COFI-tree Mining: A New Approach to Pattern Growth with Reduced Candidacy Generation

Mohammad El-Hajj
Department of Computing Science
University of Alberta Edmonton, AB, Canada
mohammad@cs.ualberta.ca

Osmar R. Zaïane
Department of Computing Science
University of Alberta Edmonton, AB, Canada
zaiane@cs.ualberta.ca

Abstract

Existing association rule mining algorithms suffer from many problems when mining massive transactional datasets. Some of these major problems are: (1) the repetitive I/O disk scans, (2) the huge computation involved during the candidacy generation, and (3) the high memory dependency. This paper presents the implementation of our frequent itemset mining algorithm, COFI, *which achieves its efficiency by applying four new ideas. First, it can mine using a compact memory based data structures. Second, for each frequent item assigned, a relatively small independent tree is built summarizing co-occurrences. Third, clever pruning reduces the search space drastically. Finally, a simple and non-recursive mining process reduces the memory requirements as minimum candidacy generation and counting is needed to generate all relevant frequent patterns.*

1 Introduction

Frequent pattern discovery has become a common topic of investigation in the data mining research area. Its main theme is to discover the sets of items that occur together more than a given threshold defined by the decision maker. A well-known application domain that counts on the frequent pattern discovery is the market basket analysis. In most cases when the support threshold is low and the number of frequent patterns "explodes", the discovery of these patterns becomes problematic for reasons such as: high memory dependencies, huge search space, and massive I/O required. However, recently new studies have been proposed to reduce the memory requirements [8], to decrease the I/O dependencies [7], still more promising issues need to be investigated such as pruning techniques to reduce the search space. In this paper we introduce a new method for frequent pattern discovery that is based on the Co-Occurrence Frequent Item tree concept [8, 9]. The new proposed method uses a pruning technique that dramatically saves the memory space. These relatively small trees are constructed based on a memory-based structure called FP-Trees [11]. This data structure is studied in detail in the following sections. In short, we introduced in [8] the COFI-tree stucture and an algorithm to mine it. In [7] we presented a disk based data structure, inverted matrix, that replaces the memory-based FP-tree and scales the interactive frequent pattern mining significantly. Our contributions in this paper are the introduction of a clever pruning technique based on an interesting property drawn from our top-down approach, and some implementation tricks and issues. We included the pruning in the algorithm of building the tree so that the pruning is done on the fly.

1.1 Problem Statement

The problem of mining association rules over market basket analysis was introduced in [2]. The problem consists of finding associations between items or itemsets in transactional data. The data could be retail sales in the form of customer transactions or even medical images [16]. Association rules have been shown to be useful for other applications such as recommender systems, diagnosis, decision support, telecommunication, and even supervised classification [5]. Formally, as defined in [3], the problem 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 $\mathcal{D}$ be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. A unique identifier *TID* is given to each transaction. A transaction T is said to contain X, a set of items in I, if $X \subseteq T$. An *association rule* is an implication of the form "$X \Rightarrow Y$", where $X \subseteq I$, $Y \subseteq I$, and $X \cap Y = \emptyset$. An itemset X is said to be *large* or *frequent* if its *support* s is greater or equal than a given minimum support threshold σ. An itemset X satisfies a constraint C if and only if C(X) is *true*. The rule $X \Rightarrow Y$ has a *support s* in the transaction set $\mathcal{D}$ if *s%* of the transactions in $\mathcal{D}$ contain $X \cup Y$. In other words, the support of the

rule is the probability that X and Y hold together among all the possible presented cases. It is said that the rule $X \Rightarrow Y$ holds in the transaction set $\mathcal{D}$ with *confidence c* if *c%* of transactions in $\mathcal{D}$ that contain X also contain Y. In other words, the confidence of the rule is the conditional probability that the consequent Y is true under the condition of the antecedent X. The problem of discovering all association rules from a set of transactions $\mathcal{D}$ consists of generating the rules that have a *support* and *confidence* greater than a given threshold. These rules are called *strong rules*. This association-mining task can be broken into two steps:
1. A step for finding all frequent k-itemsets known for its extreme I/O scan expense, and the massive computational costs;
2. A straightforward step for generating strong rules.

In this paper and our attached code, we focus exclusively on the first step: generating frequent itemsets.

1.2 Related Work

Several algorithms have been proposed in the literature to address the problem of mining association rules [12, 10]. One of the key algorithms, which seems to be the most popular in many applications for enumerating frequent itemsets, is the *apriori* algorithm [3]. This *apriori* algorithm also forms the foundation of most known algorithms. It uses an *anti-monotone* property stating that for a k-itemset to be frequent, all its (k-1)-itemsets have to be frequent. The use of this fundamental property reduces the computational cost of candidate frequent itemset generation. However, in the cases of extremely large input sets with big frequent 1-items set, the *Apriori* algorithm still suffers from two main problems of repeated I/O scanning and high computational cost. One major hurdle observed with most real datasets is the sheer size of the candidate frequent 2-itemsets and 3-itemsets.

TreeProjection is an efficient algorithm presented in [1]. This algorithm builds a lexicographic tree in which each node of this tree presents a frequent pattern. The authors report that their algorithm is one order of magnitude faster than the existing techniques in the literature. Another innovative approach of discovering frequent patterns in transactional databases, FP-Growth, was proposed by Han et al. in [11]. This algorithm creates a compact tree-structure, FP-Tree, representing frequent patterns, that alleviates the multi-scan problem and improves the candidate itemset generation. The algorithm requires only two full I/O scans of the dataset to build the prefix tree in main memory and then mines directly this structure. The authors of this algorithm report that their algorithm is faster than the *Apriori* and the TreeProjection algorithms. Mining the FP-tree structure is done recursively by building conditional trees that are of the same order of magnitude in number as the frequent patterns. This massive creation of conditional trees makes this algorithm not scalable to mine large datasets beyond few millions. In [14] the same authors propose a new algorithm, H-mine, that invokes FP-Tree to mine condensed data. This algorithm is still not scalable as reported by its authors in [13].

1.3 Preliminaries, Motivations and Contributions

The Co-Occurrence Frequent Item tree (or COFI-tree for short) and the *COFI* algorithm presented in this paper are based on our previous work in [7, 8]. The main motivation of our current research is the pruning technique that reduces the memory space needed by the COFI-trees. The presented algorithm is done in two phases in which phase 1 requires two full I/O scans of the transactional database to build the FP-Tree structure[11]. The second phase starts by building small Co-Occurrence Frequent trees for each frequent item. These trees are pruned first to eliminate any non-frequent items with respect to the COFI-tree based frequent item. Finally the mining process is executed.

The remainder of this paper is organized as follows: Section 2 describes the Frequent Pattern tree, design and construction. Section 3 illustrates the design, constructions, pruning, and mining of the Co-Occurrence Frequent Item trees. Section 4 presents the implementation procedure of this algorithm. Experimental results are given in Section 5. Finally, Section 6 concludes by discussing some issues and highlighting our future work.

2 Frequent Pattern Tree: Design and Construction

The COFI-tree approach we propose consists of two main stages. Stage one is the construction of a modified Frequent Pattern tree. Stage two is the repetitive building of small data structures, the actual mining for these data structures, and their release.

2.1 Construction of the Frequent Pattern Tree

The goal of this stage is to build the compact data structure called Frequent Pattern Tree [11]. This construction is done in two phases, where each phase requires a full I/O scan of the dataset. A first initial scan of the database identifies the frequent 1-itemsets. The goal is to generate an ordered list of frequent items that would be used when building the tree in the second phase.

This phase starts by enumerating the items appearing in the transactions. After enumeration these items (i.e. after reading the whole dataset), infrequent items with a support less than the support threshold are weeded out and the remaining frequent items are sorted by their frequency. This

Table 1. Transactional database

T.No.	Items				
T1	A	G	D	C	B
T2	B	C	H	E	D
T3	B	D	E	A	M
T4	C	E	F	A	N
T5	A	B	N	O	P
T6	A	C	Q	R	G
T7	A	C	H	I	G
T8	L	E	F	K	B
T9	A	F	M	N	O
T10	C	F	P	G	R
T11	A	D	B	H	I
T12	D	E	B	K	L
T13	M	D	C	G	O
T14	C	F	P	Q	J
T15	B	D	E	F	I
T16	J	E	B	A	D
T17	A	K	E	F	C
T18	C	D	L	B	A

list is organized in a table, called header table, where the items and their respective support are stored along with pointers to the first occurrence of the item in the frequent pattern tree. Phase 2 would construct a frequent pattern tree.

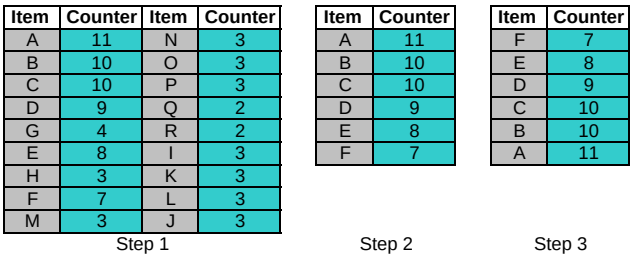


Figure 1. Steps of phase 1

Phase 2 of constructing the Frequent Pattern tree structure is the actual building of this compact tree. This phase requires a second complete I/O scan from the dataset. For each transaction read, only the set of frequent items present in the header table is collected and sorted in descending order according to their frequency. These sorted transaction items are used in constructing the FP-Trees as follows: for the first item on the sorted transactional dataset, check if it exists as one of the children of the root. If it exists then increment the support for this node. Otherwise, add a new node for this item as a child for the root node with 1 as support. Then, consider the current item node as the new temporary root and repeat the same procedure with the next item on the sorted transaction. During the process of adding any new item-node to the FP-Tree, a link is maintained between this item-node in the tree and its entry in the header table. The header table holds as one pointer per item that points to the first occurrences of this item in the FP-Tree structure.

2.2 Illustrative Example

For illustration, we use an example with the transactions shown in Table 1. Let the minimum support threshold be set to 4. Phase 1 starts by accumulating the support for all items that occur in the transactions. Step 2 of phase 1 removes all non-frequent items, in our example (G, H, I, J, K, L,M, N, O, P, Q and R), leaving only the frequent items (A, B, C, D, E, and F). Finally all frequent items are sorted according to their support to generate the sorted frequent 1-itemset. This last step ends phase 1 in Figure 1 of the COFI-tree algorithm and starts the second phase. In phase 2, the first transaction (A, G, D, C, B) is filtered to consider only the frequent items that occur in the header table (i.e. A, D, C and B). This frequent list is sorted according to the items' supports (A, B, C and D). This ordered transaction generates the first path of the FP-Tree with all item-node support initially equal to 1. A link is established between each item-node in the tree and its corresponding item entry in the header table. The same procedure is executed for the second transaction (B, C, H, E, and D), which yields a sorted frequent item list (B, C, D, E) that forms the second path of the FP-Tree. Transaction 3 (B, D, E, A, and M) yields the sorted frequent item list (A, B, D, E) that shares the same prefix (A, B) with an existing path on the tree. Item-nodes (A and B) support is incremented by 1 making the support of (A) and (B) equal to 2 and a new sub-path is created with the remaining items on the list (D, E) all with support equal to 1. The same process occurs for all transactions until we build the FP-Tree for the transactions given in Table 1. Figure 2 shows the result of the tree building process. Notice that in our tree structure, contrary to the original FP-tree [11], our links are bi-directional. This, and other differences presented later, are used by our mining algorithm.

3 Co-Occurrence Frequent-Item-trees: Construction, Pruning and Mining

Our approach for computing frequencies relies first on building independent, relatively small trees for each frequent item in the header table of the FP-Tree called COFI-trees. A pruning technique is applied to remove all non-frequent items with respect to the main frequent item of the tested COFI-tree. Then we mine separately each one of the trees as soon as they are built, minimizing the candidacy generation and without building conditional sub-trees recursively. The trees are discarded as soon as mined. At

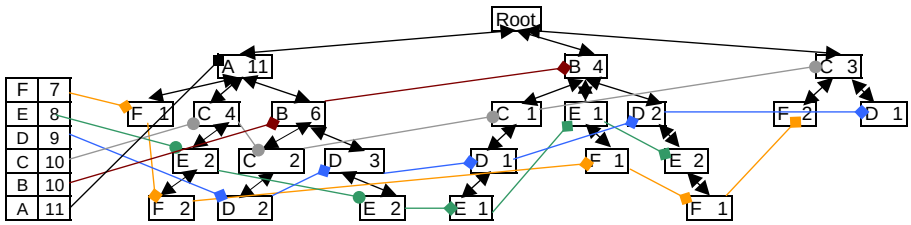


Figure 2. Frequent Pattern Tree.

any given time, only one COFI-tree is present in main memory. In our following examples we always assume that we are building the COFI-trees based on the modified FP-Tree data-structure presented above.

3.1 Pruning the COFI-trees

Pruning can be done after building a tree or, even better, while building it. We opted for pruning on the fly since the overhead is minimal but the consequences are drastic reduction in memory requirements. We will discuss the pruning idea, then present the building algorithm that considers the pruning on the fly.

In this section we are introducing a new *anti-monotone* property called global frequent/local non-frequent property. This property is similar to the *Apriori* one in the sense that it eliminates at the i^{th} level all non-frequent items that will not participate in the (i+1) level of candidate itemsets generation. The difference between the two properties is that we extended our property to eliminate also frequent items which are among the i-itemset and we are sure that they will not participate in the (i+1) candidate set. The *Apriori* property states that *all nonempty subsets of a frequent itemset must also be frequent*. An example is given later in this section to illustrate both properties. In our approach, we are trying to find all frequent patterns with respect to one frequent item, which is the base item of the tested COFI-tree. We already know that all items that participate in the creation of the COFI-tree are frequent with respect to the global transaction database, but that does not mean that they are also locally frequent with respect to the based item in the COFI-tree. The global frequent/local non-frequent property states that *all nonempty subsets of a frequent itemset with respect to the item A of the A-COFI-tree , must also be frequent with respect to item A*. For each frequent item A we traverse the FP-Tree to find all frequent items that occur with A in at least one transaction (or branch in the FP-Tree) with their number of occurrences. All items that are locally frequent with item A will participate in building the A-COFI-tree, other global frequent items, locally non-frequent items will not participate in the creation of the A-COFI-tree. In our example we can find that all items that participate in the creation of the F-COFI-tree are locally not frequent with respect to item F as the support for all these items are not greater than the support threshold σ which is equal to 4, Figure 3. From knowing this, there will be no need to mine the F-COFI-tree, we already know that no frequent patterns other than the item F will be generated. We can extend our knowledge at this stage to know that item F will not appear in any of the frequent patterns. The COFI-tree for item E indicates that only items D, and B are frequent with respect to item E, which means that there will be no need to test patterns as EC, and EA. The COFI-tree for item D indicates that item C will be eliminated, as it is not frequent with respect to item D. C-COFI-tree ignores item B for the same reason. To sum up the *Apriori* property states in our example of 6 1-frequent itemset that we need to generate 15 2-Candidate itemset which are (A,B), (A,C), (A,D), (A,E), (A,F), (B,C), (B,D), (B,E), (B,F), (C,D), (C,E), (C,F), (D,E), (D,F), (E,F), using our property we have eliminated (not generated or counted) 9 patterns which are (A,E), (A,F), (B,C), (B,F), (C,D), (C,E), (C,F), (D,F), (E,F) leaving only 6 patterns to test which are (A,B), (A,C), (A,D), (B,D), (B,E), (D,E).

3.2 Construction of the Co-Occurrence Frequent-Item-trees

The small COFI-trees we build are similar to the conditional FP-Trees [11] in general in the sense that they have a header with ordered frequent items and horizontal pointers pointing to a succession of nodes containing the same frequent item, and the prefix tree per se with paths representing sub-transactions. However, the COFI-trees have bi-directional links in the tree allowing bottom-up scanning as well, and the nodes contain not only the item label and a frequency counter, but also a participation counter as explained later in this section. The COFI-tree for a given frequent item x contains only nodes labeled with items that are more frequent or as frequent as x.

To illustrate the idea of the COFI-trees, we will explain step by step the process of creating COFI-trees for the FP-Tree of Figure 2. With our example, the first Co-Occurrence Frequent Item tree is built for item F as it is the least frequent item in the header table. In this tree for F, all frequent items, which are more frequent than F, and share transac-

tions with F, participate in building the tree. This can be found by following the chain of item F in the FP-Tree structure. The F-COFI-tree starts with the root node containing the item in question, then a scan of part of the FP-Tree is applied following he chain of the F item in the FP-Tree. The first branch FA has frequency of 1, as the frequency of the branch is the frequency of the test item, which is F. The goal of this traversal is to count the frequency of each frequent item with respect to item F. By doing so we can find that item E occurs 4 times, D occurs 2 times, C occurs 4 times, B 2 times, and A 3 times, by applying the *anti-monotone* constraint property we can predict that item F will never appear in any frequent pattern except itself. Consequently there will be no need to continue building the F-COFI-tree.

The next frequent item to test is E. The same process is done to compute the frequency of each frequent items with respect to item E. From this we can find that only two globally frequent items are also locally frequent which are (D:5 and B:6). For each sub-transaction or branch in the FP-Tree containing item E with other locally frequent items that are more frequent than E which are parent nodes of E, a branch is formed starting from the root node E. the support of this branch is equal to the support of the E node in its corresponding branch in FP-Tree. If multiple frequent items share the same prefix, they are merged into one branch and a counter for each node of the tree is adjusted accordingly. Figure 3 illustrates all COFI-trees for frequent items of Figure 2. In Figure 3, the rectangle nodes are nodes from the tree with an item label and two counters. The first counter is a *support-count* for that node while the second counter, called *participation-count*, is initialized to 0 and is used by the mining algorithm discussed later, a horizontal link which points to the next node that has the same *item-name* in the tree, and a bi-directional vertical link that links a child node with its parent and a parent with its child. The bi-directional pointers facilitate the mining process by making the traversal of the tree easier. The squares are actually cells from the header table as with the FP-Tree. This is a list made of all frequent items that participate in building the tree structure sorted in ascending order of their global support. Each entry in this list contains the *item-name*, *item-counter*, and a *pointer* to the first node in the tree that has the same *item-name*.

To explain the COFI-tree building process, we will highlight the building steps for the E-COFI-tree in Figure 3. Frequent item E is read from the header table and its first location in the FP-Tree is located using the pointer in the header table. The first location of item E indicate that it shares a branch with items CA, with support = 2, since none of these items are locally frequent then only the support of the E root node is incremented by 2. the second node of item E indicates that it shares items DBA with support equals to 2 for this branch as the support of the E-item is considered the

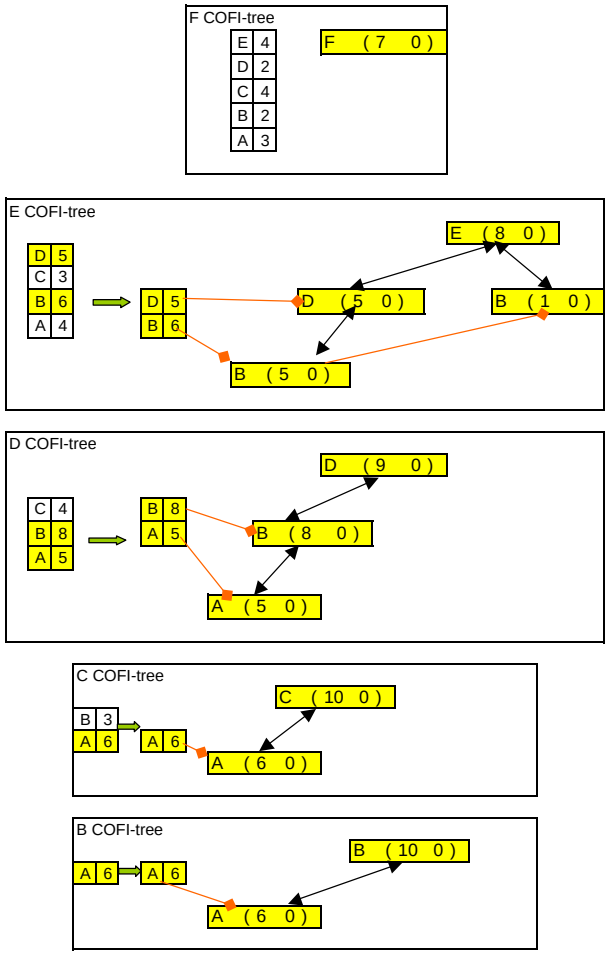


Figure 3. COFI-trees

support for this branch (following the upper links for this item). Two nodes are created, for items D and B with support equals to 2, D is a child node of B, and B is a child node of E. The third location of E indicate having EDB:1, which shares an existing branch in the E-COFI-tree, all counters are adjusted accordingly. A new branch of EB: 1 is created as the support of E=1 for the fourth occurrences of E. The final occurrence EDB: 2 uses an existing branch and only counters are adjusted. Like with FP-Trees, the header constitutes a list of all frequent items to maintain the location of first entry for each item in the COFI-tree. A link is also made for each node in the tree that points to the next location of the same item in the tree if it exists. The mining process is the last step done on the E-COFI-tree before removing it and creating the next COFI-tree for the next item in the header table.

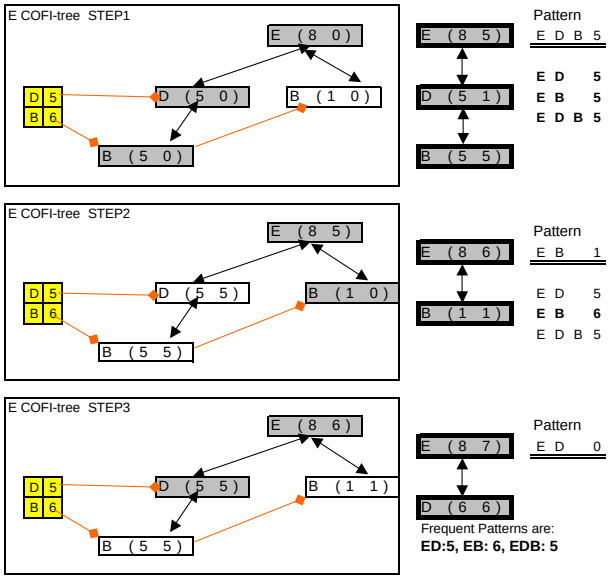


Figure 4. Steps needed to generate frequent patterns related to item E

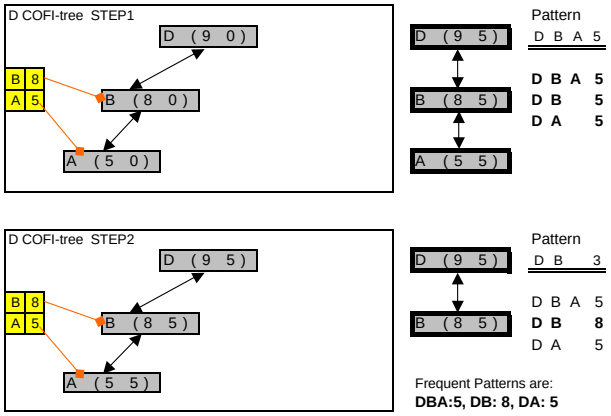


Figure 5. Steps needed to generate frequent patterns related to item D

3.3 Mining the COFI-trees

The COFI-trees of all frequent items are not constructed together. Each tree is built, mined, then discarded before the next COFI-tree is built. The mining process is done for each tree independently with the purpose of finding all frequent *k*-itemset patterns in which the item on the root of the tree participates.

Steps to produce frequent patterns related to the E item for example, as the F-COFI-tree will not be mined based on the pruning results we found on the previous step, are illustrated in Figure 4. From each branch of the tree, using the *support-count* and the *participation-count*, candidate frequent patterns are identified and stored temporarily in a list. The non-frequent ones are discarded at the end when all branches are processed. The mining process for the E-COFI-tree starts from the most locally frequent item in the header table of the tree, which is item B. Item B exists in two branches in the E-COFI-tree which are (B:5, D:5 and E:8), and (B:1, and E:8). The frequency of each branch is the frequency of the first item in the branch minus the participation value of the same node. Item B in the first branch has a frequency value of 5 and participation value of 0 which makes the first pattern EDB frequency equals to 5. The participation values for all nodes in this branch are incremented by 5, which is the frequency of this pattern. In the first pattern EDB: 5. We need to generate all sub-patterns that item E participates in, which are ED: 5, EB: 5, and EDB: 5. The second branch that has B generates the pattern EB: 1. EB already exists and its counter is adjusted to become 6. The COFI-tree of Item E can be removed at this time and another tree can be generated and tested to produce all the frequent patterns related to the root node. The same process is executed to generate the frequent patterns. The D-COFI-tree (Figure 5) is created after the E-COFI-tree. Mining this tree generates the following frequent patterns: DBA: 5, DA: 5, and DB:8. The same process occurs for the remaining trees that would produce AC: 6 for the C-COFI-tree and BA:6 for the B-COFI-tree.

The following is our algorithm for building and mining the COFI-trees with pruning.

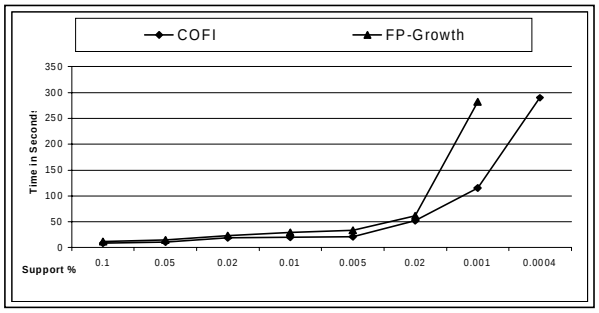
Algorithm COFI: Creating with pruning and Mining COFI-trees
Input: modified FP-Tree, a minimum support threshold σ
Output: Full set of frequent patterns
Method:
1. A = the least frequent item on the header table of FP-Tree
2. While (There are still frequent items) do
   2.1 count the frequency of all items that share item (A) a path. Frequency of all items that share the same path are the same as of the frequency of the (A) items
   2.2 Remove all non-locally frequent items for the frequent list of item (A)
   2.3 Create a root node for the (A)-COFI-tree with both *frequency-count* and *participation-count* = 0
      2.3.1 C is the path of locally frequent items in the path of item A to the root
      2.3.2 Items on C form a prefix of the (A)-COFI-tree.
      2.3.3 If the prefix is new then Set *frequency-count*= frequency of (A) node and *participation-count*= 0 for all nodes in the path
      Else

2.3.4 Adjust the *frequency-count* of the already exist part of the path.
2.3.5 Adjust the pointers of the *Header list* if needed
2.3.6 find the next node for item A in the FP-tree and go to 2.3.1
2.4 MineCOFI-tree (A)
2.5 Release (A) COFI-tree
2.6 A = next frequent item from the header table
3. Goto 2

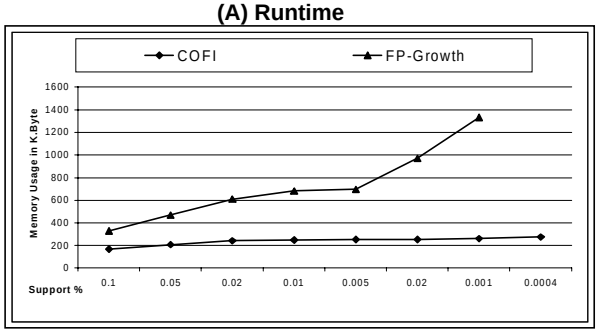
Function: MineCOFI-tree (A)
1. nodeA = select_next_node //Selection of nodes starts with the node of most locally frequent item and following its chain, then the next less frequent item with its chain, until we reach the least frequent item in the *Header list* of the (A)-COFI-tree
2. while there are still nodes do
2.1 D = set of nodes from nodeA to the root
2.2 F = nodeA.*frequency-count*-nodeA.*participation-count*
2.3 Generate all Candidate patterns X from items in D. Patterns that do not have A will be discarded.
2.4 Patterns in X that do not exist in the A-Candidate List will be added to it with frequency = F otherwise just increment their frequency with F
2.5 Increment the value of *participation-count* by F for all items in D
2.6 nodeA = select_next_node
3. Goto 2
4. Based on support threshold σ remove non-frequent patterns from A Candidate List.

4 Experimental Studies

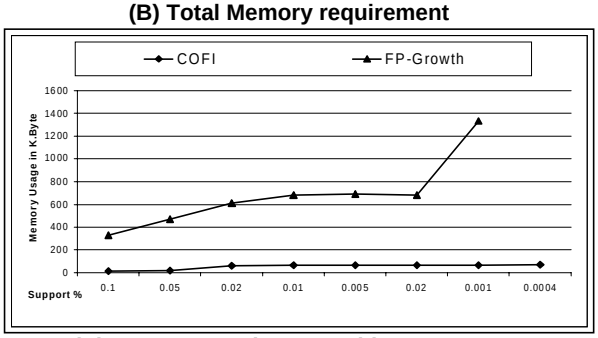
To study the COFI-tree mining strategies we have conducted several experiments on a variety of data sizes comparing our approach with the well-known FP-Growth [11] algorithm written by its original authors. The experiments were conducted on 2.6 GHz CPU machine with 2 Gbytes of memory using Win2000 operating system. Transactions were generated using IBM synthetic data generator [4]. We have conducted several types of experiments to test the effect of changing the support, transaction size, dimension, and transaction length. The first set of experiments were tested on a transaction database of 500K transactions, 10K the dimension, and the average transaction length was 12. We have varied the support from absolute value of 500 to 2 in which frequent patterns generated varied from 15K to 3400K patterns. FP-Growth could not mine the last experiment in this set as it used all available memory space. In all experiments the COFI-tree approach outperforms the FP-Growth approach. The major accomplishment of our approach is in the memory space saved. Our algorithm outperforms the FP-Growth by one order of magnitude in terms of memory space requirements. We have also tested the memory space used during the mining process only, (i.e, isolating the memory space used to create the FP-Tree by both FP-growth and COFI-tree FP-Tree based algorithms). We have found also that the COFI-tree approach outperforms the FP-tree by one order of magnitude in terms of memory space used by the COFI-tree compared with the conditional trees used by FP-Growth during the mining process. Figure 6A presents the time needed to mine 500K transactions using different support levels. Figure 6B depicts the memory needed during the mining process of the previous experiments. Figure 6C illustrates the memory needed by



(A) Runtime



(B) Total Memory requirement



(C) Memory requirement without FP-tree

No.of transactions = 500K, Dimension= 10K,
Average no. of items / transaction = 12

Figure 6. Mining dataset of 500K transactions

Table 2. Time and Memory Scalability with respect to support on the T10I4D100K dataset

	Time in Seconds		Memory in KB	
Support %	COFI	FP-Growth	COFI	FP-Growth
0.50	1.5	3.0	18	173
0.25	1.7	5.2	19	285
0.10	2.7	12.3	26	289
0.05	14.0	20.9	19	403

the COFI-trees and Conditional trees during the mining process. Other experiments were conducted to test the effect of changing the dimension, transaction size, transaction length using the same support which is 0.05%. Some of these experiments are represented in Figure 7. Figures 7A and 7B represent the time needed during the mining process. Figures 7C and 7D represent the memory space needed during the whole mining process. Figures 7E and 7F represent the memory space needed by the COFI-trees or conditional trees during the mining process. In these experiments we have varied the dimension, which is the number of distinct items from 5K to 10K, the average transaction length from 12 to 24 items in one transaction, and the number of transactions from 10K to 500K. All these experiments depicted the fact that our approach is one order of magnitude better than the FP-Growth approach in terms of memory usage.

We also run experiments using the public UCI datasets provided on the FIMI workshop website, which are Mushroom, Chess, Connect, Pumsb, T40I10D100K, and T10I4D100K. The COFI algorithm scales relatively well vis-à-vis the support threshold with these datasets. Results are not reported here for lack of space. Our approach revealed good results with high support value on all datasets. However, like with other approaches, in cases of low support value, where the number of frequent patterns increases significantly, our approach faces some difficulties. For such cases it is recommended to consider discovering closed itemsets or maximal patterns instead of just frequent itemsets. The sheer number of frequent itemsets becomes overwhelming, and some argue even useless. Closed itemsets and maximal itemsets represent all frequent patterns by eliminating the redundant ones. For illustration, Table 2 compares the CPU time and memory requirement for COFI and FP-Growth on the T10I4D100K dataset.

5 Implementations

The COFI-tree program submitted with this paper is a C++ code. The executable of this code runs with 3 parameters, which are: (1) the path to the input file name. (2) a positive integer that presents the absolute support. (3) An optional file name for the out patterns. This code generates ALL frequent patterns from the provided input file. The code scans the database twice. The goal of the first database scan is to find the frequency of each item in this transactional database. These frequencies are stored in a data structure called Candidate-Items. Each entry of this candidate items is a structure called ItemsStructure that is made of two long integers representing the item and its frequency. All frequent items are then stored in a special data structure called F1-Items. This data structure is sorted in descending order based on the frequency of each item. To access the location of each item we map it with a specific location using a new data structure called FindInHashTable. In brief, since we do not know the number of unique items at runtime, and thus can't create an array for counting the items, rather than having a linked list of items, we create blocks of p items. The number p could arbitrarily be 100 or 1000. Indeed, following links in a linked list each time to find and increment a counter could be expensive. Instead, blocs of items are easily indexed. In the worst case, we could lose the space of $p - 1$ unused items.

The second scan starts by eliminating all non frequent items from each transaction read and then sort this transaction based on the frequency of each frequent item. This process occurred in the Sort-Transaction method. The FP-tree is built based on the sub-transaction made of the frequent items. The FP-tree data structure is a tree of n children. The structure struct FPTTree { long Element; long counter; FPTTree* child; FPTTree* brother; FPTTree* father; FPTTree* next; } has been used to create each node of this tree, where a link is created between each node and its first child, and the brother link is maintained to create a linked list of all children of the same node. This linked list is built ordered based on the frequency of each item. The header list is maintained using the structure FrequentStruc { long Item; long Frequency; long COFIFrequency; long COFIFrequency1; FPTTree* first; COFITree* firstCOFI; }; After building the FP-tree we start building the first COFI-tree by selecting the item with least frequency from the frequent list. A scan is made of the FP-tree starting from the linked list of this item to find the frequency of other items with respect to this item. After that, the COFI-tree is created based on only the locally frequent items. Finally frequent patterns are generated and stored in the FrequentTree data structure. All nodes that have support greater or equal than the given support present a frequent pattern. The COFI-tree and the FrequentTree are removed from memory and the next COFI-tree is created until we mine all frequent trees.

One interesting implementation improvement is the fact that the participation counter was also added to the header table of the COFI-tree this counter cumulates the participation of the item in all paterns already discovered in the current COFI-tree. The difference between the participa-

tion in the node and the participation in the header is that the counter in the node counts the participation of the node item in all paths where the node appears, while the new counter in the COFI-tree header counts the participation of the item globally in the tree. This trick does not compromise the effectiveness and usefulness of the participation counting. One main advantage of this counter is that it looks ahead to see if all nodes of a specific item have already been traversed or not to reduce the unneeded scans of the COFI-tree.

6 Conclusion and future work

The COFI algorithm, based on our COFI-tree structure, we propose in this paper is one order of magnitude better than the FP-Growth algorithm in terms of memory usage, and sometimes in terms of speed. This Algorithm achieves this results thanks to: (1) the non recursive technique used during the mining process, in which with a simple traversal of the COFI-tree a full set of frequent patterns can be generated. (2) The pruning method that is used to remove all locally non frequent patterns, leaving the COFI-tree with only locally frequent items.

The major advantage of our algorithm *COFI* over FP-Growth is that it needs a significantly smaller memory footprint, and thus can mine larger transactional databases with smaller main memory available. The fundamental difference, is that COFI tries to find a compromise between a fully pattern growth approach, that FP-Growth adopts, and a total candidacy generation approach that apriori is known for. COFI grows targeted patterns but performs a reduced and focused generation of candidates during the mining. This is to avoid the recursion that FP-growth uses, and notorious to blow the stack with large datasets.

We have developed algorithms for closed itemset mining and maximal itemset mining based on our COFI-tree approach. However, their efficient implementations were not ready by the deadline of this workshop. These efficient algorithms and experimental results will be compared to existing algorithms such as CHARM[17], MAFIA[6] and CLOSET+[15], and will be reported in the future.

7 Acknowledgments

This research is partially supported by a Research Grant from NSERC, Canada.

References

[1] R. Agarwal, C.Aggarwal, and V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Parallel and distributed Computing*, 2000.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data*, pages 207–216, Washington, D.C., May 1993.

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

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

[5] M.-L. Antonie and O. R. Zaïane. Text document categorization by term association. In *IEEE International Conference on Data Mining*, pages 19–26, December 2002.

[6] C. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *IEEE International Conference on Data Mining (ICDM 01)*, April 2001.

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

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

[9] M. El-Hajj and O. R. Zaïane. Parallel association rule mining with minimum inter-processor communication. In *Fifth International Workshop on Parallel and Distributed Databases (PaDD'2003) in conjunction with the 14th Int' Conf. on Database and Expert Systems Applications DEXA2003*, September 2003.

[10] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufman, San Francisco, CA, 2001.

[11] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM-SIGMOD*, Dallas, 2000.

[12] J. Hipp, U. Guntzer, and G. Nakaeizadeh. Algorithms for association rule mining - a general survey and comparison. *ACM SIGKDD Explorations*, 2(1):58–64, June 2000.

[13] J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by oppotunistic projection. In *Eight ACM SIGKDD Internationa Conf. on Knowledge Discovery and Data Mining*, pages 229–238, Edmonton, Alberta, August 2002.

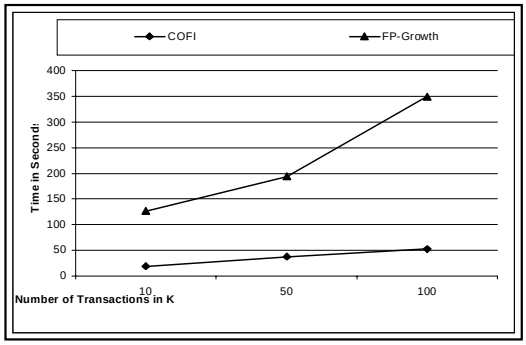
[14] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *ICDM*, pages 441–448, 2001.

[15] J. Wang, J. Han, and J. Pei. CLOSET+: Searching for the best strategies for mining frequent closed itemsets. In *9th ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining*, July 2003.

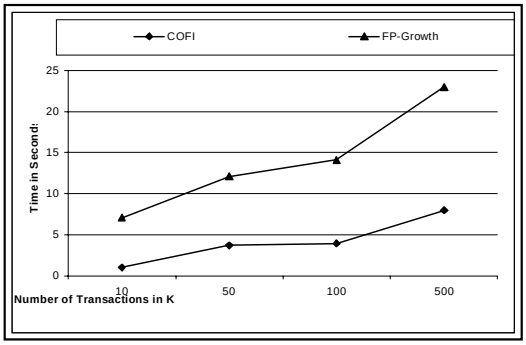
[16] O. R. Zaïane, J. Han, and H. Zhu. Mining recurrent items in multimedia with progressive resolution refinement. In *Int. Conf. on Data Engineering (ICDE'2000)*, pages 461–470, San Diego, CA, February 2000.

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

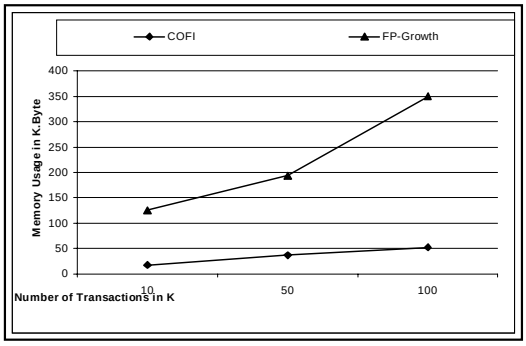
D = Dimension, L = Average number of items in one transaction
Support = 0.05%



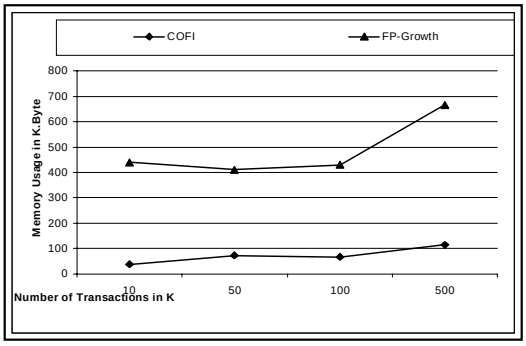
(A) D=5K, L=12



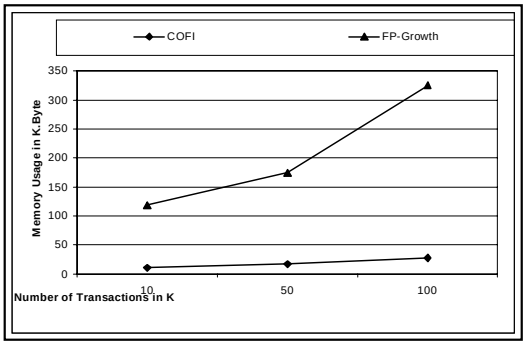
(B) D=10K, L=24



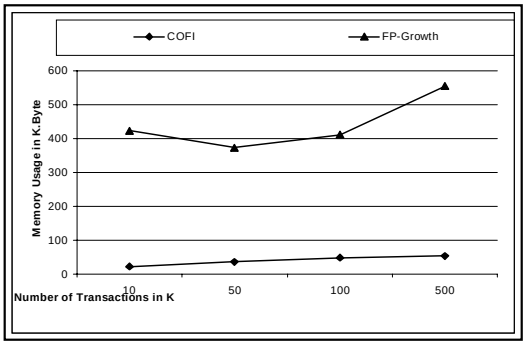
(C) D=5K, L=12



(D) D=10K, L=24



(E) D=5K, L=12



(F) D=10K, L=24

Figure 7. Mining dataset of different sizes