YAFIMA: Yet Another Frequent Itemset Mining Algorithm

Mohammad El-Hajj, Osmar R. Zaïane
Department of Computing Science
University of Alberta, Edmonton, AB, Canada
{mohammad, zaiane}@cs.ualberta.ca

ABSTRACT: *Efficient discovery of frequent patterns from large databases is an active research area in data mining with broad applications in industry and deep implications in many areas of data mining. Although many efficient frequent-pattern mining techniques have been developed in the last decade, most of them assume relatively small databases, leaving extremely large but realistic datasets out of reach. A practical and appealing direction is to mine for closed or maximal itemsets. These are subsets of all frequent patterns but good representatives since they eliminate what is known as redundant patterns. The practicality of discovering closed or maximal itemsets comes from the relatively inexpensive process to mine them in comparison to finding all patterns. In this paper we introduce a new approach for traversing the search space to discover all frequent patterns, the closed or the maximal patterns efficiently in extremely large datasets. We present experimental results for finding all three types of patterns with very large database sizes never reported before. Our implementation tested on real and synthetic data shows that our approach outperforms similar state-of-the-art algorithms by at least one order of magnitude in terms of both execution time and memory usage, in particular when dealing with very large databases.*

Categories and Subject Descriptors
H.2 [Database Management] H.2.8[Database Applications]: **Data Mining**

General Terms
Large Data sets, Database algorithm, Data Mining

Keywords: Database, Data Mining, Frequent pattern mining, Traversal approaches

Received 30 Oct. 2004; Reviewed and accepted 30 Jan. 2005

1. Introduction

Discovering frequent patterns is a fundamental problem in data mining. Many efficient algorithms have been published on this problem in the last 10 years. Most of the existing methods operate on databases made of comparatively small database sizes. Given different small datasets with different characteristics, it is difficult to say which approach would be a winner. Moreover, on the same dataset with different support thresholds different winners could be proclaimed. Difference in performance becomes clear only when dealing with very large datasets. Novel algorithms, otherwise victorious with small and medium datasets, can perform poorly with extremely large datasets. There is obviously a chasm between what we can mine today and what needs to be mined. It is true that new attempts toward solving such problems are made by finding the set of frequent closed itemsets (FCI) [10, 11, 13] and the set of maximal frequent patterns [2, 6, 7].

1.1 Problem Statement

The problem of generating frequent itemset, as defined in [1] in the context of association rule mining, is stated as follows: Let $I = \{i_1, i_2, ..., i_m\}$ be a set of literals, called items and m is considered the dimensionality of the problem. Let D be a set of transactions, where each transaction T is a set of items such that T *is a subset of* I. A transaction T is said to contain X, a set of items in I, if X *is a subset of* T. An itemset X is said to be *frequent* if its *support* s is greater than or equal to a given minimum support threshold δ. A frequent itemset M is considered maximal if there is no other frequent set that is a superset of M. A frequent itemset C is considered closed if there is no other frequent set that is a superset of C and has at least the same support as C.

1.2 Related Work

The most important, and at the basis of many other approaches and algorithms, is *apriori* [1]. The property that is at the heart of *apriori* and forms the foundation of most algorithms simply states that for an itemset to be frequent all its subsets have to be frequent. This anti-monotone property reduces the candidate itemset space drastically. However, the generation of candidate sets, especially when very long frequent patterns exist, is still very expensive. Moreover, *apriori* is heavily I/O bound. Another approach that avoids generating and testing many itemsets is FP-Growth [8]. FP-Growth generates, after only two I/O scans, a compact prefix tree representing all sub-transactions with only frequent items. A clever and elegant recursive method mines the tree by creating projections called conditional trees and discovers patterns of all lengths without directly generating all candidates the way *apriori* does. However, the recursive method to mine the FP-tree requires significant memory, and large databases quickly blow out the memory stack. Another innovative approach is COFI [4]. COFI is faster or as fast as FP-Growth and requires significantly less memory. The idea of COFI, which we adopt in this paper, is to build projections from the FP-tree each corresponding to sub-transactions of items co-occurring with a given frequent item. These trees are built, efficiently mined and discarded one at a time making the footprint in memory significantly small. The COFI algorithm generates candidates using a top down approach, where its performance shows to be severely affected while mining databases that have potentially long candidate patterns that turn to be not frequent, as COFI needs to generate candidate sub-patterns for all its candidate patterns. We build upon this approach by changing the traversal approach, and finding the set of closed or maximal frequent patterns. We also resolve the problem of COFI using a new data structure. Finding the set of closed frequent patterns has been studied in the literature. In this section we will examine four state-of-the-art algorithms in this area, A-Close [10] is an *apriori*-like algorithm. This algorithm mines directly for closed frequent itemsets. It uses a breadth-search strategy. This algorithm shows to be one-order of magnitude faster than *apriori*, when mining with a small support. This algorithm shows, however, poor performance compared to *apriori* when mining with high support especially when we find a small set of frequent patterns as it consumes most of its computation power in computing the closure of itemsets. This algorithm did also show weak results when mining relatively long patterns. CLOSET+ [11] is an extension of the FP-Growth algorithm. MAFIA [2] is originally designed to mine for maximal itemsets, but it has an option to mine for closed itemsets. It uses a vertical bitmap representation. CHARM [13] is just like MAFIA in using a vertical representation of the database. It adopts the *Diffset* technique to reduce the size of intermediate tidsets. Mining the set of maximal patterns has also been investigated at length where some of the state-of-the-art algorithms are: GenMax [6], MAFIA [2], and FPMAX [7].

1.3 Contributions

In this paper we propose a new traversal approach for the itemset lattice called leap-traversal. Based on this approach we propose aframework of efficient algorithms, founded on our previously proposed COFI-trees in [3, 4] and FP-trees presented in [8], for finding closed, maximal, and all patterns called COFI-CLOSED, COFI-MAX, and COFI-ALL, forming the YAFIMA package (**Y**et **A**nother **F**requent **I**temset **M**ining **A**lgorithms). These algorithms mine effectively small and extremely large databases. They use a novel technique to generate candidates and to count their supports. We also propose a new structure to partition the itemsets helping the handling of patterns of arbitrary length. Since testing whether patterns are subsets of others is a fundamental operation for closed

and maximal itemset mining. We show that our approach with the new set of algorithms is efficient to mine extremely large datasets. The rest of this paper is organized as follows: We explain our new traversal approach in Section 2. The new COFI-based algorithms YAFIMA are explained in Section 3 with illustrative examples. Section 4 depicts the performance evaluation of this new approach comparing it with existing state-of-the-art algorithms in particular for its speed, scalability, and memory usage on dense and sparse data. Section 5 concludes and highlights our observations.

2 Leap-Traversal Approach

Current algorithms find patterns by using one of the two methods, namely: breadth-search or depth-search. Breadth-search can be viewed as a bottom-up approach where the algorithms visit patterns of size *k* + 1 after finishing the *k* sized patterns. The depth-search approach does the opposite where the algorithm starts by visiting patterns of size *k* before visiting those of size *k*+ 1. Both methods show some redundancy in mining specific datasets. [11] stated that the breadth-search is not the way to go. It recommends the depth-search method to mine for frequent long patterns. It is known that many datasets have short frequent patterns. Even for datasets that have long patterns, it does not mean at all that the patterns generated will be long. Many other frequent patterns are still short, and mining them using depth-search method might result in bad performance. In general, both methods show some efficiency while mining some databases. On the other hand, they showed weaknesses or inefficiency in many other cases.

To understand this process fully, we will try to focus on the main advantages and drawbacks of each one of these methods in order to find a way to make use of the best of both of them, and to diminish as much as possible their drawbacks. As an example, in the context for mining for maximal patterns, assume we have a transactional database that is made of a large number of frequent 1-itemsets, and has maximal patterns with relatively small lengths. The trees built from such database are usually deep as they have a large number of frequent 1-itemsets especially if they are made of relatively long transactions. Traversing in depth-search manner would provide us with potential long patterns that end-up non-frequent. In such cases, the depth-search method is not favored. However, if the maximal patterns generated are relatively long with respect to the depth of the tree, then the depth-search strategy is favored as most of the potential long patterns that could be found early tend to be frequent. Consequently, many pruning techniques could be applied to reduce the search space. On the other hand, mining transactional databases that reveal long maximal patterns is not favored using breadth-search manner, as such algorithms consume many passes over the database to reach the long patterns. Such algorithms generate many frequent patterns at level *k* that would be omitted once longer supersetpatterns at level *k* + 1, or *k* + *l* for any *l*, appear. These generation and deletion steps become a bottleneck while mining transactional databases of long frequent patterns using the breadth-search methods. From the above, we can summarize the main drawbacks of each one of these methods as follows: Bottom-up approaches generate too many frequent patterns that turn out to be not part of the maximal set and get deleted as longer frequent superset patterns appear. The longer maximal patterns we have, the more severe this problem becomes. For example; to find a frequent pattern of size 10 we need to find all patterns of sizes 1 to 9 before we reach the maximal pattern of size 10. Consequently, all these patterns are deleted and do not participate in the maximal answer set. On the other hand, the top-down approaches suffer from traversing many *n* - *k* non-frequent patterns until we reach the maximal frequent patterns of size *k*, *n* being the maximal depth of the tree. The following example demonstrates how both approaches work. Figure 1.A depicts 10 projected transactions made of 12 items. All items share at least one transaction with item A. If we want to mine with support greater than 4, then all items would become frequent. Three maximal patterns are generated. These patterns are (ABCDE: 6) of size 5, (AFGHI: 5) also of size 5 and (AJKL: 5) of size 4. The number after a pattern represents its support. Discovering these patterns using the top-down approach (depth-search) requires mining a tree of depth 9 which is the size of longest potential pattern: (ABCDEFGI). Although none of the potential patterns of size 9 to 6 are frequent, we still need to generate and test them. This generation process continues until we reach the first maximal patterns of size 5: (ABCDE: 6) and (AFGHI: 5). Many pruning techniques can then be applied to reduce the remaining search space. The bottom-up approach needs to create all patterns from sizes 2 to 6 at which point it can detect that there are no more maximal patterns to discover. All non-maximal frequent patterns of sizes 2 to 6 would be removed. We propose a combination of these approaches that takes into account a partitioning of the search space ála COFI-tree. We call this method the leap-traversal approach since it selectively jumps within the lattice and suggests a set of patterns from different sizes to test where the frequent patterns (all, closed, or maximals) are subset of this suggested set. To illustrate the traversal, we take the case of closed itemsets. Step one of this approach is to look at the nature of the distribution of frequent items in the transactional database. If we revisit the example presented above from Figure 1.A, we can see that there are only 4 distributions of frequent items. Indeed, A, B, C, D, E, F, G, H, I occurs 3 times; A, B, C, D, E, J, K, L occurs also 3 times; A, F, G, H, I occurs twice; and A, J, K, L also occurs twice. We call each one of these patterns a *frequent-path-base*. Step 2 of this process intersects each one of these patterns with all other combinations of *frequent-path-bases* to get a set of potential candidates. Step 3 counts the support of each one of the generated patterns. The support of each one of them is the summation of supports of all its superset of *frequent-path-base* patterns. Step 4 scans these patterns to remove non-

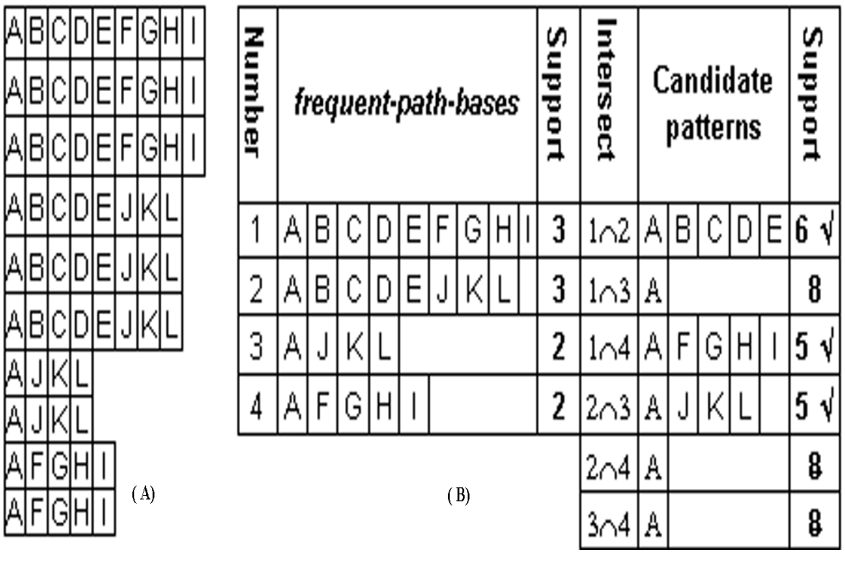

Figure 1: (A): Projected transactional database with respect to item A. (B): Steps needed to generate closed patterns using the leap-traversal approach (“ indicates a discovered closed pattern. Barred entries are the eliminated candidates)

frequent ones or frequent ones that already have a frequent superset with the same support. The remaining patterns can be declared as closed patterns. Figure 1.B illustrates the steps needed to generate the closed patterns of our example from Figure 1. A. The major goals of this approach are the followings:

1. Avoid the redundancy of testing patterns either from size 1 until patterns of size *k*, where *k* is the size of the longest frequent pattern or from patterns of size *n* until patterns of size *k*, where *n* is the size of the longest candidate pattern.

2. We only check the longest potential patterns that already exist in the transactional database, even if they are of different lengths. In Figure 1.A we can find that there is no need to test patterns such as ABJ or AFC since they never occur in the transactional database. We also do not need to test patterns such as AB since they never occur alone without any other frequent items in the transactional databases.

The main question in this approach is whether we could efficiently find the *frequent-path-bases*. The answer is yes, by using the FP-tree [8] structure to compress the database and to avoid multiple scans of the databases and COFI-trees [4] to partition the sub-transactions as we wish to do, to generate the *frequent-path-bases* as illustrate in the next section.

The well-known FP-tree [8] data-structure is a prefix tree. The data structure presents the complete set of frequent itemsets in a compressed fashion. The construction of FP-Tree requires two full I/O scans. The first scan generates the frequent 1-itemsets. In the second scan, non-frequent items are stripped off the transactions and the sub-transactions with frequent ones are ordered based on their support. These sorted sub-transactions form the paths of the tree. Sub-transactions that share the same prefix share the same portion of the path starting from the root. The FP-tree has also a header table containing frequent items and holds the head link for each item in the FP-tree, connecting nodes of the same item to facilitate the item traversal during the mining process. We invite the reader to see [8] for more details.

A COFI-tree [4] is a projection of each frequent item in the FP-tree. Each COFI-tree, for a given frequent item, presents the co-occurrence of this item with other frequent items that have more support than it. In other words, if we have 4 frequent items A, B, C, D where A has the smallest support, and D has the highest, then the COFI-tree for A presents co-occurrence of item A with respect to B, C and D, the COFI-tree for B presents item B with C and D. COFI-tree for C presents item C with D. Finally, the COFI-tree for D is a rootnode tree. Each node in the COFI-tree has two main variables, support and participation. Participation indicates the number of patterns the node has participated in at a given time during the mining step. Based on the difference between these two variables, participation and support, frequent-path-bases are generated.

The steps are the same as the steps done in our previous work in [4] to prepare the long patterns before generating sub-pattern. The COFI-tree has also a header table that contains all locally frequent items with respect to the root item of the COFI-tree. Each entry in this table holds the local support, and a link to connect its item with its first occurrences in the COFI-tree. A link list is also maintained between nodes that hold the same item to facilitate the mining procedure.

3 COFI-Based Algorithms

In this section we explain the three mining algorithms COFI-CLOSED, COFI-MAX, and COFI-ALL. COFI-CLOSED is explained in details while COFI-MAX and COFI-ALL are explained later by only highlighting their differences from the COFI-CLOSED algorithm.

3.1 COFI-CLOSED Algorithm

COFI-CLOSED algorithm is explained by a running example. The complete pseudo-code of the algorithm is detailed in Figure 2. The transactional database in Figure 4.A needs to be mined using support greater or equal to 3. The first step is to build the FP-tree data-structure in Figure 4.B. This FP-tree data structure reveals that we have 8 frequent 1-itemsets. These are (A:10, B:8, C:7, D:7, E:7, F:6, G:5, H:3). COFI-trees are built after that one at a time starting from the COFI-tree of the frequent item with lowest support, which is H, we refer the reader to [4] for more reading about COFI-tree. Since, in the order imposed, no other COFI-tree has item H then any closed pattern generated from this tree is considered globally closed. This COFI-tree generates the first closed pattern HA: 3. After that, H-COFI-tree is discarded and G-COFI-tree, in Figure 4.C, is built and it generates (GABCD:3, GABC:4, GAE:3, GAD:4, and GA:5), a detailed explanation of the steps in generating these frequent patterns are described later in this section. F-COFI-tree is created next and it generates all its closed patterns using the same method explained later.

3.2 Mining a COFI-Tree

Mining a COFI tree starts by finding the *frequent-path-bases*. As an example, we will mine the G-COFItree in Figure 4.C for closed patterns. We start from the most globally frequent item, which is A, and then traverse all the A nodes. If the *support* is greater than *participation* then the complete path from this node to the COFI-root

Algorithm : *COFI-CLOSED*
Input : *Transactional Database f, and minimum support σ*
Output : *A set of closed frequent patterns CT*
Method :
1 Scan the database *f* twice to build the FP-Tree based on σ
2 Initialize the CLOSED-TREE *CT* to ψ
3 Start from item *A*, where *A* is the frequent item with lowest support
4. While *A* exists
  i. Build *A-COFI-tree*
  ii. Mine-Closed (*A-COFI-tree*, σ)
  b. *A* = next frequent item with lowest support
  c. Go To 4
5. **Release FP-Tree**

Function : *Mine-Closed*
Input : *A-COFI-tree and minimum support σ*
Output : *Update CT with the set of closed items with respect to item A*
Method :
1. Initialize the Ordered-Partitioning-Bases *OPB*
2. Scan the *A-COFI-tree* for Frequent-Path-Bases *FPB*.
  a. Add *FPB* to *OPB*. The location of *FPB* is in the link that corresponds to the length of *FPB* in *OPB*. The *support* and *branch-support* for *FPB* are the same and equal to the *branch-support* for that pattern generated from the **A-COFI-tree.**
  b. If there are more *FPB* then go to 2
3. Intersect all *FPBs* in *OPB*.
  a. If a new pattern *C* generated (i.e not in *OPB*) then
  i. Add *C* to *OPB*
  ii. Find the global support of *C* (The support of an *X* pattern in *FPB* = Summation of *branch-supports* of *Y FPBs* in *OPB*. Where *Y* is {∀ *FPB* such that $X \subseteq Y$}
4. Remove all non-frequent patterns and frequent patterns that are subset of already existing frequent patterns from *OPB* that have the same support
5. For any remaining frequent pattern *X*. if *X* not subset of *CT* or ($X \subseteq CT$, with different support) then declare *X* as Closed frequent pattern and $CT = CT \cup X$
6. **Release OPB**
7. **Release** *A-COFI-tree*

Figure 2: COFI-CLOSED Algorithm

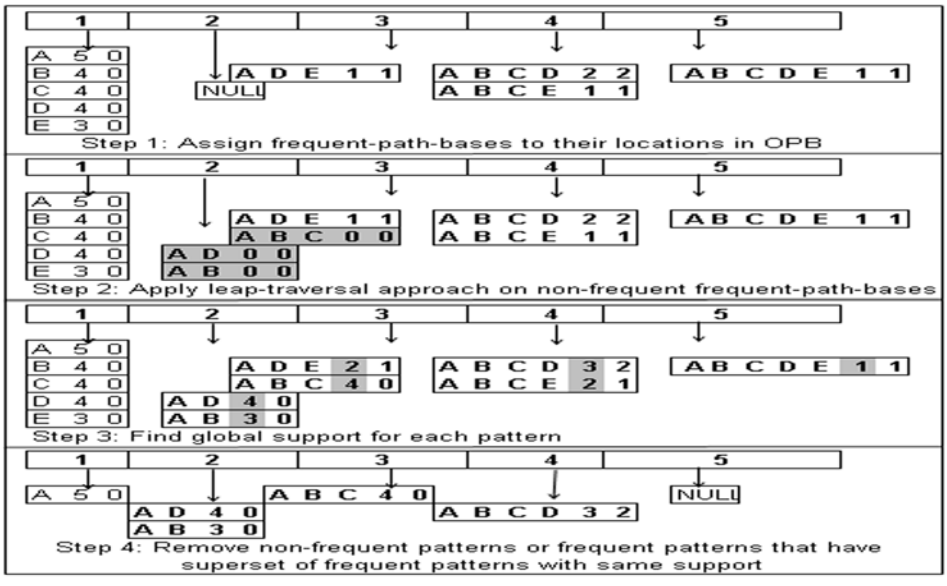

Figure 3: Steps needed to generate closed patterns using OPB for the G-COFI-tree

is built with *branch-support* equals to the difference between the *support* and *participation* of that node. All values of *participation* for all nodes in these paths are updated with the *participation* of the original node A. *Frequent-path-bases* (A, B, C, D: 2), (A, B, C, E: 1), (A, D, E: 1), and (A, B, C, D, E: 1) are generated from this tree. From these bases we create a special data structure called Ordered-Partitioning-Bases (OPB). The goal of this data structure is to partition the patterns by their length. Patterns with the same length are grouped together. This, on one hand allows dealing with patterns of arbitrary length, and on the other hand allows traversing the pattern space from the longest ones to the shortest ones and directly prunes the short ones if a frequent superset with same support is discovered as a candidate closed pattern. This OPB structure is an array of pointers that has a size equal to the length of the largest *frequent-path-base*. Each entry in this array connects all *frequent-path-bases* of the same size. The first entry links all *frequent-path-bases* of size 1, the second one refers to all *frequent-path-bases* of size 2, the *nth* one points to all *frequent-path-bases* of size *n*. An illustrative example can be found in Figure 3. Each node of the connected link list is made of 4 variables which are: the pattern, a pointer to the next node, and two number variables that represent the *support* and *branch-support* of this pattern. The *support* reports the number of times this pattern occurs in the database. The *branch-support* records the number of times this pattern occurs alone without other frequent items, i.e. not part of any other superset of frequent patterns. This *branch-support* is used to identify the *frequent-path-bases* from *non-frequent-path-bases* as *non-frequent-path-bases* have *branch-support* equal to 0, while a *frequent-path-base* has *branch-support* equals to the number of times this pattern occurs independently. The *branch-support* is also used to count the support of any pattern in the OPB. The support of any pattern is the summation of the *branch-supports* of all its supersets of *frequent-path-bases*. For example, to find the support for pattern *X* that has a length of *k*, all what we need to do is to scan the OPB from *k* + 1 to *n* where *n* is the size of OPB, and sum the *branch-supports* of all supersets of *X* that do not have a *branch-support* equal to 0, i.e. the *frequent-path-bases*.

In the above example, the first step is to build the OPB structure. The first pointer of this OPB structure points to 5 nodes which are (A, 5, 0), (B, 4, 0), (C, 4, 0), (D, 4, 0), and (E, 3, 0) which can be taken from the local frequent array of the G-COFI-tree (Figure 4.C). The first number after the pattern presents the *support,* while the second number presents the *branch-support*. The Second entry in this array points to all *frequent-path-bases* of size two. A null pointer is being linked to this node since no *frequent-path- bases* of size two are created. The third pointer points to one node which is (ADE, 1,1), the fourth points to (ABCD: 2: 2) and (ABCE: 1, 1), the fifth and last points to (ABCDE: 1:1). The leap-traversal approach is applied in the second step on the 4 *frequent-path-bases*, which are (ABCDE: 1: 1, ABCD:2 :2, ABCE: 2: 1, and ADE: 2: 1). Intersecting ABCDE with ABCD gives ABCD, which already exists, so nothing needs to be done. Same occurs when interesting ABCDE with ABCE. Intersecting ABCDE with ADE gives back ADE, which also already exists. Intersecting ABCD with ABCE gives ABC. ABC is a new node of size three. It is added to the OPB data structure and linked from the third pointer as it has a pattern size of 3. The *support* and the *branch-support* of this node equals to 0. *Branch-support* equals to 0 indicates that this pattern is a result of intersecting between *frequent-path-bases* and a *non-frequent-path- base*. Intersecting ABCD with ADE gives AD. AD is a new node of size two. It is added to the OPB

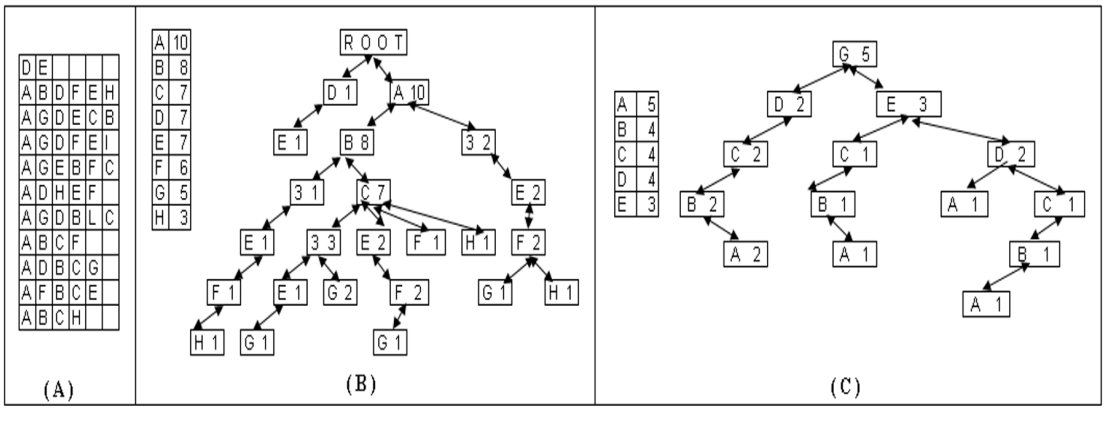

Figure 4: (A) A Transactional database. (B) FP-Tree built from (A). (C) G-COFI-tree

data structure and linked from the second pointer. The *support* and the *branch-support* of this node equal 0. Intersecting ABCE with ADE gives AE. AE is also a new node of size two and is also added to the OPB structure, and at this stage we can detect that no more intersection is needed. The third step in the mining process is to find the global support for all patterns. Applying a top-down traversal between these nodes does this. If node *X* is a subset of a *frequent-path-base Y* then its support is incremented by the *branch-support* of node *Y*. By doing this we can find that ABCD is a subset of ABCDE, which has a *branch-support* equals to 1. The ABCD support becomes 3 (2+1). ABCE support becomes 2, as it is a subset of ABCDE. At level 3 we find that ADE is a subset of only ABCDE so its support becomes 2. ABC support equals to 4. AD support equals to 4, and AE support equals to 3. At this stage all non-frequent patterns and frequent patterns that have a local frequent superset with the same support are removed from OPB. The remaining nodes (ABCD: 3, ABC: 4, AE: 3, AD: 4, and A:5) are potentially global closed. We test to see if they are a subset of already discovered closed patterns with the same support. If not, then we declare them as closed patterns and add them to the pool of closed patterns. These steps are presented in Figure 3. The G-COFI-tree and its OBP data structure are cleared from memory as there is no need for them any more. The same process repeats with the remaining COFI-trees for F and E, where any newly discovered closed pattern is added to the global pool of closed patterns.

3.3 COFI-MAX

Finding the maximal frequent items differs from finding the set of closed patterns by applying the following pruning techniques:

(A) skip building some COFI-trees: Before we build any COFI-tree, we check all its local frequent items, if all its items are subsets of already discovered maximal patterns, then there is no need to build and mine this COFI-tree as all its sub-patterns are subsets of already discovered maximal patterns.

(B) Remove all *frequent-path-bases* **that are subset of already found maximal patterns**: Each *frequent-path-base* is checked first to see if it is part of an already existing maximal pattern. If this check returns true, then there is no need to test this pattern, as it is already frequent, and a subset of a maximal pattern.

(C) Count the support of *frequent-path-bases* **early, and remove the frequent ones from the leap-traversal approach**: The first step done after finding the *frequent-path-bases* is to find their global support. The support of each *frequent-path-base* is the summation of its *branch-support* with the *branch-support* of all its superset of *frequent-path-bases*. All frequent *frequent-path-bases* are removed from the leap-traversal approach as they will only give subsets of frequent patterns and will not provide us with any new information in the context of searching for maximal patterns. By doing this, we reduce the intersection and counting operations done during the leap-traversal approach. Other than this, the same steps applied to find the closed patterns are also applicable to find the set of maximal pattern.

3.4 COFI-ALL

Finding the set of all frequent patterns is simply done by finding the set of local maximal for each COFItree. There is no need to keep track of globally maximal patterns. From each locally maximal pattern, we generate its subsets. These are the frequent itemsets. We only need to find their respective *support*. The *support* of each pattern is the summation of the *branch-support* of all its superset of *frequent-path-bases*.

4 Performance Evaluations

In this section we present a performance study to evaluate our new approach YAFIMA against most of the state-of-art algorithms that mine all, closed and maximal patterns. FPMAX and MAFIA were used for all three types of patterns, while Eclat was used to mine for all frequent patterns, CHARM to mine for closed itemsets, and finally GENMAX to mine for maximal frequent itemsets. Their respective authors provided us with the source code for these programs. All our experiments were conducted on an IBM P4 2.6GHz with 1GB memory running Linux 2.4.20-20.9 Red Hat Linux release 9. Timing for all algorithms includes the pre-processing cost such as horizontal to vertical conversions. The time reported also includes the program output time. We have tested these algorithms using both real and synthetic datasets on small and very large datasets. All experiments were forced to stop if their execution time reached our wall time of 5000 seconds. We made sure that all algorithms reported the same exact set of frequent itemsets on each dataset. Finally, we could not report all our excrements in this work due to the lack of space.

4.1 Experiments on Small Datasets

In this set of experiments we tested many datasets, UCI datasets and synthetic ones, with one goal in our mind: Finding the best mining algorithm. We tested all the enumerated algorithms using 4 databases downloaded from [5]. These databases are *chess, mushroom, pumsb, and accidents*. We have also generated synthetic datasets using [9]. In these sets of experiments we confirmed the conclusion made at the FIMI 2003 workshop [5], that there are no clear winners with small databases. Indeed, algorithms that were shown to be winners with some databases were not the winners with others. Some algorithms quickly lose their lead once the support level becomes smaller. Figure 5, depicts as an example the winner algorithms for these datasets.

4.2 Scalability

Mining extremely large databases is the main objective of this research work. We used five synthetic datasets made of 5M, 25M, 50M, 75M, 100M transactions, with a dimension of 100K items, and an average transaction length of 24 items. To the best of our knowledge these data sizes have never been reported in the literature before. Eclat, CHARM, and GENMAX could not mine these datasets. MAFIA could not mine the smallest dataset 5M in the allowable time frame. Only FPMAX and YAFIMA algorithms participated in this set of experiments. FP-Growth was able to mine efficiently for all frequent patterns up to 5 millions. After that point, FP-Growth could not return a result. COFI-ALL efficiently found the set of all patterns up to 100M transactions. For the set of closed itemsets and the set of maximal itemsets FP-CLOSED and FPMAX mined up to 50M transactions, while COFI-CLOSED and COFI-MAX mined all databases up to 100M transactions efficiently. All results are depicted in Figure 6. From these experiments we can see that the difference between FPMAX implementations and the YAFIMA algorithms become clearer once we mine extremely large datasets. YAFIMA saves at least one third of the execution time and in some cases goes up to half of the execution time compared to FP-Growth approach.

4.3 Memory Usage

We also tested the memory usage by FPMAX, MAFIA and our approach. In many cases we noticed that our approach consumes one order of magnitude less memory than FPMAX and two orders of magnitude less memory than MAFIA. Figure 7 illustrates these results.

5 Conclusion

Most of frequent itemset mining algorithms assume to work on relatively small dataset in comparison to what we are expected to deal with in real applications such as affinity analysis in very large web sites, basket analysis for large department stores, or analysis of tracking data from radio-frequency identification chips on merchandize. Performance analysis on small datasets cannot discriminate between frequent itemset mining algorithms; and when the execution time is in order of few seconds, the selection of algorithms becomes irrelevant. Yet with extremely large databases the issue seems still an open-problem since most of the existing algorithms cannot mine huge databases using either low or high support. In this work we presented YAFIMA, a set of new algorithms for mining all, closed and maximal patterns. These novel algorithms are based on existing data structures FP-tree and COFI-tree. Our contribution is a new way to mine those existing structures using a novel traversal approach. A set of pruning methods can also be implemented to accelerate the discovery process [12]. The idea is to either find the maximal or closed itemsets and expend them to find all frequent patterns. Our performance studies show that our approach can compete with the existing state of the art algorithms in many small datasets (real and synthetic). Even when it was not the winner, the difference was relatively small between its performance and the performance of winner. Yet, when we mine extremely

Algorithm	ALL		CLOSED		MAXIMAL	
	High support winner	Low support winner	High support winner	Low support winner	High support winner	Low support winner
Mushroom	FP-Growth	FP-Growth	FP-CLOSED	FP-CLOSED	FPMAX	FPMAX
Chess	COFI-ALL	COFI-ALL	COFI-CLOSED	COFI-CLOSED	COFI-MAX	COFI-MAX
Pumsb	FP-Growth	MAFIA	FP-CLOSED	FP-CLOSED	GENMAX	COFI-MAX
Accidents	Eclat	FP-Growth	CHARM	FP-CLOSED	GENMAX	FPMAX
T10K5DL12	COFI-ALL	COFI-ALL	COFI-CLOSED	COFI-CLOSED	COFI-MAX	COFI-MAX

Figure 5. Mining different small datasets: The winner algorithms

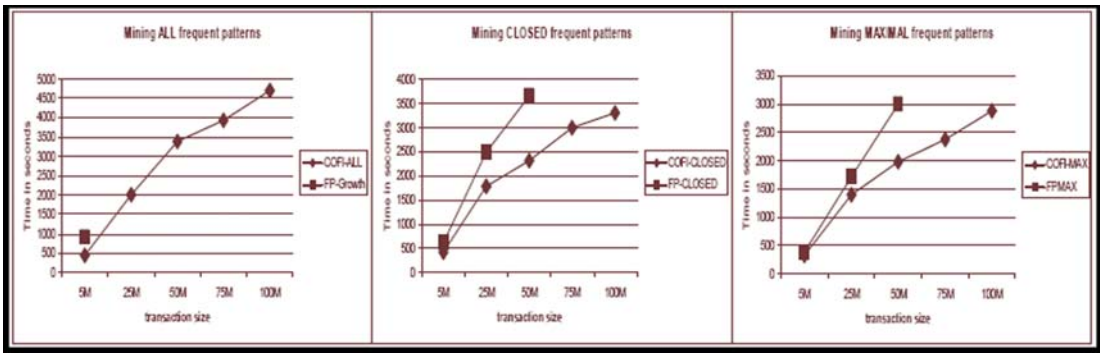

Figure 6. Scalability with very large datasets

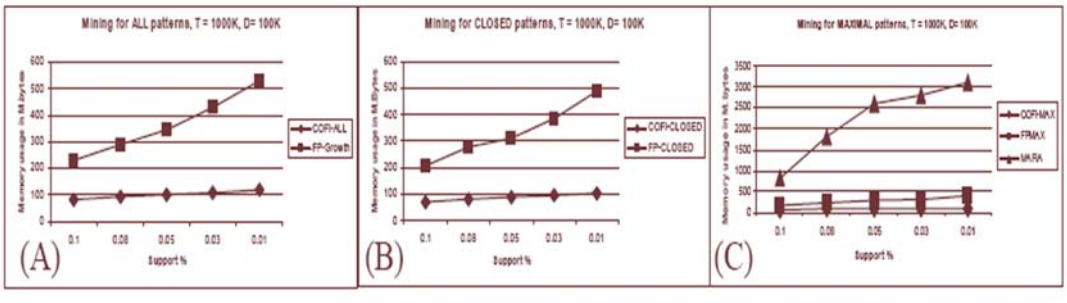

Figure 7. Disparity in memory usage

largedatasets, our performance studies showed that the YAFIMA set was able to mine efficiently 100 million transactions in less than 5000 seconds on a small desktop while other known approaches failed.

References

[1] Agrawal, R. Srikant, R (1994). Fast algorithms for mining association rules. *In: Proc. 1994 Int. Conf. Very Large Data Bases*, pages 487–499, Santiago, Chile.

[2] Burdick, D. Calimlim, M. Gehrke, J (2001). Mafia: A maximal frequent itemset algorithm for transactional databases. *In: ICDE*, 443–452.

[3] El-Hajj, M. Zaïane, O.R (2003). Inverted matrix: Efficient discovery of frequent items in large datasets in the context of interactive mining. *In: Proc. 2003 Int'l Conf. on Data Mining and Knowledge Discovery (ACM SIGKDD)*, August.

[4] El-Hajj, M. Zaïane, O.R (2003). Non recursive generation of frequent k-itemsets from frequent pattern tree representations. *In: Proc. of 5th International Conference on Data Warehousing and Knowledge Discovery (DaWak'2003)*, September.

[5] Goethals, B. Zaki, M (2003). Advances in frequent itemset mining implementations: Introduction to fimi03. *In: Workshop on Frequent Itemset Mining Implementations (FIMI'03) in conjunction with IEEE-ICDM*.

[6] Gouda, K. Zaki, M.J (2001). Efficiently mining maximal frequent itemsets. *In*: *ICDM*, 163–170.

[7] Grahne, G. Zhu, J (2003). Efficiently using prefix-trees in mining frequent itemsets. *In: FIMI'03, Workshop on Frequent Itemset Mining Implementations*, November.

[8] Han, J. Pei, J. Yin, Y. (2000). Mining frequent patterns without candidate generation. *In: 2000 ACM SIGMOD Intl. Conference on Management of Data*, 1–12.

[9] IBM Almaden. Quest synthetic data generation code. http://www.almaden.ibm.com/cs/quest/syndata.html.

[10] Pasquier, N. Bastide, Y. Taouil, R. Lakhal, L (1999). Discovering frequent closed itemsets for association rules. *In: International Conference on Database Theory (ICDT)*, 398–416, January.

[11] Wang, J. Han, J Pei. Closet, J (2003). Searching for the best strategies for mining frequent closed itemsets. *In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'03)*, Washington, DC, USA.

[12] Zaïane, O. ElHajj, M (2005). Pattern Lattice Traversal by Selective Jumps. *In:* Proc. 2005 Int'l Conf. on Knowledge Discovery and Data Mining (ACM SIGKDD) 729-735.

[13] Zaki, M. Hsiao, C.-J (2002). ChARM: An efficient algorithm for closed itemset mining. *In: 2nd SIAM International Conference on Data Mining*, April.