A Breadth-First Algorithm for Mining Frequent Patterns from Event Logs *

Risto Vaarandi

Department of Computer Engineering, Tallinn University of Technology, Estonia
risto.vaarandi@eyp.ee

Abstract. Today, event logs contain vast amounts of data that can easily overwhelm a human. Therefore, the mining of frequent patterns from event logs is an important system and network management task. This paper discusses the properties of event log data, analyses the suitability of popular mining algorithms for processing event log data, and proposes an efficient algorithm for mining frequent patterns from event logs.

1 Introduction

Event logging and log files are playing an increasingly important role in system and network administration (e.g., see [1]), and the mining of frequent patterns from event logs is an important system and network management task [2][3][4][5][6][7]. Recently proposed mining algorithms have often been variants of the Apriori algorithm [2][3][4][7], and they have been mainly designed for detecting frequent event type patterns [2][3][4][5][7]. The algorithms assume that each event from the event log has two attributes – time of event occurrence and event type. There are several ways for defining the *frequent event type pattern*, with two definitions being most common. In the case of the first definition (e.g., see [7]), the algorithm views the event log as a set of overlapping *windows*, where each window contains events from a time frame of t seconds (the *window size* t is given by the user). A certain combination of event types is considered a frequent pattern if this combination is present at least in s windows, where the threshold s is specified by the user. In the case of the second definition (e.g., see [5]), the algorithm assumes that the event log has been divided into non-overlapping slices according to some criteria (e.g., events from the same slice were all issued by the same host). A certain combination of event types is considered a frequent pattern if this combination is present at least in s slices (the threshold s is given by the user). Although the use of this definition requires more elaborate preprocessing of the event log, it also eliminates the noise that could appear when events from different slices are mixed. In the rest of this paper, we will employ the second approach for defining the frequent event type pattern.

Events in windows or slices are usually ordered in occurrence time ascending order. The order of events in windows or slices is often taken into account

* This work is supported by the Union Bank of Estonia and partly sponsored by the Estonian Science Foundation under the grant 5766.

during the mining, since this could reveal causal relations between event types – e.g., instead of an unordered set {*DeviceDown*, *FanFailure*} the algorithm outputs a sequence $FanFailure \rightarrow DeviceDown$. However, as pointed out in [7], the mining of unordered frequent event type sets is equally important. Due to network latencies, events from remote nodes might arrive and be written to the log in the order that differs from their actual occurrence order. Even if events are timestamped by the sender, system clocks of network nodes are not always synchronized, making it impossible to restore the original order of events. Also, in many cases the occurrence order of events from the same window or slice is not pre-determined (e.g., since events are not causally related). In the remainder of this paper, we will not consider the order of events in a slice important.

Note that it is often difficult to mine patterns of event types from raw event logs, since messages in raw event logs rarely contain explicit event type codes (e.g., see [1]). Fortunately, it is possible to derive event types from event log lines, since very often the events of the same type correspond to a certain line pattern. For example, the lines *Router myrouter1 interface 192.168.13.1 down*, *Router myrouter2 interface 10.10.10.12 down*, and *Router myrouter5 interface 192.168.22.5 down* represent the event type "Router interface down", and correspond to the line pattern *Router * interface * down*. Thus, the mining of frequent line patterns is an important preprocessing technique, but can be very useful for other purposes as well, e.g., for building event log models [8].

Let $I = \{i_1, ..., i_n\}$ be a set of items. If $X \subseteq I$, X is called an *itemset*, and if $|X| = k$, X is also called a *k-itemset*. A *transaction* is a tuple $T = (tid, X)$ where *tid* is a transaction identifier and X is an itemset. A *transaction database* D is a set of transactions, and the *cover* of an itemset X is the set of identifiers of transactions that contain X: $cover(X) = \{tid \mid (tid, Y) \in D, X \subseteq Y\}$. The *support* of an itemset X is defined as the number of elements in its cover: $supp(X) = |cover(X)|$. The task of *mining frequent itemsets* is formulated as follows – given the transaction database D and the *support threshold* s, find all itemsets with the support s or higher (each such set is called a *frequent itemset*).

When the event log has been divided into m slices (numbered from 1 to m), then we can view the set of all possible event types as the set of items I, and each slice can be considered a transaction with its *tid* between 1 and m. If the ith slice is $S_i = \{E_1, ..., E_k\}$, where $E_j = (t_j, e_j)$ is an event from S_i, e_j is the type of E_j, and t_j is the occurrence time of E_j, then the transaction corresponding to S_i is $(i, \cup_{j=1}^{k} \{e_j\})$. When we inspect a raw event log at the word level, each line pattern consists of fixed words and wildcards, e.g., *Router * interface * down*. Note that instead of considering entire such pattern we can just consider the fixed words together with their positions, e.g., {(*Router*, 1), (*interface*, 3), (*down*, 5)} [8]. Similarly, if the ith line from a raw event log has k words, it can be viewed as a transaction with identifier i and k word-position pairs as items.

If we view event logs as transaction databases in the ways described above, we can formulate the task of mining frequent event type patterns or frequent line patterns as the task of mining frequent itemsets. We will use this formulation in the rest of this paper, and also use the term *pattern* to denote an itemset.

In this paper, we propose an efficient algorithm for mining frequent patterns from event logs that can be employed for mining line and event type patterns. The rest of the paper is organized as follows: section 2 discusses related work on frequent itemset mining, section 3 presents the properties of event log data and the analysis of existing mining algorithms, section 4 describes a novel algorithm for mining frequent patterns from event logs, section 5 discusses the performance and implementation of the algorithm, and section 6 concludes the paper.

2 Frequent itemset mining

The frequent itemset mining problem has received a lot of attention during the past decade, and a number of mining algorithms have been developed. For the sake of efficient implementation, most algorithms order the items according to certain criteria, and use this ordering for representing itemsets. In the rest of this paper, we assume that if $X = \{x_1, ..., x_k\}$ is an itemset, then $x_1 < ... < x_k$.

The first algorithm developed for mining frequent itemsets was Apriori [9] which works in a breadth-first manner – discovered frequent k-itemsets are used to form candidate k+1-itemsets, and frequent k+1-itemsets are found from the set of candidates. Recently, an efficient *trie* (prefix tree) data structure has been proposed for the candidate support counting [10][11]. Each edge in the *itemset trie* is labeled with the name of a certain item, and when the Apriori algorithm terminates, non-root nodes of the trie represent all frequent itemsets. If the path from the root node to a non-root node N is $x_1, ..., x_k$, N identifies the frequent itemset $X = \{x_1, ..., x_k\}$ and contains a counter that equals to $supp(X)$. In the remainder of this paper, we will use notations $node(x_1, ..., x_k)$ and $node(X)$ for N, and also, we will always use the term *path* to denote a path that starts from the root node. Figure 1 depicts a sample transaction database and an itemset trie (the support threshold is 2 and items are ordered in lexicographic order).

Transaction ID	Itemset
1	*abcde*
2	*abc*
3	*bcd*
4	*abc*
5	*ab*

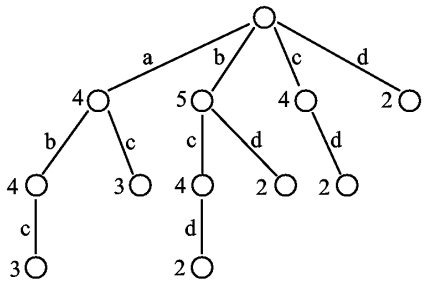

Fig. 1. A sample transaction database and an itemset trie.

As its first step, the Apriori algorithm detects frequent 1-itemsets and creates nodes for them. Since every subset of a frequent itemset must also be frequent,

the nodes for candidate k+1-itemsets are generated as follows – for each node $node(x_1, ..., x_k)$ at depth k all its siblings will be inspected. If $x_k < y_k$ for the sibling $node(x_1, ..., x_{k-1}, y_k)$, then the candidate node $node(x_1, ..., x_k, y_k)$ will be inserted into the trie with its counter set to zero. In order to find frequent k+1-itemsets, the algorithm traverses the itemset trie for each transaction $(tid, Y) \in D$, and increments the counter in $node(X)$ if $X \subseteq Y, |X| = k + 1$. After the database pass, the algorithm removes nodes for infrequent candidate itemsets.

Although the Apriori algorithm works well when frequent itemsets contain relatively few items (e.g., 4–5), its performance starts to deteriorate when the size of frequent itemsets increases [12][13]. In order to produce a frequent itemset $\{x_1, ..., x_k\}$, the algorithm must first produce its $2^k - 2$ subsets that are also frequent, and when the database contains frequent k-itemsets for larger values of k (e.g., 30–40), the number of nodes in the itemset trie could be very large. As a result, the runtime cost of the repeated traversal of the trie will be prohibitive, and the trie will consume large amounts of memory.

In recent past, several algorithms have been proposed that explore the search space in a depth-first manner, and that are reportedly by an order of a magnitude faster than Apriori. The most prominent depth-first algorithms for mining frequent itemsets are Eclat [12] and FP-growth [13]. An important assumption made by Eclat and FP-growth is that the transaction database fits into main memory. At each step of the depth-first search, the algorithms are looking for frequent k-itemsets $\{p_1, ..., p_{k-1}, x\}$, where the prefix $P = \{p_1, ..., p_{k-1}\}$ is a previously detected frequent k-1-itemset. When looking for these itemsets, the algorithms extract from the database the data describing transactions that contain the itemset P, and search only this part of the database. If frequent k-itemsets were found, one such itemset is chosen for the prefix of the next step, otherwise the new prefix is found by backtracking. Since the database is kept in main memory using data structures that facilitate the fast extraction of data, Eclat and FP-growth can explore the search space faster than Apriori.

The main difference between the Eclat and FP-growth algorithm is how the transaction database is stored in memory. Eclat keeps item covers in memory, while FP-growth saves all transactions into *FP-tree* which is a tree-like data structure. Each non-root node of the FP-tree contains a counter and is labeled with the name of a certain frequent item (frequent 1-itemset). In order to build the FP-tree, the FP-growth algorithm first detects frequent items and orders them in support ascending order. Frequent items of each transaction are then saved into FP-tree in *reverse* order as a path, by incrementing counters in existing nodes of the path and creating missing nodes with counters set to 1. In that way, nodes closer to the root node correspond to more frequent items, and are more likely to be shared by many transactions, yielding a smaller FP-tree [13].

Unfortunately, Eclat and FP-growth can't be employed for larger transaction databases which don't fit into main memory. Although some techniques have been proposed for solving this problem (e.g., the partitioning of the database), these techniques are often infeasible [14]. In the next section we will show that this problem is also relevant for event log data sets.

3 The properties of event log data

The nature of data in the transaction database plays an important role when designing an efficient mining algorithm. When conducting experiments with event log data sets, we discovered that they have several important properties. Table 1 presents eight sample data sets that we used during our experiments. The first five data sets (named *openview*, *mailserver*, *fileserver*, *webserver*, and *ibankserver*, respectively) are raw event logs from different domains: HP OpenView event log file, mail server log file, file and print server log file, web server log file, and Internet banking server log file. We used these event logs for frequent line pattern mining experiments. The rest of the data sets (named *websess*, *ibanksess*, and *snort*) were obtained from raw event logs by arranging events into slices, and we used them during our experiments of mining frequent event type patterns. In *websess* data set each slice reflects a user visit to the web server, with event types corresponding to accessed URLs. In *ibanksess* data set a slice corresponds to a user session in the Internet bank, where each event type is a certain banking transaction type. The *snort* data set was obtained from the Snort IDS alert log, and each slice reflects an attack from a certain IP address against a certain server, with event types corresponding to Snort rule IDs.

Table 1. The properties of event log data

Data set name	# of transactions	# of items	Items that occur ten times or less	Items that occur at least once per 1,000 transactions	Max. frequent itemset size (supp. 0.1%)
openview	1,835,679	1,739,185	1,582,970	1,242	65
mailserver	7,657,148	1,700,840	1,472,296	627	15
fileserver	7,935,958	11,893,846	11,716,395	817	118
webserver	16,252,925	4,273,082	3,421,834	396	24
ibankserver	14,733,696	2,008,418	1,419,138	304	11
websess	217,027	22,544	17,673	341	21
ibanksess	689,885	454	140	110	12
snort	95,044	554	476	45	7

Firstly, it is evident from Table 1 that the number of items in the transaction database can be quite large, especially when we mine frequent line patterns from raw event logs. However, only few items are relatively frequent (occur at least once per 1,000 transactions), and also, most items appear only few times in the data set. Secondly, Table 1 also indicates that frequent itemsets may contain many items (the table presents figures for the support threshold of 0.1%), which means that Apriori is not always adequate for processing event log data.

The third important property of event log data is that there are often strong correlations between frequent items in transactions. If items are event types, such strong correlations often exist because of causal relations between event types (e.g., when the *PortScan* event appears, the *TrafficAnomaly* event also

appears), or because of distinct patterns in the user behavior (e.g., if the web page A is accessed, the web page B is also accessed). In the case of raw event logs where items are word-position pairs, this effect is usually caused by the message formatting with a certain format string before the message is logged, e.g., *sprintf(message, "Connection from %s port %d", ip, port)*. When events of the same type are logged many times, constant parts of the format string will become frequent items which occur together many times in the data set. There could also be strong correlations between items corresponding to variable parts of the format string, e.g., between user names and workstation IP addresses.

In order to assess how well the Apriori, Eclat, and FP-growth algorithms are suited for mining frequent patterns from event logs, we conducted several experiments on data sets from Table 1 with support thresholds of 1% and 0.1% (during all our experiments presented in this paper, we used Apriori, Eclat, and FP-growth implementations by Bart Goethals [15]). In order to reduce the memory consumption of the algorithms, we removed very infrequent items (with the support below 0.01%) from all data sets, and as a result, the number of items was below 7,000 in all cases. A Linux workstation with 1.5 GHz Pentium 4 processor, 512 MB of main memory, and 1 GB of swap space was used during the experiments. Our experiments revealed that when the transaction database is larger, depth-first algorithms could face difficulties when they attempt to load it into main memory (see Table 2).

Table 2. The size of the memory-resident database

Data set name	Eclat 1%	FP-growth 1%	Eclat 0.1%	FP-growth 0.1%
openview	359.2 MB	2.8 MB	370.7 MB	5.7 MB
mailserver	263.3 MB	2.9 MB	280.5 MB	10.8 MB
fileserver	1009.1 MB	4.0 MB	1024.4 MB	8.6 MB
webserver	Out of VM	64.0 MB	Out of VM	249.9 MB
ibankserver	657.5 MB	37.2 MB	678.0 MB	77.5 MB
websess	5.7 MB	2.4 MB	6.0 MB	9.8 MB
ibanksess	17.5 MB	3.3 MB	17.6 MB	10.6 MB
snort	2.9 MB	2.2 MB	2.9 MB	2.2 MB

Table 2 suggests that Eclat is unsuitable for mining frequent patterns from larger event logs, even when infrequent items have been filtered out previously and the algorithm has to load only few thousand item covers into memory. With *fileserver* and *ibankserver* data sets the Eclat algorithm did run out of physical memory, and was able to continue only because of sufficient swap space; with *webserver* data set, the algorithm terminated abnormally after consuming all available virtual memory. Based on these findings, we removed Eclat from further testing. Table 2 also suggests that the FP-growth algorithm is more convenient in terms of memory consumption. The reason for this is that the FP-tree data structure is efficient for storing transactions when strong correlations

exist between frequent items in transactions. If many such correlations exist, the number of different frequent item combinations in transactions is generally quite small, and consequently relatively few different paths will be saved to FP-tree. However, it should be noted that for larger data sets the FP-tree could nevertheless be rather large, especially when the support threshold is lower (e.g., for the *webserver* data set the FP-tree consumed about 250 MB of memory when the support threshold was set to 0.1%).

We also tested the Apriori algorithm and verified that in terms of performance it is inferior to FP-growth – for example, when the support threshold was set to 1%, Apriori was 11.5 times slower on *mailserver* data set, and 9.5 times slower on *ibankserver* data set. However, on *openview* and *fileserver* data sets (which contain frequent itemsets with a large number of items) both algorithms performed poorly, and were unable to complete within 24 hours.

The experiment results indicate that all tested algorithms are not entirely suitable for mining frequent patterns from event logs. In the next section we will present an efficient mining algorithm that attempts to address the shortcomings of existing algorithms.

4 A frequent pattern mining algorithm for event logs

In this section we will present an efficient algorithm for mining frequent patterns from event logs. It combines the features of previously discussed algorithms, taking also into account the properties of event log data. Since depth-first Eclat and FP-growth algorithms are inherently dependent on the amount of main memory, our algorithm works in a breadth-first manner and employs the itemset trie data structure (see section 2). In order to avoid inherent weaknesses of Apriori, the algorithm uses several techniques for speeding up its work and reducing its memory consumption. These techniques are described in the following subsections.

4.1 Mining frequent items

The mining of frequent items is the first step of any breadth-first algorithm which creates a base for further mining. In order to detect frequent items, the algorithm must make a pass over the data set and count how many times each item occurs in the data set, keeping item counters in main memory. Unfortunately, because the number of items can be very large (see section 3), the memory cost of the item counting is often quite high [8].

In order to solve this problem, our algorithm first estimates which items *need not* to be counted. Before the counting, the algorithm makes an extra pass over the data set and builds the *item summary vector*. The item summary vector is made up of m counters (numbered from 0 to m-1) with each counter initialized to zero. During the pass over the data set, a fast hashing function is applied to each item. The function returns integer values from 0 to m-1, and each time the value i is calculated for an item, the ith counter in the vector will be incremented. Since efficient hashing functions are uniform [16], each counter in the vector will

correspond roughly to n/m items, where n is the number of different items in the data set. If items $i_1, ..., i_k$ are all items that hash to the value i, and the items $i_1, ..., i_k$ occur $t_1, ..., t_k$ times, respectively, then the value of the ith counter in the vector equals to the sum $\sum_{j=1}^{k} t_j$.

After the summary vector has been constructed, the algorithm starts counting the items, ignoring the items for which counter values in the summary vector are below the support threshold (no such item can be frequent, since its support does not exceed its counter value). Since most items appear only few times in the data set (see section 3), many counter values will never cross the support threshold. Experiment results presented in [8] indicate that even the use of a relatively small vector (e.g., 100 KB) dramatically reduces the memory cost of the item counting.

4.2 Cache tree

Eclat and FP-growth algorithms are fast not only because of their depth-first search strategy, but also because they load the transaction database from disk (or other secondary storage device) into main memory. In addition, the algorithms don't attempt to store each transaction as a separate record in memory, but rather employ efficient data structures that facilitate data compression (e.g., the FP-tree). As a result, the memory-resident database is much smaller than the original database, and a scan of the database will take much less time.

Although recent Apriori implementations have employed a prefix tree for keeping the database in memory [10][11], this technique can't be used for data sets which don't fit into main memory. As a solution, we propose to store most frequently used transaction data in the *cache tree*. Let D be the transaction database and F the set of all frequent items. We say that a set of frequent items $X \subseteq F$ *corresponds to m transactions* if $|\{(tid, Y) \mid (tid, Y) \in D, Y \cap F = X\}| = m$. Cache tree is a memory-resident tree data structure which is guaranteed to contain all sets of frequent items that correspond to c or more transactions, where the value of c is given by the user. Each edge in the cache tree is labeled with the name of a certain frequent item, and each node contains a counter. If the set of frequent items $X = \{x_1, ..., x_k\}$ corresponds to m transactions and is stored to the cache tree, it will be saved as a path $x_1, ..., x_k$, and the counter in the tree node $node(x_1, ..., x_k)$ will be set to m. This representation of data allows the algorithm to speed up its work by a considerable margin, since instead of processing m transactions from disk (or other secondary storage device), it has to process just one memory-resident itemset X that does not contain infrequent (and thus irrelevant) items.

In order to create the cache tree, the algorithm has to detect which sets of frequent items correspond to at least c transactions. Note that if the algorithm simply counts the occurrence times of sets, all sets would end up being in main memory together with their occurrence counters (as if $c = 0$). For solving this problem, the algorithm uses the summary vector technique presented in section 4.1 – for each transaction $(tid, Y) \in D$ it finds the set $X = Y \cap F$, hashes X to an integer value, and increments the corresponding counter in the *transaction*

summary vector. After the summary vector has been constructed, the algorithm makes another pass over the data, finds the set X for each transaction, and calculates the hash value for it. If the hash value is i and the ith counter in the vector is smaller than c, the itemset X is saved to the *out-of-cache* file as a separate record, otherwise the itemset X is saved into the cache tree (the counter in $node(X)$ is incremented, or set to 1 if the node didn't exist previously).

In that way, the transaction data that would be most frequently used during the mining are guaranteed to be in main memory, and the representation of this data allows the algorithm to further speed up its work. On the other hand, the algorithm does not depend on the amount of main memory available, since the amount of data stored in the cache tree is controlled by the user.

4.3 Reducing the size of the itemset trie

As discussed in section 2, the Apriori algorithm is not well suited for processing data sets which contain frequent k-itemsets for larger values of k, since the itemset trie could become very large, making the runtime and memory cost of the algorithm prohibitive. In order to narrow the search space of mining algorithms, several recent papers have proposed the mining of closed frequent itemsets only. An itemset X is *closed* if X has no superset with the same support. Although it is possible to derive all frequent itemsets from closed frequent itemsets, this task has a quadratic time complexity.

In this subsection, we will present a technique for reducing the size of the itemset trie, so that the trie would still represent all frequent itemsets. When there are many strong correlations between frequent items in transactions, many parts of the Apriori itemset trie are likely to contain information that is already present in other parts. The proposed reduction technique will enable the algorithm to develop only those trie branches that contain unique information.

Let $F = \{f_1, ..., f_n\}$ be the set of all frequent items. We call the set $dep(f_i) = \{f_j \mid f_i \neq f_j, cover(\{f_i\}) \subseteq cover(\{f_j\})\}$ the *dependency set* of f_i, and say that an item f_i *has m dependencies* if $|dep(f_i)| = m$. A *dependency prefix of the item* f_i is the set $pr(f_i) = \{f_j \mid f_j \in dep(f_i), f_j < f_i\}$. A *dependency prefix of the itemset* $\{f_{i_1}, ..., f_{i_k}\}$ is the set $pr(\{f_{i_1}, ..., f_{i_k}\}) = \cup_{j=1}^{k} pr(f_{i_j})$.

Note that the dependency prefix of the itemset has two important properties: (1) if $pr(\{f_{i_1}, ..., f_{i_k}\}) \subseteq \{f_{i_1}, ..., f_{i_k}\}$, then $pr(\{f_{i_1}, ..., f_{i_{k-1}}\}) \subseteq \{f_{i_1}, ..., f_{i_{k-1}}\}$, (2) if $pr(X) \subseteq X$, then $supp(X \setminus pr(X)) = supp(X)$ (this follows from the transitivity property – if $a, b, c \in F$, $a \in pr(b)$, and $b \in pr(c)$, then $a \in pr(c)$).

The technique for reducing the size of the itemset trie can be summarized as follows – *if the itemset does not contain its dependency prefix, don't create a node in the trie for that itemset*. As its first step, the algorithm creates the root node, detects frequent items, and finds their dependency sets. If no frequent items were found, the algorithm terminates. Otherwise, it creates nodes only for these frequent items which have empty dependency prefixes, attaching the nodes to the root node. From then on, the algorithm will build the trie layer by layer. If the current depth of the trie is k, the next layer of nodes is created by processing the itemsets previously saved to the cache tree and the out-of-cache file. If the

itemset $\{x_1, ..., x_m\}$ was read from the cache tree, i is set to the counter value from the cache tree node $node(\{x_1, ..., x_m\})$, otherwise i is set to 1. Then the itemset is processed by traversing the itemset trie recursively, starting from the root node (in the rest of this paper, this procedure is called *ProcItemset*):

1. If the current node is at depth d, $d < k$, let $l := k + 1 - d$. If $m < l$, return; otherwise, if there is an edge with the label x_1 from the current node, follow that edge and process the itemset $\{x_2, ..., x_m\}$ recursively for the new node.
2. If the current node is at depth k, the path leading to the current node is $y_1, ..., y_k$, and $pr(x_1) \subseteq \{y_1, ..., y_k\}$, check whether there is an edge with the label x_1 from the current node to a candidate node. If the edge exists, add i to the counter in the candidate node; if the edge does not exist, create the candidate node $node(y_1, ..., y_k, x_1)$ in the trie with the counter value i.
3. If $m > 1$, process the itemset $\{x_2, ..., x_m\}$ recursively for the current node.

After the data pass, the algorithm removes candidate nodes with counter values below the support threshold, and terminates if all candidate nodes were removed.

When the algorithm has completed its work, each non-root node in the trie represents a frequent itemset which contains its dependency prefix, and also, if X is a frequent itemset which contains its dependency prefix, the node $node(X)$ is present in the trie (this follows from the first property of the itemset dependency prefix). Although the trie is often much smaller than the Apriori itemset trie, all frequent itemsets can be easily derived from its nodes. For $node(X)$ that represents the frequent itemset X, derived itemsets are $\{X \setminus Y \mid Y \subseteq pr(X)\}$, with each itemset having the same support as X (this follows from the second property of the itemset dependency prefix). Also, if V is a frequent itemset, there exists a unique node $node(W)$ in the trie for deriving V, where $W = V \cup pr(V)$.

The algorithm can be optimized in several ways. The first optimization concerns the frequent item ordering. For the Apriori algorithm, a popular choice has been to order items in support ascending order [10][11]. We propose to order frequent items in dependency ascending order, i.e., in the order that satisfies the following condition – if $f_i < f_j$, then $|dep(\{f_i\})| \leq |dep(\{f_j\})|$. This ordering increases the likelihood that the dependency prefix of an item contains all elements from the dependency set of the item, and thus increases the effectiveness of the trie reduction technique. The second optimization comes from the observation that when frequent items have very few dependencies, our algorithm could produce much more candidate nodes than Apriori. Fortunately, candidates can still be generated within our framework in Apriori fashion – if the trie reduction technique was not applied at node N for reducing the number of its child nodes, and node M is a child of N, then the siblings of M contain all necessary nodes for the creation of candidate child nodes for M. After we have augmented our algorithm with the Apriori-like candidate generation, its final version can be summarized as follows:

1. Make a pass over the database, detect frequent items, and order them in lexicographic order (if the number of items is very large, an optional pass described in section 4.1 can be made for filtering out irrelevant items). If no frequent items were found, terminate.

2. Make a pass over the database, in order to calculate dependency sets for frequent items and to build the transaction summary vector.
3. Reorder frequent items in dependency ascending order and find their dependency prefixes.
4. Make a pass over the database, in order to create the cache tree and the out-of-cache file.
5. Create the root node of the itemset trie and attach nodes for frequent items with empty dependency prefixes to the root node. If all frequent items have empty dependency prefixes, set the APR-flag in the root node.
6. Let $k := 1$.
7. Check all nodes in the trie at depth k. If the parent of a node N has the APR-flag set, generate candidate child nodes for the node N in Apriori fashion (node counters are set to zero), and set the APR-flag in the node N.
8. Build the next layer of nodes in the trie using the *ProcItemset* procedure with the following modification – if the APR-flag is set in a node at depth k, don't attach any additional candidate nodes to that node.
9. Remove the candidate nodes (nodes at depth $k+1$) with counter values below the support threshold. If all candidate nodes were removed, output frequent itemsets and terminate.
10. Find the nodes at depth k for which the trie reduction technique was not applied during step 8 (during calls to the *ProcItemset* procedure) for reducing the number of their child nodes, and set the APR-flag in these nodes. Then let $k := k + 1$ and go to step 7.

It is easy to see that the Apriori algorithm is a special case of our algorithm – if frequent items have no dependencies at all, our algorithm is identical to Apriori. Otherwise, the algorithm will employ the trie reduction technique as much as possible, avoiding to develop trie branches that would contain redundant information. If the trie reduction technique is not applicable for certain branches any more, the algorithm will switch to Apriori-like behavior for these branches.

Figure 2 depicts a sample reduced itemset trie for the same transaction database as presented in Fig. 1 (the support threshold is 2). The reduced itemset trie in Fig. 2 is obtained as follows – first the set of frequent items is found, yielding $F = \{a, b, c, d\}$. Then dependency sets are calculated for frequent items: $dep(d) = \{b, c\}, dep(c) = dep(a) = \{b\}, dep(b) = \emptyset$. After ordering frequent items in dependency ascending order $b < c < a < d$, their dependency prefixes are: $pr(b) = \emptyset, pr(c) = pr(a) = \{b\}, pr(d) = \{b, c\}$. Only the node $node(b)$ can be attached to the root node, since b is the only item with an empty dependency prefix. Also, although the itemset $\{b, d\}$ is frequent, the node $node(b, d)$ will not be inserted into the trie, since the set $\{b\}$ does not contain the item c from the dependency prefix of d. Altogether, there are 11 frequent itemsets – the node $node(b)$ represents one itemset $\{b\}$ with support 5, the node $node(b, c)$ represents two itemsets $\{b, c\}$ and $\{c\}$ with support 4, the node $node(b, c, a)$ represents two itemsets $\{b, c, a\}$ and $\{c, a\}$ with support 3 (but does not represent $\{a\}$, since $\{b, c, a\} \setminus pr(\{b, c, a\}) = \{c, a\}$!), the node $node(b, c, d)$ represents four itemsets $\{b, c, d\}$, $\{b, d\}$, $\{c, d\}$, and $\{d\}$ with support 2, and the node $node(b, a)$ represents two itemsets $\{b, a\}$ and $\{a\}$ with support 4.

Transaction ID	Itemset
1	*abcde*
2	*abc*
3	*bcd*
4	*abc*
5	*ab*

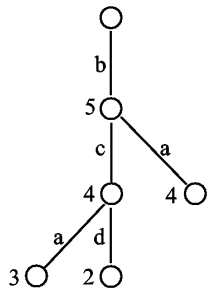

Fig. 2. A sample transaction database and a reduced itemset trie.

In the next section we will discuss the performance and implementation issues of our algorithm.

5 Performance and implementation

In order to measure the performance of our algorithm, we decided to compare it with the FP-growth algorithm, because experiment results presented in section 3 suggest that FP-growth is much better suited for mining patterns from event logs than Eclat and Apriori. Also, we wanted to verify whether our algorithm that uses the breadth-first approach is able to compete with a fast depth-first algorithm. We conducted our experiments on a Linux workstation with 1.5 GHz Pentium 4 processor and 512 MB of memory. For the sake of fair performance comparison, we configured our algorithm to load the entire transaction database into main memory for all data sets. The results of our experiments are presented in Table 3, Table 4, and Table 5.

First, the experiment results indicate that the trie reduction technique is rather effective for event log data sets, and often significantly smaller itemset trie is produced than in the case of Apriori (if there are m frequent itemsets, the number of nodes in the Apriori itemset trie is m+1). As a result, the algorithm consumes much less memory and is much faster than Apriori, since the repeated traversal of a smaller trie takes much less time. The results also indicate that our algorithm performs quite well when compared to FP-growth, and outperforms it in several cases. The only exceptions are *webserver* data set, and *websess* data set for the 0.1% support threshold. When we investigated these cases in more detail, we discovered that with a lower support threshold there are quite many different combinations of frequent items present in the transactions of these data sets, and therefore our algorithm will also generate many candidate nodes, most of which corresponding to infrequent itemsets.

On the other hand, the results suggest that our algorithm is superior to FP-growth when the data set contains frequent itemsets with a large number of items and frequent items have many dependencies – for example, on *openview*

Table 3. The performance comparison of algorithms for the 1% support threshold

Data set name	# of frequent itemsets	# of nodes in the reduced trie	Max. size of freq. itemset	Runtime of our algorithm	Runtime of FP-growth
openview	$> 2^{64}$	2,257,548	65	469 s	> 24 hours
mailserver	11,359	559	13	113 s	165 s
fileserver	$\approx 2^{57}$	135,721	57	449 s	> 24 hours
webserver	14,083,903	39,816	20	1286 s	845 s
ibankserver	18,403	4,499	10	289 s	455 s
websess	80	81	6	3 s	2 s
ibanksess	3,181	1,186	10	10 s	10 s
snort	33	34	4	2 s	1 s

Table 4. The performance comparison of algorithms for the 0.5% support threshold

Data set name	# of frequent itemsets	# of nodes in the reduced trie	Max. size of freq. itemset	Runtime of our algorithm	Runtime of FP-growth
openview	$> 2^{64}$	3,161,081	65	601 s	> 24 hours
mailserver	50,863	1,927	14	113 s	174 s
fileserver	$> 2^{64}$	275,525	87	489 s	> 24 hours
webserver	38,735,679	84,679	21	2229 s	855 s
ibankserver	53,105	11,430	11	307 s	495 s
websess	280	281	6	3 s	3 s
ibanksess	6,279	2,229	10	10 s	10 s
snort	42	43	4	2 s	1 s

Table 5. The performance comparison of algorithms for the 0.1% support threshold

Data set name	# of frequent itemsets	# of nodes in the reduced trie	Max. size of freq. itemset	Runtime of our algorithm	Runtime of FP-growth
openview	$> 2^{64}$	7,897,598	65	3395 s	> 24 hours
mailserver	302,505	8,997	15	117 s	192 s
fileserver	$> 2^{64}$	2,235,271	118	834 s	> 24 hours
webserver	319,646,847	443,625	24	8738 s	949 s
ibankserver	328,391	71,229	11	375 s	518 s
websess	2,346,654	1,076,663	21	329 s	17 s
ibanksess	41,103	12,826	12	15 s	12 s
snort	214	121	7	2 s	1 s

and *fileserver* data sets our algorithm is much faster. The reason for the poor performance of FP-growth is as follows – when the data set contains many frequent k-itemsets for larger values of k, the total number of frequent itemsets is very large, and since FP-growth must visit each frequent itemset during its work, its runtime cost is simply too high. This raises an interesting question – can the FP-growth algorithm be augmented with the same technique that our algorithm uses, i.e., if the frequent itemset P does not contain its dependency prefix, the algorithm will not search for frequent itemsets that begin with P. Unfortunately, when frequent items are ordered in dependency ascending order, frequent items of transactions will be saved to FP-tree in dependency descending (reverse) order, because the FP-growth algorithm processes the FP-tree in a bottom-up manner [13]. Since items with more dependencies tend to be less frequent, the FP-tree nodes closer to the root node are less likely to be shared by many transactions, and the resulting FP-tree is highly inefficient in terms of memory consumption. When conducting experiments with data sets from Table 1, we found that the FP-tree did not fit into main memory in several cases.

We have developed a mining tool called LogHound that implements our algorithm. The tool can be employed for mining frequent line patterns from raw event logs, but also for mining frequent event type patterns. LogHound has several options for preprocessing input data with the help of regular expressions. In order to limit the number of patterns reported to the end user, it has also an option to output only those patterns that correspond to closed frequent itemsets. Figure 3 depicts some sample patterns that have been discovered with LogHound.

```
Dec 18 * myhost * connect from
Dec 18 * myhost * log: Connection from * port
Dec 18 * myhost * fatal: Did not receive ident string.
Dec 18 * myhost * log: * authentication for * accepted.
Dec 18 * myhost * fatal: Connection closed by remote host.
```

(a) Sample frequent line patterns

```
[1:1256:7] [1:1002:5] [119:2:1] [1:1945:1]

[1:1256:7] - WEB-IIS CodeRed v2 root.exe access
[1:1002:5] - WEB-IIS cmd.exe access
[119:2:1]  - HTTP DOUBLE DECODING ATTACK
[1:1945:1] - WEB-IIS unicode directory traversal attempt
```

(b) Sample frequent event type pattern
(the CodeRed worm footprint that was detected from the Snort IDS log)

Fig. 3. Sample frequent patterns detected with LogHound.

LogHound is written is C, and has been tested on Linux and Solaris platforms. It is distributed under the terms of GNU GPL, and is available at http://kodu.neti.ee/~risto/loghound/.

6 Conclusion

In this paper, we presented an efficient breadth-first frequent itemset mining algorithm for mining frequent patterns from event logs. The algorithm combines the features of well-known breadth-first and depth-first algorithms, and also takes into account the special properties of event log data. The experiment results indicate that our algorithm is suitable for processing event log data, and is in many occasions more efficient than well-known depth-first algorithms.

References

1. C. Lonvick: The BSD syslog Protocol. RFC3164 (2001)
2. H. Mannila, H. Toivonen, and A. I. Verkamo: Discovery of frequent episodes in event sequences. Data Mining and Knowledge Discovery **1(3)** (1997) 259-289
3. Qingguo Zheng, Ke Xu, Weifeng Lv, and Shilong Ma: Intelligent Search of Correlated Alarms from Database Containing Noise Data. Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium (2002) 405-419
4. Sheng Ma and Joseph L. Hellerstein: Mining Partially Periodic Event Patterns with Unknown Periods. Proceedings of the 16th International Conference on Data Engineering (2000) 205-214
5. Jian Pei, Jiawei Han, Behzad Mortazavi-asl, and Hua Zhu: Mining Access Patterns Efficiently from Web Logs. Proceedings of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining (2000) 396-407
6. Jaideep Srivastava, Robert Cooley, Mukund Deshpande, and Pang-Ning Tan: Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data. ACM SIGKDD Explorations **1(2)** (2000) 12-23
7. Mika Klemettinen: A Knowledge Discovery Methodology for Telecommunication Network Alarm Databases. PhD thesis, University of Helsinki (1999)
8. Risto Vaarandi: A Data Clustering Algorithm for Mining Patterns From Event Logs. Proceedings of the 2003 IEEE Workshop on IP Operations and Management (2003) 119-126
9. Rakesh Agrawal and Ramakrishnan Srikant: Fast Algorithms for Mining Association Rules. Proceedings of the 20th International Conference on Very Large Data Bases (1994) 478-499
10. Ferenc Bodon: A fast APRIORI implementation. Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (2003)
11. Christian Borgelt: Efficient Implementations of Apriori and Eclat. Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (2003)
12. Mohammed J. Zaki: Scalable Algorithms for Association Mining. IEEE Transactions on Knowledge and Data Engineering **12(3)** (2000) 372-390
13. Jiawei Han, Jian Pei, and Yiwen Yin: Mining Frequent Patterns without Candidate Generation. Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (2000) 1-12
14. Bart Goethals: Memory issues in frequent itemset mining. Proceedings of the 2004 ACM Symposium on Applied Computing (2004) 530-534
15. http://www.cs.helsinki.fi/u/goethals/software/index.html
16. M. V. Ramakrishna and Justin Zobel: Performance in Practice of String Hashing Functions. Proceedings of the 5th International Conference on Database Systems for Advanced Applications (1997) 215-224