CERIAS Tech Report 2000-03

Incremental Mining of Partial Periodic Patterns in Time-series Databases

Mohamed G. Elfeky
Center for Education and Research in
Information Assurance and Security
Purdue University, West Lafayette, IN 47907

Incremental Mining of Partial Periodic Patterns in Time-Series Databases

Mohamed G. Elfeky

February 3, 2000

1 Introduction

A Time-Series Database is a database that contains data for each point in time; e.g., weather data that contains several measures (e.g., the temperature) at different times per day. Some other examples are the stock prices and the power consumption. Mining time-series databases involves two general kinds of periodic patterns: full periodic patterns and partial periodic patterns. In full periodic patterns, every point in time contributes to the cyclic behavior of the time series for each period [1]. For example, describing the weekly stock prices pattern considering all the days of the week. The other kind is partial periodic patterns, which specify the behavior of the time series at some but not all the points in time [1]. For example, discovering that a specific stock prices are high every Saturday and low every Tuesday but do not have such regularity on other days. An efficient algorithm for mining partial periodic patterns is introduced in [1].

One of the important problems of the data mining problem is how to maintain the discovered rules or patterns over the time in the sense that the data is updated regularly. Especially in time-series databases, new data is added continuously over the time. Incremental mining concerns this problem. It is how to mine a previously mined database after the addition of some data without running the mining algorithm again on the new database. A new algorithm is proposed for incrementally mine partial periodic patterns in time-series databases based on the algorithm discussed in [1].

Another problem that is arisen here is that what if there are two time-series databases and now it is wanted to combine them into one, how to discover the partial periodic patterns of the combined database, given that the two databases were previously mined, without running the mining algorithm again on the combined database. The proposed algorithm is extended to solve this problem.

The rest of this report is organized as follows. In Section 2, the algorithm of mining partial periodic patterns discussed in [1] is briefly introduced. Section 3 discusses in detail the proposed algorithm for incremental mining of partial periodic patterns. In section 4, some statistical analysis of the proposed algorithm is shown. Finally, Section 5 presents a proposed solution to the latter problem discussed above.

2 Mining Partial Periodic Patterns

2.1 Problem Definition

Assume that a sequence of *n* time-stamped datasets have been collected in a database. For each time instant *i*, let D_i be a feature derived from the dataset. Thus, the time series of features is represented as, $S = D_1, D_2, ..., D_n$. A **pattern** is: $s = s_1 ... s_p$ over the set of features L and the letter $*$, such that p is the **period** of the pattern. L-length of a pattern s is the number of s_i that is not $*$. A pattern with L-length j is also called a j-pattern. A **subpattern** of a pattern s is a pattern $s' = s'_1 ... s'_p$ such that for each position i: $s'_i = s_i$ or $s'_i = *$. For example, if the period is 5, the pattern: $a{*}cde$ is called 4-pattern. Two of his subpatterns are: $a{*}c{*}e$ and ${*}{*}cd{*}$.

Each segment of the form $D_{ip+1} ... D_{ip+p}$ is called a **period segment**. A period segment **matches** a pattern s if it is a subpattern of s. The **frequency count** of a pattern in a time series is the number of period segments of this time series that matches that pattern. The **confidence** of a pattern is defined as the division of its frequency count by the maximum number of period segments in the time series. A pattern is called **frequent** if its confidence not less than a minimum threshold. For example, in the series *abbaebaced*, the pattern: $a{*}b$, whose period is 3, has frequency count 2, and confidence $2/3$ where 3 is the maximum number of period segments of length 3.

2.2 Max-Subpattern Hit Set Method

A **max-pattern** C_{max} is the maximal pattern that can be generated from F_1 (the set of frequent 1-patterns). For example, if $F_1 = \{a{*}{*}{*}{*}, {*}b{*}{*}{*}, {*}{*}c{*}{*}, {*}{*}{*}{*}d\}$, then $C_{max} = abc{*}d$. A subpattern of C_{max} is hit in a period segment S_i if it is the maximal subpattern of C_{max} in S_i. For example, if $C_{max} = abc{*}d$ and $S_i = abdcd$, then its hit subpattern is: $ab{*}{*}d$.

Algorithm:

1. Scan the database to find F_1, the frequent patterns of length 1, and form the candidate max-pattern C_{max}.
2. Scan the database once again, and for each period segment, consider its max subpattern and update the max-subpattern tree (either add the pattern setting its count to 1 if it is not exist, or otherwise increase its count by 1).
3. Traverse the tree to determine the actual count of each pattern, and determine the frequent ones.

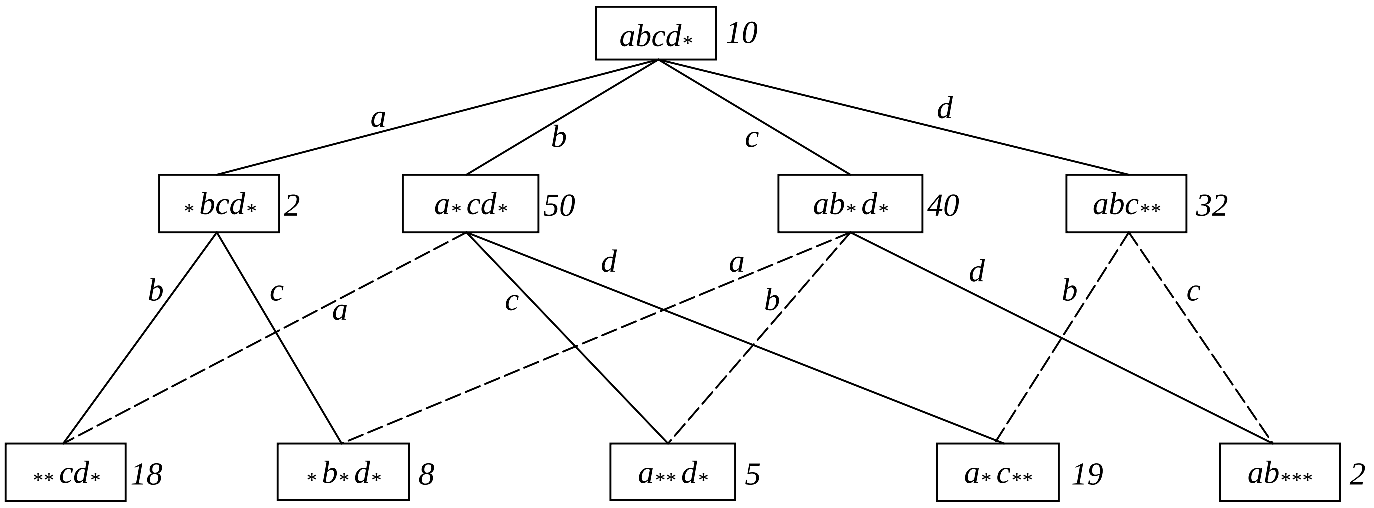

Figure 1. An example to a max-subpattern tree

Figure 1 shows an example of a max-subpattern tree used to store the set of max-subpatterns hit in the time series. It shows that the root node is the C_{max}. A child node is a subpattern of the parent node with one non $*$-letter missing. The link is labeled by this letter. The *1*-patterns are not stored in the tree since they already exist in F_1 table. Each node has a count field that registers its number of hits. A dotted link is a non-existing (virtual) link that represents a candidate parent of the child node. The frequency count of each pattern represented by a node is the sum of its count and those of all its reachable parents (physical or virtual link). For example, the frequency count of $**cd*$ is *80*, and the frequency count of $a**d*$ is *105*.

3 Incremental Mining of Partial Periodic Patterns

3.1 Problem Definition

The problem of incremental mining takes as an input a previously mined database after some additions of new data, and sufficient information about the previously mined patterns collected before from running the mining algorithm over the old database. Here, it is sufficient to store the max-subpattern tree and all the *1*-patterns along with their respective counts.

3.2 Algorithm

1. Scan the new data for the *1*-patterns and add them to the whole list of *1*-patterns, and determine the new max-pattern C_{max}.
2. If C_{max} is not changed, there will be no change on the tree.
3. If C_{max} is changed, there are two cases: deleted letters and inserted letters (either or both).
    a. For the deleted letters, delete from the new C_{max} the inserted letters, and consider the resulted pattern, that is surely a subpattern of the old C_{max}, as the root of the new tree. Insert into the new tree the hit of each pattern in the old tree with the same count. Now, the old C_{max} is the new C_{max}.
    b. For the inserted letters, the old C_{max} is surely a subpattern of the new one. Hence, consider the new C_{max} as the root of the tree making the old root a child. Scan the old database, and for each period segment containing at least one of the inserted letters, insert its hit in the tree, and decrement the node that contains the hit without the inserted letters.
4. Scan the new data and update the max-subpattern tree as step 2 of the original mining algorithm.
5. Traverse the tree to determine the actual count of each pattern, and determine the frequent ones.

The following example, followed from Figure 1, illustrates the idea behind this algorithm. Assume that the new C_{max} is $abc*e$, which means one deleted letter d and one inserted letter e. Following step 3a of the algorithm, the root of the new tree is: $abc**$, resulting in Figure 2. Notice that: (i) the frequency count of each pattern is the same as in the old tree, and this why the hits of the patterns of the old tree are inserted in the new one, (ii) the virtual links become physical, (iii) a new node appears $*bc**$ resulted from inserting the hit of the node $*bcd*$.

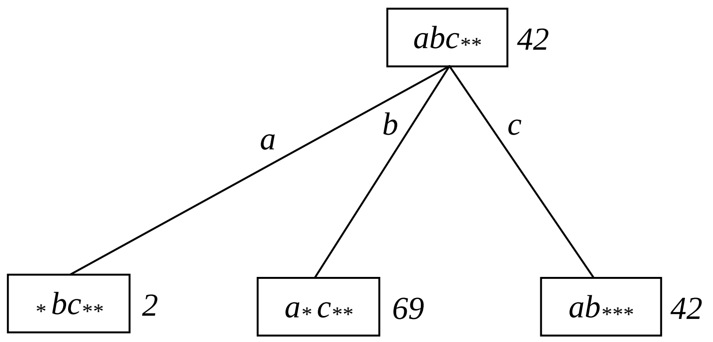

Figure 2. The max-subpattern tree of Figure 1. after deleting the letter d.

Now, following step 3b of the algorithm, the root is *abc* * *e*, resulting in Figure 3. Notice that: (i) there are new nodes other than the root, like *ab*** *e*, and this is why the old database must be rescanned to discover the hits given the new max pattern, (ii) the frequency count of each pattern is the same as in the old tree, and this why some nodes must be decremented.

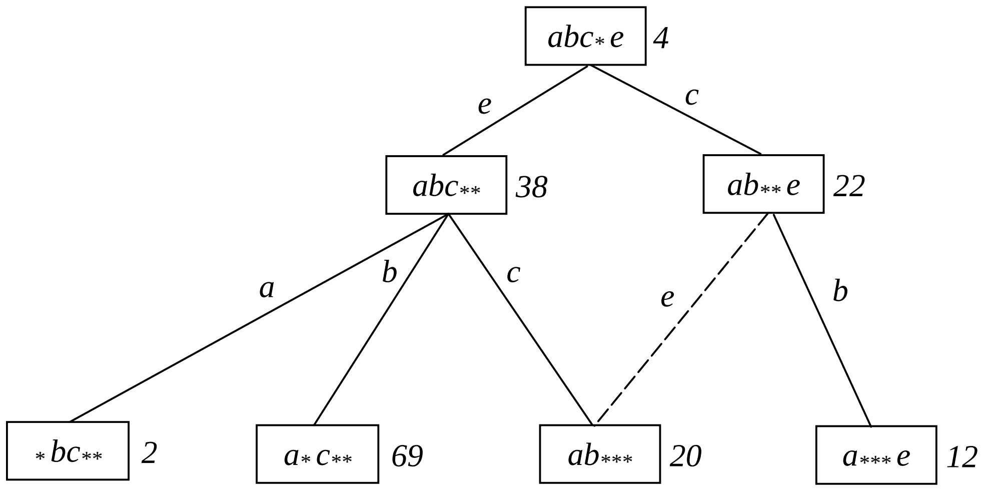

Figure 3. The max-subpattern tree of Figure 2. after inserting the letter e.

4 Analysis

Clearly, incremental mining will speed up the process of mining a previously mined database after the addition of some new data, but it relies on how much often the algorithm is applied; i.e., with each new segment of data or after a significant number of additions. The following statistics may be incomplete but give an idea about the speed up of the proposed algorithm over applying the original mining algorithm on the new database again. Also, the proposed algorithm implies some overhead of the original mining algorithm in the sense that the created tree must be stored on the physical storage, which requires additional time and space.

It can be inferred easily from the algorithm that it has a worst case of one scan over the old database if there are inserted letters in C_{max} (Step 3b). This scan can be eliminated if a list of the segments that contain any *1*-pattern is stored along with this *1*-pattern. This list is called *inverted list*. Now, step 3b is modified such that the period segments that must be reinserted in the tree are now located in the inverted list without a need to rescan the old database searching for those period segments.

5 Merging Mined Databases

5.1 Problem Definition

The problem of merging two previously mined databases and discover the partial periodic patterns from the new one can be considered a generalization to the problem of incremental mining. Now the input is two trees and two *1*-pattern lists, one for each database.

The worst case in the proposed algorithm is to scan each database once, if the last modification is not considered, instead of scanning the new database twice if the original mining algorithm is considered.

5.2 Algorithm

1. Determine the C_{max} that will be the root of the new tree. This can be done by either way of the following:
    a. Merge the two *1*-pattern lists and determine the new C_{max} as Step 1 of the original algorithm.
    b. Intersect the two C_{max}'s of the two trees, and for each position that will have a $*$, select either the letter of the first C_{max} or the second one based on how frequent is this letter in the merged list.
2. Apply Steps 2 and 3 of the proposed algorithm for each tree considering the new C_{max} determined in the previous step.
3. Now, the two trees have the same root and then can be combined into one by selecting the one that has smaller number of nodes and insert it into the other.
4. Traverse the tree to determine the actual count of each pattern, and determine the frequent ones.

References

[1] Jiawei Han, Guozhu Dong, and Yiwen Yin. Efficient Mining of Partial Periodic Patterns in Time Series Databases. In *Proceedings of 1999 Int. Conf. on Data Engineering (ICDE'99)*, Sydney, Australia, March 1999.

[2] Shiby Thomas, Sreenath Bodagala, Khaled Alsabti, and Sanjay Ranka. An Efficient Algorithm for the Incremental Updation of Association Rules in Large Databases. In *Proceedings of the 3rd Int. Conf. on Knowledge Discovery and Data Mining*, New Port Beach, California, August 1997.

Implementation

Both the original mining algorithm and the proposed incremental one are implemented using Java programming language. To use the original mining algorithm: use *java Mine arg1 arg2 arg3*, such that *arg1* is the number of the data file, *arg2* is the period, and *arg3* is the confidence threshold. For examples, *java Mine 1 7 60* means to mine the first data file with period *7* (7 days per week), and the confidence threshold is 0.6. For each data file there are two versions, one with the original data, and the other after the addition of new data. Now, to use the incremental algorithm: use *java IncMine arg1*, such that *arg1* is the number of the data file. For example, *java IncMine 1* means to apply the incremental mining algorithm over the new data file. To ensure that the results of the incremental mining algorithm are correct, it is needed to apply the original mining algorithm over the new data file and this can be done using: *java Mine 1i 7 60*, such that *1i* is the number of the new data file.