithm performance. The symbol "-" means an algorithm crashes with the parameters provided, and memory usage could not be measured.

Careful investigation of Figure 7 shows how slow the apriori-based SPAM algorithm could become as data set size grows from medium (| D |=200K) to large (| D |=800K), due to the increased number of AND operations and the traversal of the large lexicographical tree; although it is a little faster than PrefixSpan on large data sets due to the utilization of bitmaps as compared to the projected databases of PrefixSpan. We can see that PLWAP enjoys the fastest execution times, as it clearly separates itself from WAP-mine and PrefixSpan (from the same category of algorithms), especially at low minimum support values when more frequent patterns are found and with large data sets. The version we used for PrefixSpan is the one using *pseudo projection* [Pei et al. 2001] in managing its projected databases. PrefixSpan initially scans the whole projected database to find frequent sequences and count their supports; the same process is used in WAP-mine with its conditional trees, as each conditional tree presents candidate sequences that require counting, support. Tracing PrefixSpan, WAP-mine and PLWAP manually on problem 1 with $min\_sup = 3$ (discussed in detail in Section 4) shows the following in the case of PrefixSpan, 9 projected databases were created and at least 4 scans of the projected databases took place; while for WAP-mine, 5 conditional trees were created, and conditional databases were scanned at least 7 times in total, including the initial 2 scans of the sequence database. On the other hand, the case of PLWAP shows that only 2 scans of the sequence database took place, and support-counting for frequent sequences simply entails adding up node counts during recursive mining of the virtual conditional subtrees, without having to scan any conditional databases. These findings are reflected on memory consumption during actual runtime of the algorithms on the data sets used for experimentation. It was observed, at that time, that PrefixSpan uses more memory gradually, it then releases it, also gradually, as mining progresses; while in the case of WAP-mine, more memory gets consumed faster than in PrefixSpan as intermediate trees are created and mined. Then, suddenly, all memory is released when recursion has completely unfolded. In the case of PLWAP, more is consumed, fast, then consumption stays at a stable volume (smaller than for the WAP-tree) during mining of the tree, then suddenly it is all released at once (not gradually) towards the end of the mining process. This also verifies why projected database techniques use more memory than tree projection (represented by PLWAP and WAP-mine) techniques, as can be seen from Figure 7, when small minimum support values are introduced. We can also see that PLWAP and WAP-mine keep execution times close to each other for larger minimum support values where there are no frequent patterns mined. This is because the gain in performance by PLWAP mostly occurs when there are more frequent patterns mined. PrefixSpan also shows a high speed on data sets with high support when no frequent patterns are found. For example, it only took 0.3sec to mine the medium data set with a support of 10%. While LAPIN_Suffix shows performance close to PrefixSpan, it uses less physical memory. PLWAP shows less memory usage than WAP-mine. PrefixSpan, on the other hand, uses more memory than any pattern-growth algorithm, even for small data sets as minimum support decreases. We also see that it requires at least 2MB of memory just for scanning the sequence database looking for frequent 1-sequences, and then memory consumption increases exponentially as projected databases are being created. Most algorithms could not handle mining long sequences with more than 10 events.

In another experiment, the algorithms' scalability was tested as the data set went from sparse to dense. In this case, four data sets were used, namely, C8T5S4N100D200K (a sparse data set with maximum sequence length of 22 items); C10T6S5N80D200K (a less sparse data set with maximum sequence length of 24 items); C12T8S6N60D200K (a dense data set with maximum sequence length of 31 items); and C15T10S8N20D200K (a more dense data set with maximum sequence length of 33 items). Dense data sets are characterized by having a small number of unique items | N | and a large number of customer sequences | C | and | T |; while sparse data sets are characterized by having a large number of unique items | N | (e.g., a large number of web pages in a given web site) and shorter customer sequences | C | and | T | (e.g., short user browsing sessions). Here we show only execution time results (in seconds), reported in Figure 8, at minimum support of 1%. WAP-mine and PLWAP perform faster on sparse data sets than the other algorithms because they employ depth-first search of the projected sparse trees. Also, as the data set gets more dense, PLWAP emerges with the least CPU time, as no intermediate trees or projected databases are created. GSP also has increased execution time because more frequent 1-sequences occur in dense data sets than in sparse ones.

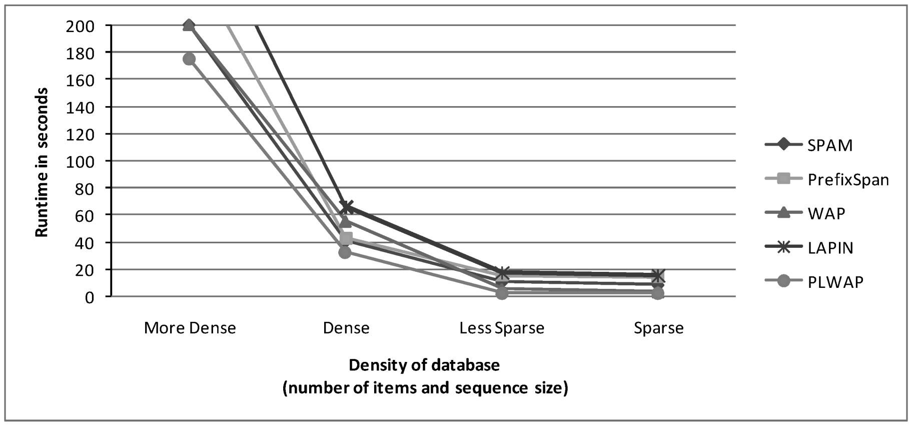
4. SEQUENTIAL PATTERN-MINING ALGORITHMS

Discussion of the surveyed algorithms is presented with running examples in this section. Section 4.1 reviews apriori-based algorithms; Section 4.2 summarizes pattern-growth algorithms; Section 4.3 reviews early-pruning algorithm; and Section 4.4 discusses hybrid algorithms. While algorithms mining single-item event sequences are discussed via the Problem 1 database (Table I), those mining multiple-itemset event sequences are discussed via the Problem 2 database (Table II).

Problem 1*:* Suppose D (shown in Table I) is a database of web access sequences extracted from some web log, and arranged according to users (User IDs) and their transactions (TIDs), where each transaction t_i is a sequence of page views, and $C_1 = \{a, b, c, d, e\}$ denotes the set of candidate 1-sequences, then the task is to find the set L of frequent sequences (also known as frequent sequential patterns), given a minimum support threshold of 60%.

	GSP *Apriori-based*	**SPAM** *Apriori-based*	**PrefixSpan** *Pattern-Growth*	**WAP-mine** *Pattern-Growth*	**LAPIN_Suffix** *Early Pruning*	**PLWAP** *Hybrid*
More Dense C15T10S8N20D200K	1171	200	252	200	300	175
Dense C12T8S6N60D200K	1782	40	43	55	66	32
Less Sparse C10T6S5N80D200K	2000	10	15	5	17	2
Sparse C8T5S4N100D200K	2421	8	14	3	15	2

(a)



(b)

Fig. 8. (a) Sequence database density vs. algorithm execution time (in sec), at minimum support of 1%. (b) Sequence Database density vs. algorithm execution time (in sec), at minimum support of 1%.GSP is not shown, as it is out of range of the vertical axis.

Table I. Sample Sequence Database D

User ID	TID	Access Sequence
100	t_1	$< bcbae >$
200	t_2	$< bacbae >$
300	t_3	$< abe >$
400	t_4	$< abebd >$
500	t_5	$< abad >$

Table II. Sequence Database D

Sequence ID for each Customer	Data Sequence
1	*<(a)(e)>*
2	*<(fg)a(fbkc)>*
3	*<(ahcd)>*
4	*<a(abcd)e>*
5	*<e>*

4.1. Apriori-Based Techniques

4.1.1. AprioriAll [Agrawal and Srikant 1995]. AprioriAll scans the database several times to find frequent itemsets of size k at each kth-iteration (starting from $k = 2$). It also has the generate-and-test feature by performing the *Apriori-generate* join procedure [Agrawal and Srikant 1994] to join L_{k-1} with itself to generate C_k, the set of candidate sequences in the kth-iteration, it then prunes sequences in C_k which have subsequences not in L_{k-1} (i.e., are not large), creates L_k by adding all sequences from C_k with support $\geq$ *min_sup* until there are no more candidate sequences. For example, with C_1 for the database in Table I, $C_1 = \{a{:}5,\ b{:}5,\ c{:}2,\ d{:}2,\ e{:}4\}$, where each item is represented with its support in the form *sequence:support count*, and a minimum absolute support of *min_sup* $= 3$ (60%), after pruning, the list of frequent 1-sequences L_1 will be $L_1 = \{a{:}5,\ b{:}5,\ e{:}4\}$. AprioriAll now generates candidate sequences at $k = 2$, $C_2 = L_1 \bowtie_{ap-gen} L_1 = \{aa{:}2, ab{:}4, ae{:}4, ba{:}3, bb{:}3, be{:}4, ea{:}0, eb{:}1, ee{:}0\}$ next it prunes all sequences in C_2 that do not have frequent subsequences (i.e., their subsequences do not appear in L_1), and so on, resulting in the frequent set $fs = \cup_k L_k = \{a,b,e,ab,ae,ba,bb,be,abe\}$. The pruning procedure performed on itemsets in C_k removes sequences that have infrequent subsequences. The AprioriAll technique suffers from increased delays in mining as the number of sequences in the database gets larger. It also suffers from exponential growth of candidate sequences during execution, as we have previously shown in Section 3.1. Several solutions were presented, including hashing to reduce the size of the candidate sequences [Park et al. 1995]; transaction reduction [Agrawal and Srikant 1994; Han and Fu 1995]; database partitioning [Savasere el al. 1995]; sampling [Toivonen 1996]; and dynamic itemset counting [Brin et al. 1997].

4.1.2. GSP [Srikant and Agrawal 1996]. The GSP algorithm, running on database D of Table I adopts a multiple-pass candidate generate-and-test method for finding sequential patterns. The first pass on the database determines the support for each item in finding frequent 1-sequences based on the minimum support of 3 records out of the 5 (i.e., 60%). The first ($k = 1$) scan over the table generates the set of candidate 1-sequences $C_1 = \{a{:}5,\ b{:}5,\ c{:}2,\ d{:}2, e{:}4\}$, giving the seed set of frequent 1-sequences $L_1 = \{a, b, e\}$. These sequences provide a *seed set* L_k which is used to generate next-level candidate sequences in the next pass k+1. The next C_{k+1} candidate set is generated by performing a GSP-join of L_k on itself. The GSP-join, like the apriori-generate join requires that two sequences in L_k join together if two conditions are met. Here, a sequence s_1 (e.g., ab) in L_k joins another sequence s_2 (e.g., ba) in L_k if the subsequence obtained by dropping the first $(k - 1)$ items of s_1 is the same as the subsequence obtained by dropping the last $(k - 1)$ items of s_2, hence the candidate sequence generated is the sequence s_1 extended with the last item in s_2 (e.g., aba is the result of joining ab and ba) and is added to C_k. The prune phase deletes candidate sequences that have a contiguous $(k - 1)$-subsequence with support less than *min_sup*. Each candidate sequence has one more item than a seed k-sequence, so all candidate sequences have the same number of items at each level. Support for these candidate sequences is again found during a pass over the data. The algorithm terminates when no new sequential pattern is found in a pass, or no candidate sequence can be generated. The generate-and-test feature of candidate sequences in GSP consists of two phases: Join Phase and Prune Phase. During the join phase, candidate sequences are generated by joining L_{k-1} with itself using *GSP-join*. From the seed set of three 1-sequences $L_1 = \{a, b, e\}$, a candidate set C_2 of twelve 2-sequences ($3 \times 3 + \frac{3 \times 2}{2} = 12$) is generated, giving an exploding number of candidate sequences, $C_2 = \{aa, ab, ae, ba, bb, be, ea, eb, ee, (ab), (ae), (be)\}$, where parenthesis denote contiguous sequences (i.e., no time gaps), the algorithm then scans the sequence database for each candidate sequence to count its support, after the pruning phase, to get $L_2 = \{ab{:}3, ae{:}4, ba{:}3, bb{:}3, be{:}4, (ab){:}3\}$, now GSP-join L_2 with L_2 (i.e., $L_2 \bowtie_{GSP} L_2$) to get $C_3 = \{aba, abb, abe, bab, bae, b(ab), bba, bbe, bbb, (ab)a, (ab)b, (ab)e\}$, and so on, until $C_k = \{\}$ or $L_{k-1} = \{\}$. The set of mined frequent sequences is eventually $fs = \cup_k L_k$. To show an example of pruning the contiguous subsequence, suppose we are given some $L_3 = \{(ab)c, (ab)d, a(cd), (ac)e, b(cd),$

bce}, then C_4={*(ab)(cd), (ab)ce*}, and *(ab)ce* will be pruned because its contiguous subsequence *ace* is not in L_3. For increased efficiency, GSP employs a hash-tree to reduce the number of candidates in C that are checked for sequences. Note that, at each step, GSP only maintains in memory the already discovered patterns and the *k*-candidates, thus making it not a memory-only algorithm. GSP is reported to be 2 to 20 times faster than AprioriAll.

4.1.3. PSP [Masseglia et al. 1999]. This is another apriori-based algorithm, also built around GSP, but the difference of using a prefix-tree. In PSP [Masseglia et al. 1999], user access sequences are sorted in the web log according to the IP address. The user is allowed to provide a time period Δt by which access sequences that are temporally close to each other are grouped. A prefix-tree is then built to handle the mining procedure in a way similar to GSP. Following up on tree traversal, graph traversal mining was proposed by Nanopoulos and Manolopoulos [2000], which uses a simple unweighted graph to reflect the relationship between pages of websites. The algorithm is similar to apriori, without performing the *Apriori-generate* join. The database still has to be scanned several times, but it is more efficient than GSP.

Problem 2*:* Given the sequence database in Table II, which we use instead of Table I as a different sequence database for algorithms that consider *k*-itemset sequences (and not only 1-itemset sequences) in their sequence database, the task is to find all frequent patterns, given a minimum support threshold of 25%.

4.1.4. SPAM [Ayers et al. 2002]. SPAM integrates the ideas of GSP, SPADE, and FreeSpan. The entire algorithm with its data structures fits in main memory, and is claimed to be the first strategy for mining sequential patterns to traverse the lexicographical sequence tree in depth-first fashion. For example, for SPAM, consider the database of Table II. The lexicographical subtree for item *a* is provided in Ayers et al. [2002], Figure 1 shows the lexicographical subtree for item *b*, assuming a maximum sequence size of 3. Each node in the tree has sequence-extended children sequences generated in the *S-Step* of the algorithm, and itemset-extended children sequences generated by the *I-Step* of the algorithm at each node. SPAM traverses the sequence tree in depth-first search manner and checks the support of each sequence-extended or itemset-extended child against *min_sup* recursively. If the support of a certain child *s* is less than *min_sup*, there is no need to repeat depth-first search on *s* by the apriori property. Apriori-based pruning is also applied at each S-Step and I-Step of the algorithm, minimizing the number of children nodes and making sure that all nodes corresponding to frequent sequences are visited. For efficient support-counting, SPAM uses a vertical bitmap data structure representation of the database similar to the id-list in SPADE. Each bitmap has a bit corresponding to each element of the sequences in the database. Each bitmap partition of a sequence to be extended in the S-Step is first transformed using a lookup table, such that all the bits after the index of the first "1" bit (call it index *y*) are set to one and all the bits with index less than or equal to *y* are set to zero. Then, the resulting bitmap can be obtained by the ANDing operation of the transformed bitmap and the bitmap of the appended item. In the I-step, ANDing is performed directly without transformation of the sequence. Now support-counting becomes a simple count of how many bitmap partitions, not containing all zeros. Each item in Table II will be represented by a vertical bitmap. The AND bitwise operation is used for checking support in the S-Step generation of, say, <*ae*>. <*a*>'s bitmap, is transformed as described above and a regular AND operation is performed with the bitmap for <*e*> to generate the bitmap of <*ae*>. If we use *min_sup* = 2, then the candidate sequence <*ae*> will not be pruned because the number of nonzero bitmap partitions in this case is ≥ 2.

SPAM is similar to SPADE, but it uses bitwise operations rather than regular and temporal joins. When SPAM was compared to SPADE, it was found to outperform SPADE by a factor of 2.5, while SPADE is 5 to 20 times more space-efficient than SPAM, making the choice between the two a matter of a space-time trade-off. Here, Ayres et al. [2002] propose to use a compressed bitmap representation in SPAM to save space, but do not elaborate on the idea.

4.2. Pattern-Growth Techniques

Soon after the apriori-based methods of the mid-1990s, the pattern growth-method emerged in the early 2000s, as a solution to the problem of generate-and-test. The key idea is to avoid the candidate generation step altogether, and to focus the search on a restricted portion of the initial database. The search space partitioning feature plays an important role in pattern-growth. Almost every pattern-growth algorithm starts by building a representation of the database to be mined, then proposes a way to partition the search space, and generates as few candidate sequences as possible by growing on the already mined frequent sequences, and applying the apriori property as the search space is being traversed recursively looking for frequent sequences. The early algorithms started by using *projected databases*, for example, FreeSpan [Han et al. 2000], PrefixSpan [Pei et al. 2001], with the latter being the most influential. A subsequence α' of sequence α is called *a projection* of α w.r.t. (with respect to) prefix β if and only if (1) α' has prefix β, and (2) there exists no proper supersequence α'' of α' such that α'' is a subsequence of α and also has prefix β [Pei at al. 2001]. PrefixSpan is based on recursively constructing the patterns by growing on the prefix, and simultaneously, restricting the search to projected databases. This way, the search space is reduced at each step, allowing for better performance in the presence of small support thresholds. PrefixSpan is still considered a benchmark and one of the fastest sequential mining algorithms alongside SPADE. Another algorithm, WAP-mine [Pei et al. 2000] is the first of the pattern-growth algorithms to use a physical tree structure as a representation of the sequence database along with support counts, and then to mine this tree for frequent sequences instead of scanning the complete sequence database in each step.

4.2.1. FreeSpan [Han et al. 2000]. FreeSpan stands for Frequent Pattern-Projected Sequential Pattern Mining, and starts by creating a list of frequent 1-sequences from the sequence database called the *frequent item list (f-list)*, it then constructs a lower-triangular matrix of the items in this list. This matrix contains information about the support count of every 2-sequence candidate sequence that can be generated using items in the f-list, and is called *S-Matrix*. For a sequential pattern α from S-Matrix, the *α-projected database* is considered the collection of frequent sequences having α as a subsequence. Infrequent items and items following those infrequent items in α are ignored. For the next step a table is constructed with length 2-sequences (i.e., frequent 2-sequences) along with annotations on repeating items and annotations on projected databases that help in locating these projected databases in the third and last scans of the database without referring to the S-Matrix. The S-Matrix is now discarded, and mining picks up using the projected databases. As an example, we discuss a solution to Problem 2 with database D of Table II and *min_sup* of 25% using FreeSpan algorithm. The first scan of D generates the list of frequent 1-sequences as *f-list* = $\{a{:}4, c{:}3, e{:}3, b{:}2, d{:}2\}$, thus the complete set of sequential patterns can be divided into 5 disjoint subsets, each of which has its own projected database and is mined in a divide-and-conquer method. Now construct the S-Matrix to count the support of each 2-sequence, as follows. Consider the f-list $\{i_1, i_2, \ldots, i_n\}$, F is a triangular matrix $F[j, k]$ where

Table III. S-Matrix for Constructing 2-Sequences from Database in Table II

a	1				
c	(2,0,2)	0			
e	(2,0,0)	(1,0,0)	0		
b	(2,0,1)	(0,0,0)	(0,1,0)	0	
d	(1,0,2)	(0,0,2)	(0,1,0)	(0,0,1)	0
	a	c	e	b	d

Table IV. Pattern Generation from S-Matrix

Item	Frequent 2-Sequences	Annotations on Repeating Items	Annotations on Projected DBs
d	*<(ad)>:2, <(cd)>:2*	*<(ad)>, <(cd)>*	*<(ad)>:{c}*
b	*<ab>:2*	*<ab>*	Ø
e	*<ae>:2*	*<ae>*	Ø
c	*<ac>:2, <(ac)>:2*	*{ac}*	Ø
a	Ø	Ø	Ø

$1 \le j \le m$ and $1 \le k \le j$, such that m is the number of items in the itemset of the sequence under consideration. $F[j, j]$ has only one counter, which is the support count for 2-sequence $<i_j i_j>$. Every other entry has three counters (A, B, C); A is the number of occurrences of $< i_j i_k>$, B is the number of occurrences of $< i_k i_j>$, and C is the number of occurrences in which i_k occurs concurrently with i_j as an itemset $<(i_j\, i_k)>$.

The S-Matrix for the 2-sequences of Table II can be seen in Table III, which is filled up during a second scan of the database D. For example, the entry (2,0,2) in the second row, first column of the matrix in Table III, means that the sequence $<ac>$ occurs 2 times, the sequence $<ca>$ occurs zero times, and the sequence $<(ac)>$ occurs 2 times in D.

The third step builds level-2-sequences from the candidate sequences in the S-Matrix and finds annotations for repeating items and projected databases in order to discard the matrix and generate level-3 projected databases; see Table IV, built from parsing the matrix row-wise, bottom-up. Consider the row for d, since, $F[\ a, d]$, $F[\ c, d]$ and $F[\ a, c]$ form a pattern generating triple and $F[\ a, d] =$ (1,0,2), meaning only $<(ad)>$ is valid (because its support count is above the threshold), the annotation for the projected database should be *<(ad)>* :{c}, which indicates generating *<(ad)>*-projected database, with c as the only additional item included.

From the generated annotations, scan the database one more time to generate item-repeating patterns {*(ad):2, (cd):2, ab:2, ae:2, ac:2, (ac):2*}. There is only one projected database $<(ad)>$:{c} whose annotation contains exactly 3 items, and its associated sequential pattern is obtained by a simple scan of the projected database. If it contains more than 3 items, S-Matrix is constructed for this projected database and recursively mines it for sequential patterns the same way. FreeSpan examines substantially fewer combinations of subsequences and runs faster than GSP, due to the process of pruning candidate sequences in the S-Matrix before they are generated, especially when the dataset grows larger. The major cost of FreeSpan is the computation and creation of the projected databases. If a pattern appears in each sequence of a database, its projected database does not shrink. Also, since a k-subsequence may grow at any position, the search for a length $(k + 1)$ candidate sequence needs to check every possible combination, which is costly.

4.2.2. PrefixSpan [Pei et al. 2001]. PrefixSpan examines only the prefix subsequences and projects only their corresponding postfix subsequences into projected databases. This way, sequential patterns are grown in each projected database by exploring only

Table V. Running PrefixSpan on Table II

Prefix	Projected Database	Sequential Patterns
<a>	*<e>, <(bc)>, <(_cd)>, <(abcd)e>*	*a, ab, ac, (ac), (ad), ae*
<b>	*<(_c)>, <(_cd)e>*	*b, (bc)*
<c>	*<(_d)>, <(_d)e>*	*(cd)*
<d>	*<e>*	Ø
<e>	Ø	Ø

local frequent sequences. To illustrate the idea of projected databases, consider *<f>*, *<(fg)>*, *<(fg)a>* which are all prefixes of sequence *<(fg)a(fbkc)>* from Table II, but neither *<fa>* nor *<ga>* is considered a prefix. On the other hand *<(_ g)a(fbkc)>* is the *postfix* of the same sequence w.r.t. *<f>*, and *<a(fbkc)>* is the postfix w.r.t. prefix *<(fg)>*. A running example of PrefixSpan on database *D* (Table II) of Problem 2 acts in three steps:

(1) Find all 1-itemset sequential patterns by scanning the database *D*. We get *a:4, c:3, e:3, b:2, d:2* along with their support counts.
(2) Divide the search space to get projected databases like FreeSpan. This example generates 5 disjoint subsets according to the 5 prefixes *<a>*, *< b>*, *<c>*, *<d>*, *<e>*.
(3) Find subsets of sequential patterns; these subsets can be mined by constructing projected databases, like FreeSpan, and mining each one recursively.

To find sequential patterns having prefix *<a>*, we extend it by adding one item at a time. To add the next item *x*, there are two possibilities: (1) the algorithm joins the last itemset of the prefix (i.e., *<(ax)>*) and (2) it forms a separate itemset (i.e., *<ax>*) [Liu 2007]. So to produce *<a>* - projected database: if a sequence contains item *<a>*, then the suffix following the first *<a>* is extracted as a sequence in the projected database. Look at the second sequence (second row) of Table II, *<(fg)a(fbkc)>*, which is projected to *<(bc)>* where *f* and *k* are removed because they are infrequent. The third sequence is projected to *<(_cd)>*, the fourth to *<(abcd)e>*, eventually the final projected database for prefix *<a>* contains the following sequences: *e*, *(bc)*, *(_cd)*, *(abcd)e*; for the other prefixes see Table V. Now we need to find all frequent sequences of the form *<(ax)>*, two templates are used: *<(_x)>* and *<ax>* to match each projected sequence to accumulate the support count for each possible *x*(remember that *x* matches any item). The second template uses the last itemset in the prefix rather than only its last item. In the example here, they are the same because there is only one item in the last itemset of the prefix. Then, we need to find all frequent sequences of the form *<ax>*; in this case, *x*s are frequent items in the projected database that are not in the same itemset as the last item of the prefix. Table V contains all frequent sequential patterns generated for this example using PrefixSpan. Looking at the patterns generated for prefix *<a>*, after finding the frequent 2-sequences (namely, *ab, ac, (ac), (ad), ae)*, we recursively create projected databases for them and start mining for frequent 3-sequences (the example here does not have any).

The key advantage of PrefixSpan is that it does not generate any candidates. It only counts the frequency of local items. It utilizes a divide-and-conquer framework by creating subsets of sequential patterns (i.e., projected databases) that can be further divided when necessary.

PrefixSpan performs much better than both GSP and FreeSpan. The major cost of PrefixSpan is the construction of projected databases. To further improve mining efficiency, *bilevel projection* and *pseudo-projection* can be used. Bilevel projection uses the S-Matrix, introduced in FreeSpan [Han et al. 2000], instead of the projected databases. It contains support counters as well as all 1-sequences. Support counters in

Table VI. Frequent Subsequence of Web Access Sequence Database from Table I

TID	Access Sequence	Frequent Subsequences
t_1	<*bcbae*>	<*bbae*>
t_2	<*bacbae*>	<*babae*>
t_3	<*abe*>	<*abe*>
t_4	<*abebd*>	<*abeb*>
t_5	<*abad*>	<*aba*>

the S-Matrix tell which 3-sequences can be generated and which not, in order for the algorithm to search for them in the database. The authors refer to this as *3-way apriori checking*. Pseudo-projection is used when PrefixSpan runs only in main memory (no disk-based processing is allowed). That is, instead of creating physical projected databases in memory, pointers or pointer objects can be used, such that every projection contains two pieces of information: pointer to the sequence in the database and offset of the postfix in the sequence for each size of projection. Experimental results show that for disk-based processing, it is better to use bilevel projection, while for main memory processing using pseudo-projection gives a better performance [Pei et al. 2001].

4.2.3. WAP-mine [Pei et al. 2000]. *Pattern-Growth Miner with Tree Projection.* At the same time as FreeSpan and PrefixSpan in 2000/2001, another major contribution was made as a pattern growth and tree structure-mining technique, that is, is the WAP-mine algorithm [Pei et al. 2000] with its WAP-tree structure. Here the sequence database is scanned only twice to build the WAP-tree from frequent sequences along with their support, a "*header table*" is maintained to point at the first occurrence for each item in a frequent itemset, which is later tracked in a threaded way to mine the tree for frequent sequences, building on the suffix. The first scan of the database finds frequent 1-sequences and the second scan builds the WAP-tree with only frequent subsequences. As an example of a WAP-tree, the database for Problem 1 (in Tables I and VI) is mined with the same *min_sup* = 3 transactions.

The frequent subsequences (third column of Table VI) of each original database sequence is created by removing infrequent 1-sequences. The tree starts with an empty root node and builds downward by inserting each frequent subsequence as a branch from root to leaf. During the construction of the tree, a header link table is also constructed, which contains frequent 1-sequences (in our example *a*, *b*, and *e*), each one with a link connecting to its first occurrence in the tree and threading its way through the branches to each subsequent occurrence of its node type as shown in Figure 9(a). To mine the tree, WAP-mine algorithm starts with the least frequent item in the header link table and uses it as a *conditional suffix* to construct an intermediate conditional WAP-tree and finds frequent items building on the suffix to get frequent *k*-sequences. In this example (as in Figure 9(a)), we start with item *e*, which is added to the set of frequent sequences as $fs = \{e\}$ and follow its header links to find sequences *bba:1, baba:1, ab:2*, having supports of *b*(4) and *a*(4) to have *a* and *b* as frequent 1-sequences. Now, build a WAP-*sub*tree for suffix |*e* as in Figure 9(b) for sequences *bba:1, baba:1, ab:2* and mine it to obtain conditional suffix |*be* as shown in Figure 9(c) following the b-header link, causing sequence *be* to get added to the set of frequent sequences mined as $fs = \{e, be\}$. Following the header links of *b* in WAP-tree|*e* (conditional search on *e*) gives *b:1, ba:1, b:-1, a:2*, with supports of *b* (1) and *a* (3), *b*'s support is less than *min_sup* so we remove it, resulting in *a*:1, *a*:2, giving the conditional WAP-tree|*be* as in Figure 9(c).

Next, add the newly discovered frequent sequence to *fs*, building on the suffix as $fs = \{e, be, abe\}$ and follow its header link in WAP-tree|*be* to get Ø (Figure 9(c)). Now the

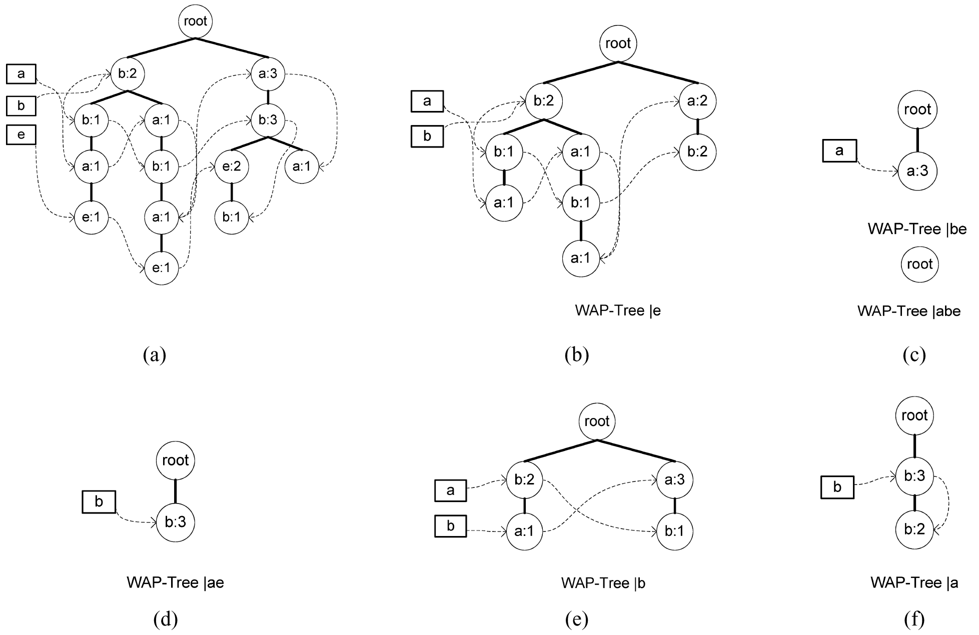


Fig. 9. Mining the WAP-tree using WAP-mine.

algorithm recursively backtracks to WAP-tree|*e* in Figure 9(b) to mine the *a* link for WAP-tree|*ae* (Figure 9(d)). Complete mining of our example WAP-tree can be traced in the rest of Figure 9 with a complete set of a frequent sequence *fs* = {*e, be, abe, ae, b, bb, ab, a, ba*}.

The WAP-mine algorithm is reported to have better scalability than GSP and to outperform it by a margin [Pei et al. 2000]. Although it scans the database only twice and can avoid the problem of generating explosive candidates as in apriori-based and candidate generate-and-test methods, WAP-mine suffers from a memory consumption problem, as it recursively reconstructs numerous intermediate WAP-trees during mining, and in particular, as the number of mined frequent patterns increases. This problem was solved by the PLWAP algorithm [Lu and Ezeife 2003], which builds on the prefix using position- coded nodes.

4.2.4. FS-Miner [El-Sayed et al. 2004]. Inspired by FP-tree [Han et al. 2000] and ISM [Parthasarathy et al. 1999], FS-Miner is a tree projection pattern growth algorithm that resembles WAP-mine and supports incremental and interactive mining. The significance of FS-Miner is that it starts mining immediately with 2-subsequences from the second (which is also the last scan) of the database (at $k = 2$). It is able to do so due to the compressed representation in the FS-tree, which utilizes a header table of edges (referred to as *links* in El-Sayed at el. [2004]) rather than single nodes and items, compared to WAP-tree and PLWAP-tree. It is also considered a variation of a trie, as it stores support count in nodes as well as edges of the tree that represents 2-sequences and is required for the incremental mining process. Consider Table I for a running example on FS-Miner, and keep in mind that it only considers contiguous sequences (e.g., *cba* is a contiguous subsequence of *bcbae*, but *ca* is not). Figure 10 shows the header table generated along with the FS-tree for sequences in

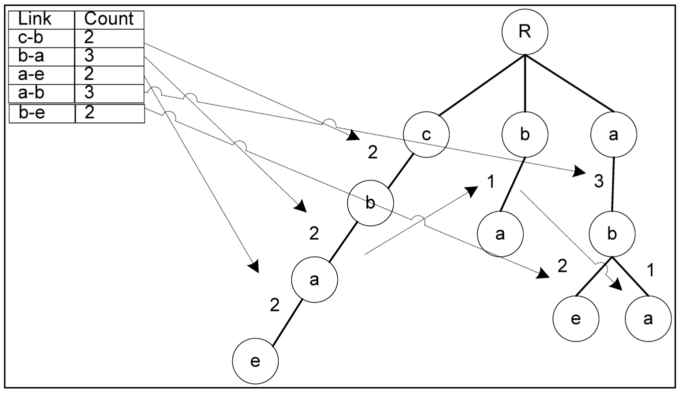


Fig. 10. FS-tree and header table HT for the web log from Table I.

Table I. FS-miner and the FS-tree maintain two kinds of support counts and minimum support, namely *MSuppC*^{seq}, the minimum *frequent sequence* support, similar to *min_sup* used throughout this article, and *MSuppC*^{link}, the minimum *frequent link* (a frequent 2-sequence) support. A link h with support count *Supp*$^{link}(h)$ is considered a frequent link if $Supp^{link}(h) \geq MSuppC^{seq}$, and is considered *potentially frequent link* if $MSuppC^{link} \leq Supp^{link}(h) < MSuppC^{seq}$. If $Supp^{link}(h)$ does not satisfy both $MSuppC^{link}$ and $MSuppC^{seq}$, then it is a *infrequent link*. Potentially frequent links are maintained to support and enable the incremental and interactive mining capability of FS-Miner.

Frequent links and potentially frequent links are marked in the header table (HT) in Figure 10. Only frequent links are used in mining. Assume $MSuppC^{link} = 2$ and $MSuppC^{seq} = 3$ for our example. The sequence database (Table I) is scanned once to find counts for links and to insert them in the header table. The FS-tree is built during the second scan of the database in a manner similar to WAP-tree and PLWAP-tree, except that a sequence that contains infrequent links is split and each part is inserted in the tree separately. Pointer links from the header table are built to connect occurrences of header table entries, as they are inserted into the FS-tree similar to WAP-tree linkage (i.e., first occurrence-based and not preordered). As only frequent links may appear in frequent sequences, only frequent links are considered during mining. El-Sayed et al. [2004] mention four important properties of the FS-tree, as follows:

(1) Any input sequence that has infrequent link(s) is pruned before being inserted into the FS-tree.
(2) If $MSuppC^{link} < MSuppC^{seq}$, the FS-tree is storing more information than required. This is a drawback as discussed in the features earlier in Section 3.2, but is required by FS-Miner to support incremental and interactive mining in a manner similar to ISM.
(3) All possible subsequences that end with a given frequent link h can be obtained by following the pointer of h from the header table to the correct FS-tree branches.
(4) In order to extract a sequence that ends with a certain link $h$from an FS-tree branch, we only need to examine the branch prefix path that ends with that link backward up to the tree root.

Now, the mining process takes four steps. First, and following the example, extract *derived paths,* by following pointer links from the header table for each frequent link, we get the link $b - a$, from which we can extract two derived paths ($c - b$:2, $b - a$:2), ($b-a$:1), ($a-b$:3, $b-a$:1), and the link $a-b$ with derived path ($a-b$:3). Second, *construct* *conditional sequence base*, by setting the frequency count of each link in the path to the count of the link h in its derived path, for $b-a$ we have the conditional set $(c-b{:}2)$ and $(a-b{:}1)$, $a-b$ does not have a conditional sequence base because it is the only link in its own path. Third, *construct conditional FS-tree*, and insert each of the paths from the conditional base of h ($b-a$ in this example) into it in a backward manner. Fourth, *extract frequent sequences* by performing depth-first traversal of the conditional tree and returning only sequences satisfying $MSuppC^{seq}$. Now the conditional base tree does not have any links that satisfy $MSuppC^{seq}$; suppose the link $c-b$ does satisfy $MSuppC^{seq}$, in this case we extract it from the tree and append to it $b-a$ because it is in the conditional suffix tree of $b-a$ (from step (3)) to get the frequent sequence $<cba>$. Eventually, the mining process will end up with list of contiguous frequent sequences (including the frequent links and the frequent 1-sequences) *fs* = {*a:5, b:5, e:4, ba:3, ab:3*}. No performance comparison of FS-Miner with WAP-mine or similar tree projection algorithms is provided in El-Sayed et al. [2004], rather it is compared to a variation of Apriori [Agrawal and Srikant 1994] that supports sequence data and is reported to outperform it. FS-Miner also shows excellent time performance when handling datasets with large number of distinct items. We mentioned several drawbacks of FS-Miner previously, such as the sparse and fast-growing tree structure due to the fact that sequences with infrequent links are split and represented with more than one branch in the tree, and the large amount of information stored in the tree. A credit we can give to FS-Miner after careful investigation, is that, as it supports incremental and interactive mining, the way it does so is very well planned in the design of the algorithm, and is also made simple.

4.3. Early-Pruning Techniques

Early-Pruning algorithms are emerging in the literature as a new breed for sequential pattern mining. These algorithms utilize a sort of *position induction* to prune candidate sequences very early in the mining process and to avoid support counting as much as possible. The rest of the mining process is simple pattern growth. The idea of position induction is as follows:

If an item's last position is smaller than the current prefix position [during mining], the item cannot appear behind the current prefix in the same customer sequence [Yang and Kitsuregawa 2005].

These algorithms usually employ a table to track the last positions of each item in the sequence and utilize this information for early candidate sequence pruning; as the last position of an item is the key used to judge whether the item can be appended to a given prefix *k*-sequence or not, thus avoiding support counting and generation of candidate infrequent sequences. LAPIN [Yang et al. 2007] is comparable to PrefixSpan, and is one of the promising algorithms in this category. There are different previous attempts for different mining scenarios, starting with LAPIN-SPAM [Yang and Kitsuregawa 2005] which was introduced as a solution to the overhead problem of bitwise operations in SPAM [Ayers et al. 2002]; and claimed to slash memory utilization by half; then LAPIN_LCI, LAPIN_Suffix [Yang et al. 2005] for suffix growth methods; and LAPIN_WEB [Yang et al. 2006] for web log mining. Yang and Kitsuregawa [2005] in LAPIN-SPAM did not conduct experiments or discuss cases of overhead processing delay for preparing the positions table and its compressed *key position* version. Further investigation needs to be done on whether constructing the optimized table introduces start-up delay. While LAPIN is different from LAPIN-SPAM in that it does not use a bitmap representation of the database, Song et al. [2005] in HVSM go further in stressing the importance of bitmap representation, and add to it what they call a *first-horizontal last-vertical* database a scanning method, then use a special tree structure

Table VII.

(a) sequence DB from Table II

SID	Sequence
10	*<(a)(e)>*
20	*<(fg)a(fbkc)>*
30	*<(ahcd)>*
40	*<a(abcd)e>*
50	<e>

(b) item-last-position list

SID	Last position of SE Item
10	$a_{last} = 1$, $e_{last} = 2$
20	$a_{last} = 1$, $c_{last} = 3$
30	$a_{last} = 1$, $c_{last} = 1$
40	$a_{last} = 2$, $c_{last} = 2$, $e_{last} = 3$
50	$e_{last} = 1$

(c) prefix border position set for <a>

SID	Last position of SE Item
10	$a_{last} = 1$, $e_{last} = 2$
20	$a_{last} = 1$, $c_{last} = 3$
30	$a_{last} = 1$, $c_{last} = 1$
40	$a_{last} = 2$, $c_{last} = 2$, $e_{last} = 3$
50	$e_{last} = 1$

(d) prefix border position set for <c>

SID	Last position of SE Item
10	$a_{last} = 1$, $e_{last} = 2$
20	$a_{last} = 1$, $c_{last} = 3$
30	$a_{last} = 1$, $c_{last} = 1$
40	$a_{last} = 2$, $c_{last} = 2$, $e_{last} = 3$
50	$e_{last} = 1$

where each node takes its frequent sibling nodes as its child nodes in the join process to extend the itemset, thus avoiding support counting; but HVSM's performance does not exceed that of SPAM. DISC-all [Chiu et al. 2004], on the other hand, employs temporal ordering besides the regular lexicographic ordering of items and itemsets in the sequence, and uses a variation of projected databases for partitioning the search space, depending on the location order of frequent 1-sequences.

4.3.1. LAPIN [Yang et al. 2007]. LAPIN (first introduced in a technical report [Yang et al. 2005]) uses an *item-last-position* list and *prefix border position set* instead of the tree projection or candidate generate-and-test techniques introduced so far. The main difference between LAPIN and previous significant algorithms is the scope of the search space. PrefixSpan scans the whole projected database to find the frequent patterns. SPADE temporally joins the whole *id-list* of the candidates to get the frequent patterns of next layer. LAPIN can get the same results by scanning only part of the search space of PrefixSpan and SPADE, which are indeed the positions of the items. For an example on LAPIN, consider the general sequence database of Problem 2 as in Table II with further illustration in Table VII(a), and minimum support *min_sup* = 3. The first scan of the database gets all the 1-sequence patterns $fs = \{a{:}4, c{:}3, e{:}3\}$ and generates the *item-last-position list* as shown in Table VII(b), which is an ordered list of the last position of each item in the sequence.

We show the case of prefix frequent sequence <*a*>, the positions of <*a*> in the sequence table represented in the *sid:eid* format are 10:1, 20:2, 30:1, 40:1. Next, we obtain the last indices whose positions are larger than <*a*>'s for each other frequent item, highlighted in Table VII(c) as 10:2, 20:2, 40:2; this is called the *prefix border position list* for prefix <*a*>. We use the indices in this list and start from the index position for each SID to the end of the sequence, scan for items that occur after that position and increment the support count for each item, resulting in <*a*>:2, <*c*>:2, <*e*>:2—none of which meets *min_sup*—if they were to meet *min_sup*, then we could obtain the frequent sequences <*ca*>, <*cc*>, <*ce*>, and then repeat on each of these sequences as conditional prefix, until finally we get the list of all frequent sequences from the sequence database.

From the above procedure, we can describe LAPIN as a prefix growth algorithm with efficient early pruning. It uses a depth-first traversal of the lexicographic tree as a model to grow the sequences with projected databases. Its main advantage over PrefixSpan, as claimed by Yang et al. [2005], is that the search space is reduced to almost half of that of PrefixSpan because it scans only a small portion of the

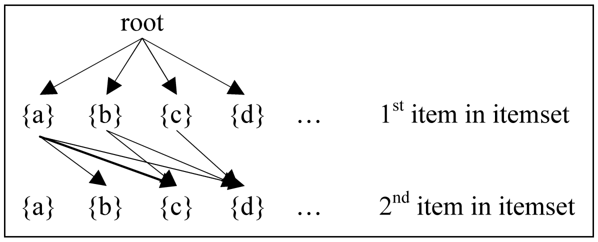

root
{a} {b} {c} {d} … 1st item in itemset
{a} {b} {c} {d} … 2nd item in itemset

Fig. 11. Candidate one-large-sequence and two-itemset C_2.

projected database (or the id-list when compared to SPADE) due to the utilization of the *item_last_position* list. Our experimental results in Figure 7 show that LAPIN performs better than PrefixSpan when the minimum support is higher than 1% and for large datasets of around 800K, but not better at lower minimum support values and smaller-sized datasets.

4.3.2. *HVSM* [Song et al. 2005]. Following the steps of SPAM, Song et al. [2005] propose the first-**H**orizontal-last-**V**ertical **S**canning database **M**ining algorithm, which also uses bitmap representation. HVSM defines the length of a sequence as the number of itemsets it contains rather than the number of single items, and it generates candidate frequent sequences by taking sibling nodes as child nodes layer by layer. In this case, the pruned frequent itemsets in one layer will no longer appear in the next layer. Support is counted by scanning a bitmap and recording the first TID in order to reduce the number of scans and improve counting efficiency. As an example, consider the database in Table II again, with $min\_sup = 2$. Consider a vertical database for this table showing only bitmaps for a, b, c, and d. To generate candidate 1-sequences, the algorithm scans the vertical bitmap for each itemset of each CID, when the first 1 appears, it records the corresponding TID and increments the support count of the itemset. If, at the end, the support count is more than or equal to *min_sup*, the itemset is appended to the list of frequent itemsets L_k and stored in the bitmap of the frequent 1-sequences along with the 1st TID of the first 1. In our example, this step scans the $<a>$ column for CID = 1; when it finds the first 1, in this case it happens to be in the 1st row, it records its TID as "1st-TID=1" and increments the support count for $<a>$. It does the same for CID = 2 and CID = 3,..., and so on. Eventually, we get support(a) = 4 $\geq$ *min_sup*. In this case, this step outputs $<a>$ as a frequent 1-sequence and adds its column to the bitmap of frequent 1-sequences. The end result of this step is the list L_1 of frequent 1-sequences with itemset size 1, namely, L_1: {a:4, b:2, c:3, d:2}. The next step searches for frequent 1-sequences with itemsets of size 2, called *one-large-sequence-two-itemset*. The lexicographical tree is created from the previous step's output, in which each leaf node now takes its sibling nodes as child nodes in order (combined in *Apriori-generate* join) as shown in Figure 11.

We obtain $C_2 = \{ab, ac, ad, bc, bd, cd\}$. Now, for each itemset in C_2 and for each CID, perform a bitwise AND operation between the first item and the second item in the itemset, and scan the resulting bitmap for the first 1, record its 1^{st} TID value and compute its support as in the previous step. For example, take the itemset $<ab>$, AND the bitmap of $<a>$ with the bitmap of $<b>$ in L_1, then scan the resulting bitmap $<ab>$ for each CID, when the first 1 appears record its TID as 1^{st} TID and add 1 to the $<ab>$ support count for each CID, at the end support $(ab) = 1 \ngeq min\_sup$, so it does not make it into the L_2 set (i.e., $ab \notin L_2$). Continue with the process to obtain all $L_2=\{bc{:}2, cd{:}2\}$. The lexicographical tree is pruned by removing edges that did not make it into L_2, then repeat the previous steps for frequent 1-sequences that have k-size itemsets. At the end, we get the complete bitmap for all frequent 1-sequences, which are used as input for mining frequent 2-sequences SL_k. For finding frequent k-sequences, all the itemsets generated for frequent $(k-1)$-sequences are used as nodes for the lexicographical tree of candidate k-sequences, which are then joined together by *seqgen*() in lexicographical order [Song et al. 2005]. The result is totally ordered, and is different from previous combinations. No bitwise ANDing operations are performed anymore, instead 1stTID information is used to indicate where to start support counting in the vertical database for candidate itemsets; this is the position induction feature in HVSM.

Performance evaluations, conducted by Song et al. [2005] revealed that HVSM outperforms SPAM for large transaction databases. One reason for this is that HVSM defines the length of a sequence as the number of itemsets it contains, extending the granularity of the sequential pattern. Another reason is that support counting happens simply by vertically scanning the bitmap and the fact that only 1st TID information is used in this process in an effort to avoid support counting and minimize bitwise operations. (In fact, they are no longer needed in high iterations of the algorithm.) The trade-off is that HVSM consumes more memory than SPAM.

4.3.3. DISC-all [Chiu et al. 2004] . As apriori-based algorithms prune infrequent sequences based on the antimonotone property, Direct Sequence Comparison (DISC) prunes infrequent sequences according to other sequences of the same length. To compare sequences of the same length, the procedure employs lexicographical ordering and temporal ordering, respectively. For example, the sequence A = <*a(bc)(de)*> is smaller than the sequence B = <*a(cd)(bf)*> in lexicographical order because $b < c$ in position 2, which is called the *differential point*. Moreover, if C = <*(abc)(de)*>, then C < A because of the temporal order of itemsets in position 1. According to this order, if μ_k is the smallest k-length subsequence of A, we call it *k-minimum subsequence*. Chiu et al. [2004] also use this relation to define a *k-minimum order* by which the customer sequences can be sorted into k-sorted databases. Given a minimum support count ξ, the k-minimum sequence at position ξ in the k-sorted database is called α_ξ, any k-minimum sequence appearing at position less than that of α_ξ (according to k-minimum order) is infrequent because the database is sorted. All k-minimum sequences less than α_ξ are pruned and, for each CID of these sequences, the algorithm tries to find the next k-minimum sequence (called *conditional k-minimum sequence*) such that it can be arranged in a position $\geq_k$ to α_ξ, generating a new k-sorted database with new α_ξ. We can see that all infrequent k-sequences smaller than α_ξ are skipped and only the conditional k-minimum subsequences are checked; this is how the algorithm is able to prune infrequent candidate sequences without counting their support. The proposed DISC-all algorithm uses DISC technology for candidate sequence pruning. At first, the database is scanned to divide the customer sequences into *first-level partitions* by their minimum 1-sequences and are ordered accordingly. Those partitions are mined for all frequent 2-sequences using PrefixSpan as projected databases, infrequent 1- and 2-sequencs are removed to generate a shorter customer sequence. Now each of these partitions in its turn is divided into *second-level partitions* by its 2-minimum sequence to find frequent 3-sequences, and so on. This is called *multi-level partitioning* and the number of levels is adaptive, depending on the trade-off between overhead and profit. The generation of partitions is executed in depth-first order. Once the overhead increases, the algorithm will switch to DISC strategy. Here, for each partition, DISC-all iteratively generates k-sorted databases to generate frequent k-sequences. After that, each customer sequence of the second-level partition is reassigned to another second-level partition by the next 2-minimum sequence. When all the second-level partitions under a first-level partition have been processed, each customer sequence of the

Table VIII. Database and First-Level Partitions of Table II

CID	Customer Sequences	Initial Partitions	After processing <*a*>-*partition*
1	<*(a)(e)*>	<(a)> - partition	<(e)> - partition
2	<*(fg)a(fbkc)*>	<(a)> - partition	<(b)> - partition
3	<*(ahcd)*>	<(a)> - partition	<(c)> - partition
4	<*a(abcd)e*>	<(a)> - partition	<(b)> - partition
5	<*e*>	<(e)> - partition	-

first-level partition is also reassigned to another first-level partition by the next minimum 1-sequence. Each partition is treated as a projected database and is mined for frequent 2- and 3-sequences in a way similar to SPADE; 4-sequences and longer ones are mined in DISC strategy.

For example, consider Table VIII with $\xi = 2$. All 1-sequences except *f*, *g*, *h*, and *k* are frequent, the first 4 sequences belong to <a>-partition because their minimum 1-sequence is <*a*>. In the second step and in depth-first, <a>-partition will be processed first to find all frequent sequences with <*a*> as prefix. Then, we find the next minimum 1-sequence in each of the five customer sequences and then reclassify them into the other first-level partitions, respectively. For instance, CID 1 and 2 are respectively reassigned into <(e)>-partition and <(b)>-partition, as shown in the last column of Table VIII. Now take the <a>-partition, a *counting array* is created similar to id-list in SPADE that contains support count for each 1-sequenec in <a> -partition (in the forms of (x) and (_x)) along with its position. Now sequences can be reduced by removing the infrequent sequences. The counting array is also used to find frequent 3-sequences in each second-level partition in one scan. After that, DISC is applied as above.

The DISC-all algorithm outperforms PrefixSpan for different kinds of databases. The improvement of the DISC-all algorithm grows as the database size increases. The reason why the PrefixSpan algorithm does not adapt to large databases is because the number of projections increases as the growth of database size. By contrast, the DISC-all algorithm can skip more infrequent sequences during direct sequence comparison because ξ increases with the growth of database size.

4.4. Hybrid Algorithms

Some algorithms combine several features that are characteristics of more than one of the three categories in the proposed taxonomy. For example, PLWAP [Lu and Ezeife 2003] combines tree projection and prefix growth features from pattern-growth category with a position-coded feature from the early-pruning category. All of these features are key characteristics of their respective categories, so we consider PLWAP as a pattern-growth/early-pruning hybrid algorithm. The ability of some algorithms to utilize a wide range of efficient features gives them an edge over other algorithms. It is also important to mention here that some features cannot be combined into one technique, such as, for example, “multiple scans of the database” and “tree projection;” since “tree projection” is used as an alternative in-memory database as does “support counting avoidance,” it cannot be combined with “multiple scans of the database”.

4.4.1. SPADE [Zaki 2001] – *Apriori-Based and Pattern-Growth Hybrid Miner.* SPADE is a major contribution to the literature, and is still considered one of the benchmark sequential pattern-mining algorithms. It relies on the *Lattice theory* [Davey and Priestley 1990] to generate candidate sequences. This idea is borrowed from the incremental sequential mining algorithm ISM introduced earlier by Parthasarathy [1999]. SPADE discovers sequences of subsets of items, not just single item sequences as is the case with Apriori [Agrawal et al. 1993]; it also discovers sequences with arbitrary time

Table IX. Variation of Table I as Sequence Database for SPADE

SID	TID , timestamp	Access Sequence
1	t_1, 10	<*bcbae*>
1	t_2, 15	<*bacbae*>
2	t_1, 15	<*abe*>
2	t_2, 25	<*abebd*>
3	t_1, 10	<*abad*>
3	t_2, 25	<*ade*>

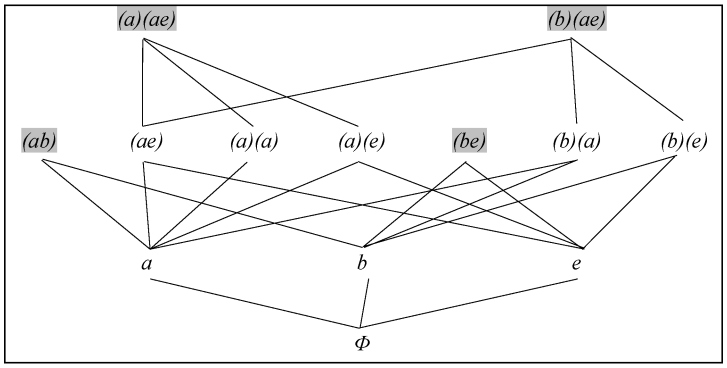


Fig. 12. Lattice induced by highlighted maximal frequent sequences.

gaps among items, and not just consecutive subsequences. To be able to show all the features of SPADE in our example, we will introduce a small change in the sequence database in Table I. This change is reflected in Table IX, where transactions are redistributed over user sequences with unique timestamps, and a new transaction is added also, namely, <(*ade*)>. With a *min_sup* of 3 and the property of subsequences, using the rule generation algorithm in Zaki [2001], frequent sequences are formulated as follows (maximal sequences are highlighted). Frequent 1-sequences are <*a*>, <*b*>, and <*e*>. Frequent 2-sequences are <(*ab*)>, <(*ae*)>, <(*a*)(*a*)>, <(*a*)(*e*)>, <(*be*)>, <(*b*)(*a*)>, and <(*b*)(*e*)>, while frequent 3-sequences are <(*a*)(*ae*)> and <(*b*)(*ae*)>. All happen to have a support count of 3.

SPADE relies on a lattice of frequent sequences generated by applying lattice theory on frequent sequences and their subsequences. Applying the lattice theory as a model on our example produces the lattice shown in Figure 12. SPADE works mainly in three steps, first finding frequent 1-sequences in an apriori-like way; and second, frequent 2-sequences. The third step traverses the lattice for support counting and enumeration of frequent sequences. The lattice can be traversed in either breadth-first or depth-first search. In both cases, *id-list* (a vertical layout of the sequence database) is produced for each atom and frequent sequence in the lattice by converting the horizontal database into vertical form. Depth-first search is considered better than breadth-first because it dramatically reduces computation and the number of id-lists generated. Pruning is also applied in the third step when intersecting id-lists (using temporal joins) and counting support to make sure that subsequences generated from the intersection are frequent.

One important problem in using the lattice theory, is that the lattice can grow very big and cannot fit in main memory, also making depth-first or breadth-first search very long. To solve this problem, Zaki [2001] proposed to partition the lattice into disjoint

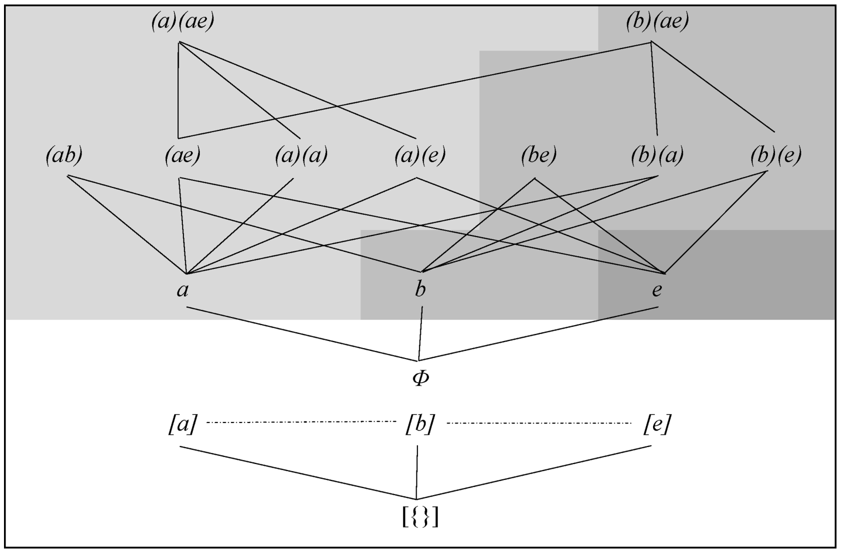
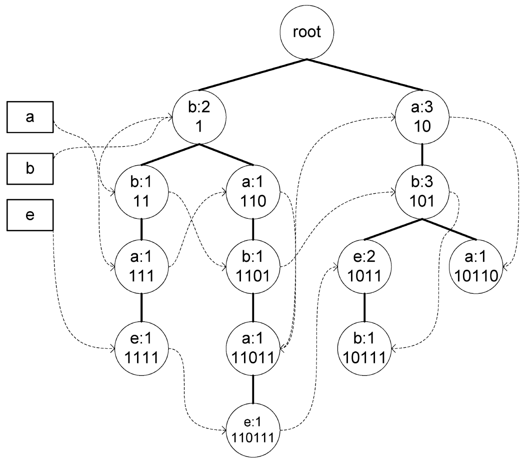


Fig. 13. Equivalence classes $\{[\mathrm{a}]_{\theta 1}, [\mathrm{b}]_{\theta 1}, [\mathrm{e}]_{\theta 1}\}$ induced by θ_k on S.

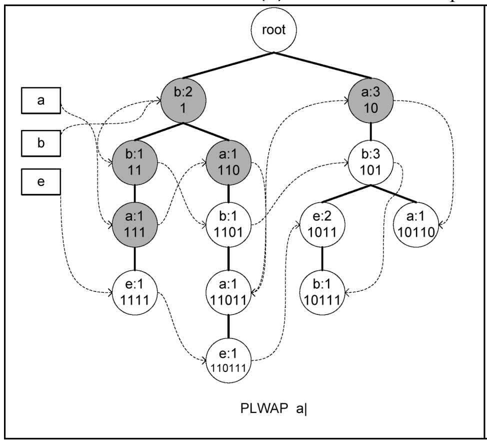
subsets called *equivalence classes* (defined next), where each subset can be loaded and searched separately in main memory. Define a function p: $(S, N) \rightarrow S$ where S is the set of sequences and N is the set of non-negative integers, and $p(X,k) = X[1{:}k]$. In other words, $p(X,k)$ returns the k-length prefix of X. Two sequences are considered to be in the same class if they share a common k-length prefix. This relation is called a *prefix-based* equivalence relation, denoted as θ_k [Zaki 1998]. A lattice induced by a θ_k equivalence relation is partitioned into equivalence classes as in our example in Figure 13.

The performance of SPADE can really be hindered and brought to the same level as that of GSP when mining only general but not maximal sequences.

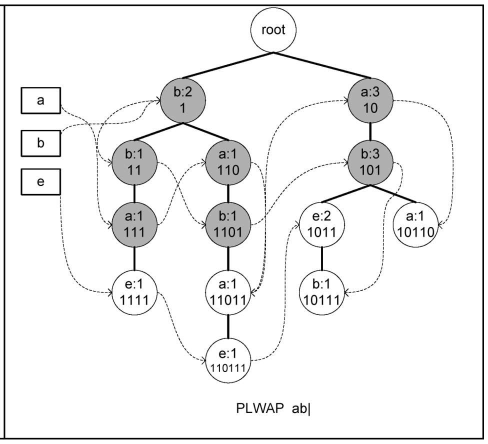
4.4.2. PLWAP [Ezeife and Lu 2005] *– Pattern-Growth and Early-Pruning Hybrid Miner.* PLWAP [Ezeife and Lu 2005] utilizes a binary code assignment algorithm to construct a preordered position-coded linked WAP-tree, where each node is assigned a binary code used during mining to determine which sequences are the suffix sequences of the last event and to find the next prefix for a mined suffix without having to reconstruct intermediate WAP-trees. As an example, consider the sequence database in Table I again, with *min_sup* = 3. Figure 14(a) shows the PLWAP-tree constructed from the WAP-tree in Figure 9(a), each node contains information of its support count and its binary position code. The position code of each node is assigned following the rule that the root has null position code, and the leftmost child (L) of any node (P) has a position code equivalent to appending "1" to the position code of its parent node (P), while the position code of any other child of parent node (P) is "0" appended to the position code of its nearest left sibling. During tree construction, each node is labeled as *node:count:position_code*, and during mining, descendant nodes of a parent node P on the same suffix tree are quickly identified such that the support count of P is used during mining without investigating the children node counts without having to construct intermediate conditional WAP-trees, as is the case in WAP-mine. Ezeife and Lu [2005] state that:



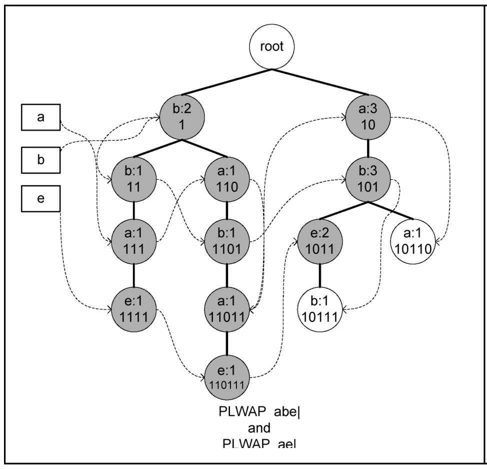
(a) PLWAP-tree of sequence DB from Table I



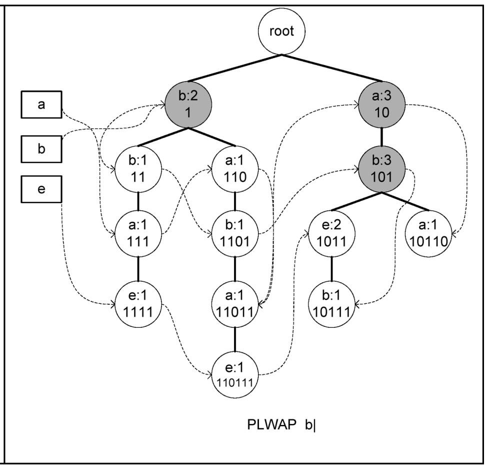
(b) *fs={a}*.



(c) *fs={a, ab}*.



(d) *fs={a, ab, abe, ae}*.



(e) *fs={a, ab, abe, ae, b}*.

Fig. 14

A node α is an occurrence of another node β if and only if the position code of α with "1" appended to its end, equals the first x number of bits in the position code of β, where x is (the number of bits in the position code of α) + 1.

This property is a byproduct of the binary code assignment process and enables PLWAP algorithm to identify on-the-fly which nodes fall within the suffix tree of P such that they are excluded from calculating support of P.

While building the PLWAP-tree (Figure 14(a)), a header linkage table is maintained for all frequent 1-sequences similar to WAP-tree. Each item in the table is also a header pointer for a queue of all the nodes in the tree that are occurrences of this frequent 1-sequence, these nodes are linked in an *event-node queue* e_i in preorder fashion. In WAP-tree the nodes are linked in the order they are inserted in the tree. The preorder linkage fashion is another key player in PLWAP that enables the mining process to traverse the tree depth-first for a conditional search on the prefix sequences and identify their suffix trees. To return to the example, to mine the PLWAP-tree in Figure 14(a), start from a in the header linkage table and follow its links to gather the support count for all its first occurrences of the suffix trees of the root, namely a:1:111, a:1:110, a:3:10 with total support count a(5), which is greater than *min_sup*, so it is added to the set of frequent sequences, *fs* = $\{a\}$. Now, recursively mine the suffix tree of a in Figure 14(b) for frequent 2-sequences that start with prefix a rooted at b:1:1101, b:3:101 with support b(4), so ab is added to the list of frequent sequences *fs* = {*a, ab*}. Now, mine the suffix tree of ab in Figure 14(c) for frequent 3-sequences that start with prefix ab, if roots a and b are tried respectively, and, recursively, no frequent sequences are found, the algorithm will backtrack to Figure 14(c) each time. Eventually, take the root e for the current subtree that has satisfactory support e:1:1111, e:1:110111, e:2:1011, resulting in *fs*={*a, ab, abe*}. Since there are no more frequent sequences after this point, the algorithm backtracks to Figure 14(b) with the suffix tree of a and now mines the suffix tree rooted at *ae* in Figure 14(d) resulting in *fs* = {*a, ab, abe, ae*}. There are no more frequent sequences in the suffix tree of a, so it backtracks to Figure 14(a) and tries b from the header linkage table for the suffix tree of b (Figure 14(e)). The algorithm keeps mining in the manner described for all 1-sequences in the header linkage table associated with their subtrees. The end result is the set of all frequent sequences *fs* = {*a, ab, abe, ae, b, bb, ba, be, e*}.

The WAP-tree uses conditional search in mining that is based on finding common suffix sequences first, while the PLWAP technique finds the common prefix sequences first by utilizing the binary code assignment and following the header table links in a depth-first way. Although memory space is needed to store the position code in PLWAP, it is still very small (an additional 4-byte integer with each node) compared to storing complete intermediate suffix trees as in WAP-mine. Another advantage of avoiding these intermediate trees and using binary code assignment is that mining time in PLWAP is cut to more than half of WAP-mine. Although CPU time dominates in both algorithms, WAP-mine spends more time on I/O than PLWAP does (5 to 13 times more) [Ezeife et al. 2005].

5. CONCLUSIONS AND FUTURE WORK

This article presents a taxonomy of sequential pattern-mining algorithms, and shows that current algorithms in the area can be classified into three main categories, namely, apriori-based, pattern-growth, and early-pruning with a fourth category as a hybrid of the main three. A thorough discussion of 13 characteristic features of the four categories of algorithms, with an investigation of the different methods and techniques, is presented. This investigation of sequential pattern-mining algorithms in the literature shows that the important heuristics employed include the following: using optimally sized data structure representations of the sequence database; early pruning of candidate sequences; mechanisms to reduce support counting; and maintaining a narrow search space. The quest for finding a reliable sequential pattern-mining algorithm should take these points into consideration. It is also noted that literature has recently introduced ways to minimize support counting, although some scholars claim they can avoid it [Yang and Kitsuregawa 2005; Wang and Han 2004; Chiu et al. 2004]. Minimizing support counting is strongly related to minimizing the search space.

In lights of the above, the following requirements should be considered for a reliable sequential pattern-mining algorithm. *First*, a method must generate a search space that is as small as possible. Features that allow this include early candidate sequence pruning and search space partitioning. Sampling of the database and lossy compression (i.e., concise representation) can also be used to generate a smaller search space. *Second*, it is important to narrow the search process within the search space. An algorithm can have a narrow search procedure such as depth-first search. *Third*, methods other than tree projection should be investigated for finding reliable sequential pattern-mining techniques. Ideas for future work in this direction include:

(1) Applying telescoping in tree projection pattern-growth algorithms to reduce the trie size, where more than one item can be compressed into one node or edge.

(2) Using the Fibonacci sequence to partition or sample the search space may be useful for effective mining of very long sequences.

(3) Distributed mining of sequences can provide a way to handle scalability in very large sequence databases and long sequences. In the area of web usage mining, this can be applied to mine several web logs distributed on multiple servers.

(4) Extending the capabilities of existing approaches. For example, modifying the S-Matrix to include support counts of candidate sequences that are only characteristic of the underlying application, extending the PLWAP algorithm to handle general multiitemset event sequence mining.

(5) Introducing object-orientedness in sequential pattern mining, to provide the flexibility of mining only focused parts of the database. Two scenarios may be considered: mining the sequence as a set of objects or having the mining process itself as an object-oriented process with the ability to perform different types of mining operations on data.

(6) Semantics inclusion. This entails having a model that can infer semantic relationships from domain knowledge obtained through an ontology or as annotations in the sequence and avoid support counting if a candidate sequence is semantically not likely to occur. A number of Ph.D. dissertations propose a web usage-mining framework which integrates usage, structure and content into the mining process for more directed constrained mining (e.g., Zheng [2004], Tanasa [2005], and Jin [2006]), but do not utilize domain knowledge in the mining process.

(7) Mathematical modeling of sequences to enable n-item prediction and sequence regeneration. This entails finding mathematical methods for defining functions to represent database sequences. This can be a polynomial or probabilistic model capable of regenerating the sequence with a minimal level of error. In this way it is possible to predict the future n-items of a given sequence.

ACKNOWLEDGMENTS

The authors would like to thank Dan Zhang for his help with the experimental implementation of the algorithms, and the anonymous reviewers for their productive criticisms.

REFERENCES

AGRAWAL, R., IMIELINSKI, T., AND SWAMI, A. 1993. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. ACM, New York, 207–216.

AGRAWAL, R. AND SRIKANT, R. 1994. Fast algorithms for mining association rules. In *Proceedings of the1994 International Conference on Very Large Data Bases* (*VLDB'94*). 487–499.

AGRAWAL, R. AND SRIKANT, R. 1995. Mining sequential patterns. In *Proceedings of the 11th Conference on Data Engineering (ICDE'95)*, 3–14.

ANTUNES, C. AND OLIVEIRA, A. L. 2004. Sequential pattern mining algorithms: Trade-offs between speed and memory. In *Proceedings of the Workshop on Mining Graphs, Trees and Sequences* (*MGTS-ECML/PKDD '04*).

AYRES, J., FLANNICK, J., GEHRKE, J., AND YIU, T. 2002. Sequential pattern mining using a bitmap representation. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 429–435.

BRIN, S., MOTWANI, R., ULLMAN, J.D., AND TSUR, S. 1997. Dynamic itemset counting and implication rules for market basket analysis. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data* (*SIGMOD'97*). ACM, New York, 255–264.

CHIU, D.-Y., WU, Y.-H., AND CHEN, A. L. P. 2004. An efficient algorithm for mining frequent sequences by a new strategy without support counting. In *Proceedings of the 20th International Conference on Data Engineering*. 375–386.

DAVE, B. A. AND PRIESTLEY, H. A. 1990. *Introduction to Lattices and Order*. Cambridge University Press.

DUNHAM, M. H. 2003. *Data Mining: Introductory and Advanced Topics*. Prentice Hall, Englewood Cliffs, NJ.

EL-SAYED, M., RUIZ, C., AND RUNDENSTEINER, E. A. 2004. FS-Miner: Efficient and incremental mining of frequent sequence patterns in web logs. In *Proceedings of the 6th Annual ACM International Workshop on Web Information and Data Management*. ACM, New York, 128–135.

EZEIFE, C. I. AND LU, Y. 2005. Mining web log sequential patterns with position coded pre-order linked WAP-tree. *Int. J. Data Mining Knowl. Discovery 10,* 5–38.

EZEIFE, C. I., LU, Y., AND LIU, Y. 2005. PLWAP sequential mining: Open source code. *In Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementation* (*SIGKDD*), ACM, New York, 26–35.

FACCA, F. M. AND LANZI, P. L. 2003. Recent developments in web usage mining research. In *Proceedings of the 5th International Conference on Data Warehousing and Knowledge Discovery. (DaWaK'03)*, Lecture Notes in Computer Science, Springer, Berlin.

FACCA, F. M. AND LANZI, P. L. 2005. Mining interesting knowledge from weblogs: A survey. *Data Knowl. Eng. 53,* 3 225–241.

GOETHALS, B. 2005. Frequent set mining. In *The Data Mining and Knowledge Discovery Handbook,* O. Maimon and L. Rokach Eds., Springer, Berlin, 377–397.

HAN, J. AND FU, Y. 1995. Discovery of multiple-level association rules from large databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'95)*. 420–431.

HAN, J., PEI, J., MORTAZAVI-ASL, B., CHEN, Q., DAYAL, U., AND HSU, M.-C. 2000. Freespan: Frequent pattern-projected sequential pattern mining. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, 355–359.

HAN, J., PEI, J., AND YIN, Y. 2000. Mining frequent patterns without candidate generation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD'00*), ACM, New York, 1–12.

HUANG, J.-W., TSENG, C.-Y., OU, J.-C., AND CHEN, M.-S. 2006. On progressive sequential pattern mining. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management.* ACM, New York, 850–851.

IVÁNCSY, R. AND VAJK, I. 2006. Frequent pattern mining in web log data. *Acta Polytech. Hungarica 3*, 1, 77–90.

JIN, X. 2006. *Task*-oriented modeling for the discovery of web user navigational patterns. Ph.D. dissertation, School of Computer Science. DePaul University, Chicago, IL.

LIU, B. 2007. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*. Springer, Berlin.

LU, Y. AND EZEIFE, C. I. 2003. Position coded pre-order linked WAP-tree for web log sequential pattern mining. In *Proceedings of the 7th Pacific-Asia Conference on Knowledge Discovery and Data Mining.* Lecture Notes in Computer Science, Springer, Berlin, 337–349.

MASSEGLIA, F., PONCELET, O., AND CICCHETTI, R. 1999. An efficient algorithm for web usage mining. *Network Inform. Syst. J. 2*, 571–603.

MASSEGLIA, F., TEISSEIRE, M., AND PONCELET, P. 2005. Sequential pattern mining: A survey on issues and approaches. In *Encyclopedia of Data Warehousing and Mining*,1–14.

NANDI, A. AND JAGADISH, H.V. 2007. Effective phrase prediction. In *Proceedings of the International Conference on Very Large Data Bases* (*VLDB'07*). 219–230.

NANOPOULOS, A. AND MANOLOPOULOS, Y. 2000. Finding generalized path patterns for web log data mining. In *Proceedings of the East-European Conference on Advances in Databases and Information Systems.* (Held jointly with the *International Conference on Database Systems for Advanced Applications: Current Issues in Databases and Information Systems*, 215–228.

PARK, J. S., CHEN, M. S., AND YU, P. S. 1995. An effective hash-based algorithm for mining association rules. In *Proceedings of the 1995 ACM-SIGMOD International Conference on Management of Data* (*SIGMOD'95*). ACM, New York, 175–186.

PARTHASARATHY, S., ZAKI, M.J., OGIHARA, M., AND DWARKADAS, S. 1999. Incremental and interactive sequence mining. In *Proceedings of the 8th International Conference on Information and Knowledge Management*. ACM, New York, 251–258.

PEI, J., HAN, J., MORTAZAVI-ASL, B., AND PINTO, H. 2001. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the International Conference on Data Engineering.* 215–224.

PEI, J., HAN, J., MORTAZAVI-ASL, B., AND ZHU, H. 2000. Mining access patterns efficiently from web logs. In *Knowledge Discovery and Data Mining. Current Issues and New Applications.* Lecture Notes Computer Science, vol. 1805, Springer, Berlin, 396–407.

RYMON, R. 1992. Search through systematic set enumeration. In *Proceedings of the 3rd International Conference. on the Principles of Knowledge Representation and Reasoning*. 539–550.

SAVASERE, A., OMIECINSKI, E., AND NAVATHE, S. 1995. An efficient algorithm for mining association rules in large databases. In *Proceedings of the International Conference on Very Large Data Bases* (*VLDB'95*). 432–443.

SONG, S., HU, H., AND JIN, S. 2005. HVSM: A new sequential pattern mining algorithm using bitmap representation. In *Advanced Data Mining and Applications.* Lecture Notes in Computer Science, vol. 3584, Springer, Berlin, 455–463.

SRIKANT, R. AND AGRAWAL, R. 1996. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology.* Leture Notes in Computer Science, vol. 1057, Springer, Berlin, 3–17.

SRIVASTAVA, J., COOLEY, R., DESHPANDE, M., AND TAN, P.-N. 2000. Web usage mining: Discovery and applications of usage patterns from Web data. *ACM SIGKDD Explorations Newsl. 1, 2*, 12–23.

TANASA, D. 2005. Web usage mining: contributions to intersites logs preprocessing and sequential pattern extraction with low support. Ph.D. dissertation, Université De Nice Sophia-Antipolis.

TOIVONEN, H. 1996. Sampling large databases for association rules. In *Proceedings of the International Conference on Very Large Data Bases* (*VLDB'95*). 134–145.

Wang, J. and Han, J. 2004. BIDE: Efficient mining of frequent closed sequences. In *Proceedings of the 20th International Conference on Data Engineering*. 79–90.

YANG, Z. AND KITSUREGAWA, M. 2005. LAPIN-SPAM: An improved algorithm for mining sequential pattern. In *Proceedings of the 21st International Conference on Data Engineering Workshops*. 1222.

YANG, Z., WANG, Y., AND KITSUREGAWA, M. 2005. LAPIN: Effective sequential pattern mining algorithms by last position induction. Tech. rep., Tokyo University. http://www.tkl.iis.u-tokyo.ac.jp/~yangzl/Document/LAPIN.pdf.

YANG, Z., WANG, Y., AND KITSUREGAWA, M. 2006. An effective system for mining web log. In *Proceedings of the 8th Asia-Pacific Web Conference* (*APWeb'06*), 40–52.

YANG, Z., WANG, Y., AND KITSUREGAWA, M. 2007. LAPIN: Effective sequential pattern mining algorithms by last position induction for dense databases. In *Advances in Databases: Concepts, Systems and Applications*. Lecture Notes in Computer Science, vol. 4443, 1020–1023.

ZAKI, M. J. 1998. Efficient enumeration of frequent sequences. In *Proceedings of the 7th International Conference on Information and Knowledge Management.* 68–75.

ZAKI, M. J. 2000. Scalable algorithms for association mining. *IEEE Trans. Knowl. Data Eng. 12,* 3, 372–390.

ZAKI, M. J. 2001. SPADE: An efficient algorithm for mining frequent sequences. *Mach. Learn. 42*, 31–60.

ZHENG, T. 2004. WebFrame: In pursuit of computationally and cognitively efficient web mining. Ph.D. dissertation, Department of Computing Science. University of Alberta, Edmonton.

ZILL, D. J. 1998. *Calculus with Analytic Geometry* 2nd ed. PWS-KENT.

Received April 2008; revised December 2008; accepted February 2009