American Journal of Economics and Business Administration 3 (1): 58-65, 2011
ISSN 1945-5488

Using Unique-Prime-Factorization Theorem to Mine Frequent Patterns without Generating Tree

Hossein Tohidi and Hamidah Ibrahim
Department of Computer Science
Faculty of Computer Science and Information Technology,
University Putra Malaysia Serdang, Malaysia

Abstract: Problem statement: Ffrequent patterns are patterns that appear in a data set frequently. Finding such frequent patterns plays an essential role in mining associations, correlations and many other interesting relationships among data. **Approach:** Most of the previous studies adopt an Apriori-like approach. For huge database it may need to generate a huge number of candidate sets. An interest solution is to design an approach that without generating candidate is able to mine frequent patterns. **Results:** An interesting method to frequent pattern mining without generating candidate pattern is called frequent-pattern growth, or simply FP-growth, which adopts a divide-and-conquer strategy as follows. However, for a large database, constructing a large tree in the memory is a time consuming task and increase the time of execution. In this study we introduce an algorithm to generate frequent patterns without generating a tree and therefore improve the time complexity and memory complexity as well. Our algorithm works based on prime factorization and is called Prime Factor Miner (PFM). **Conclusion/Recommendations:** This algorithm is able to achieve low memory order at O(1) which is significantly better than FP-growth.

Key words: Data mining, frequent pattern mining, association rule mining

INTRODUCTION

Frequent patterns are patterns (such as itemsets, subsequences, or substructures) that appear in a data set frequently. For example, a set of items, such as milk and bread that appear frequently together in a transaction data set is a frequent itemset. A subsequence, such as buying first a PC, then a digital camera and then a memory card, if it occurs frequently in a shopping history database, is a (frequent) sequential pattern. Finding such frequent patterns plays an essential role in mining associations, correlations and many other interesting relationships among data. Moreover, it helps in data classification, clustering and other data mining tasks as well. Thus, frequent pattern mining has become an important data mining task and a focused theme in data mining research (Patel *et al*., 2005; Ren *et al*., 2006; Verkhovsky, 2009; Zubair Rahman and Balasubramanie, 2008).

Frequent itemset mining leads to the discovery of associations and correlations among items in large transactional or relational data sets. With massive amounts of data continuously being collected and stored, many industries are becoming interested in mining such patterns from their databases. The discovery of interesting correlation relationships among huge amounts of business transaction records can help in many business decision-making processes, such as catalog design, cross-marketing and customer shopping behaviour analysis.

A typical example of frequent itemset mining is market basket analysis. This process analyzes customer buying habits by finding associations between the different items that customers place in their "shopping baskets". The discovery of such associations can help retailers develop marketing strategies by gaining insight into which items are frequently purchased together by customers. For instance, if customers are buying milk, how likely are they to also buy bread (and what kind of bread) on the same trip to the supermarket? Such information can lead to increased sales by helping retailers do selective marketing and plan their shelf space.

MATERIALS AND METHODS

Most of the previous studies adopt an Apriori-like approach, which is based on the anti-monotone Apriori heuristic: "If any length k pattern is not frequent in the database, its length (k+1) super-pattern can never be frequent".

Corresponding Author: Hossein Tohidi, Faculty of Computer Science and Information Technology, University Putra Malaysia Serdang, Malaysia

The essential idea is to iteratively generate the set of candidate patterns of length (k+1) from the set of frequent-patterns of length k (for k≥1) and check their corresponding occurrence frequencies in the database. The Apriori candidate generate-and-test method significantly reduces the size of candidate sets, leading to good performance gain. However, it suffers from two nontrivial costs:

- It may need to generate a huge number of candidate sets. For example, if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 candidate 2-itemsets. Moreover, to discover a frequent pattern of size 100, such as $\{a_1 \ldots a_{100}\}$, it has to generate at least $2^{100}-1 \sim 10^{30}$ candidates in total
- It may need to repeatedly scan the database and check a large set of candidates by pattern matching. It is costly to go over each transaction in the database to determine the support of the candidate itemsets

Can we design a method that mine the complete set of frequent itemsets without candidate generation? An interesting method in this attempt is called frequent-pattern growth, or simply FP-growth, which adopts a divide-and-conquer strategy as follows. First, it compresses the database representing frequent items into a frequent-pattern tree, or FP-tree, which retains the itemset association information. It then divides the compressed database into a set of conditional databases (a special kind of projected database), each associated with one frequent item or "pattern fragment," and mines each such database separately. The FP-growth method transforms the problem of finding long frequent patterns to searching for shorter ones recursively and then concatenating the suffix.

When the database is large, it is sometimes unrealistic to construct a main memory based FP-tree. An interesting alternative is to first partition the database into a set of projected databases and then construct an FP-tree and mine it in each projected database. Such a process can be recursively applied to any projected database if its FP-tree still cannot fit in main memory. A study on the performance of the FP-growth method shows that it is efficient and scalable for mining both long and short frequent patterns and is about an order of magnitude faster than the Apriori algorithm. It is also faster than a tree-projection algorithm, which recursively projects a database into a tree of projected databases.

FP-growth uses the least frequent items as a suffix, offering good selectivity. The method substantially reduces the search costs. For the given suffix like "I_j" FP-growth finds all possible prefix for "I_j" which their support count is greater than minimum support count. But for this operation a tree must be created and updated which for large database it needs high amount of memory.

This study is to design an approach for the frequent pattern mining without candidate generation which is efficient and fast even for large database. The most significant benefit of this approach is low memory complexity as compared to FP-growth. Our approach called Prime Factor Miner (PFM) is similar to FP-growth where the least frequent item is candidate as a suffix then all frequent patterns which end with the given suffix are generated. The PFM is based on the prime factorization from the number theory and does not require the creation of a tree structure.

This study is organized as follows. First the related study is presented and the FP-growth algorithm is discussed and explained by an example. After that our proposed approach is presented while result section presents the result and discuss about time and memory complexity. Discussion and conclusion are given in the final sections.

Related study: We have categorized previous studies into two parts. The first part focuses on the FP-growth algorithm and explains the algorithm through example while the second part focuses on some previous works related to this study.

FP-growth algorithm: For this part we examine the FP-growth algorithm over a hypothetical dataset for a sailing company. This example is picked up from the textbook Data-Mining Concepts and Techniques (Han and Kamber, 2006). The dataset is a collection of transaction records. Each transaction has a unique ID and each item is represented by an index Ij. The dataset is represented in Table 1.

The algorithm starts with the first scan of the database which derives the set of frequent items (1-itemsets) and their support counts (frequencies).Let the minimum support count is 2. The set of frequent items is sorted in the order of descending support count. This resulting set or list is denoted as L. Thus, we have:

$$L = \{I2: 7, I1: 6, I3: 6, I4: 2, I5: 2\}$$

An FP-tree is then constructed as follows. First, create the root of the tree, labeled with "null". Scan database D a second time. The items in each transaction are processed in L order (i.e., sorted according to descending support count) and a branch is created for each transaction.

Table 1: Transactional data for a sailing company

TID	List of items IDs
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

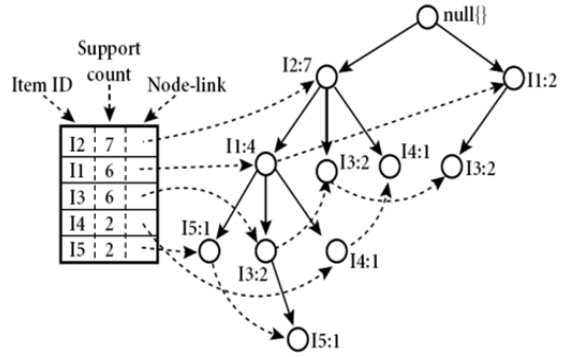

Fig. 1: An FP-tree registers compressed, frequent pattern information

For example, the scan of the first transaction, "T100: I1, I2, I5," which contains three items (I2, I1, I5 in L order), leads to the construction of the first branch of the tree with three nodes, <I2:1>, <I1:1> and <I5: 1>, where I2 is linked as a child of the root, I1 is linked to I2 and I5 is linked to I1. The second transaction, T200, contains the items I2 and I4 in L order, which would result in a branch where I2 is linked to the root and I4 is linked to I2. However, this branch would share a common prefix, I2, with the existing path for T100. Therefore, besides of incrementing the count of the I2 node by 1, a new node, <I4:1> is created which is linked as a child of <I2:2>.

In general, when considering the branch to be added for a transaction, the count of each node along a common prefix is incremented by 1 and nodes for the items following the prefix are created and linked accordingly.

To facilitate tree traversal, an item header table is built so that each item points to its occurrences in the tree via a chain of node-links. The tree obtained after scanning all of the transactions is shown in Fig. 1 with the associated node-links. In this way, the problem of mining frequent patterns in databases is transformed to that of mining the FP-tree.

Table 2: Mining the FP-tree by creating conditional (sub-) pattern bases

Item	Conditional pattern base	Conditional FP-tree	Frequent pattern
I5	{{I2, I1: 1}, {I2, I1, I3: 1}}	<I2: 2, I1: 2>	{I2, I5: 2}, {I1, I5: 2}, {I2, I1, I5: 2}
I4	{{I2, I1: 1}, {I2:1}}	<I2: 2>	{I2, I1:2}
I3	{{I2, I1: 2}, {I2: 2}, {I1: 2}}	<I2: 4, I1: 2>, <I1: 2>	{I2, I3: 4}, {I1, I3: 4}, {I2, I1, I3: 2}
I2	{{I2: 4}}	<I2: 4>	{I2, I1: 4}

The FP-tree is mined as follows: Start from each frequent length-1 pattern (as an initial suffix pattern); construct its conditional pattern base (a "subdatabase" which consists of the set of prefix paths in the FP-tree co-occurring with the suffix pattern), then construct its (conditional) FP-tree and perform mining recursively on such a tree. The pattern growth is achieved by the concatenation of the suffix pattern with the frequent patterns generated from a conditional FP-tree. Mining of the FP-tree is summarized in Table 2.

We first consider I5, which is the last item in L, rather than the first. The reason for starting at the end of the list will become apparent as we explain the FP-tree mining process. I5 occurs in two branches of the FP-tree of Fig. 1. (The occurrences of I5 can easily be found by following its chain of node-links.) The paths formed by these branches are <I2, I1, I5: 1> and <I2, I1, I3, I5: 1>. Therefore, considering I5 as a suffix, its corresponding two prefix paths are <I2, I1: 1> and <I2, I1, I3: 1>, which form its conditional pattern base. Its conditional FP-tree contains only a single path, <I2: 2, I1: 2>; I3 is not included because its support count of 1 is less than the minimum support count. The single path generates all the combinations of frequent patterns: {I2, I5: 2}, {I1, I5: 2}, {I2, I1, I5: 2}.

For I4, its two prefix paths form the conditional pattern base, {{I2 I1: 1}, {I2: 1}}, which generates a single-node conditional FP-tree, <I2: 2> and derives one frequent pattern, <I2, I1: 2>. Similar to the above analysis, I3's conditional pattern base is {{I2, I1: 2}, {I2: 2}, {I1: 2}}. Its conditional FP-tree has two branches, <I2: 4, I1: 2> and <I1: 2>, as shown in Fig. 1, which generates the set of patterns, {{I2, I3: 4}, {I1, I3: 4}, {I2, I1, I3: 2}}. Finally, I1's conditional pattern base is {{I2: 4}}, whose FP-tree contains only one node, <I2: 4>, which generates one frequent pattern, <I2, I1: 4>.

Pervious works: FP-growth (Han *et al*., 2000) is a well-known algorithm that uses the FP-tree data structure to achieve a condensed representation of the database transactions and employs a divide-and-conquer

approach to decompose the mining problem into a set of smaller problems. In essence, it mines all the frequent itemsets by recursively finding all frequent itemsets in the conditional pattern base which is efficiently constructed with the help of a node link structure. A variant of FP-growth is the H-mine algorithm (Pei *et al*., 2001). It uses array-based and trie-based data structures to deal with sparse and dense datasets, respectively. Patricia Mine (Pietracaprina and Zandolin, 2003) employs a compressed Patricia trie to store the datasets. FP-growth (Grahne and Zhu, 2003) uses an array technique to reduce the FP-tree traversal time. In FP-growth based algorithms, recursive construction of the FP-tree affects the algorithm's performance.

Eclat (Zaki *et al*., 1997) is the first algorithm to find frequent patterns by a depth-first search and it has been devised to perform well. It uses a vertical database representation and counts the itemset supports using the intersection of tids. However, because of the depth-first search, pruning used in the Apriori algorithm is not applicable during the candidate itemsets generation. The Eclat (Zaki *et al*., 1997) uses the vertical database representation. They store the difference of tids called diffset between a candidate k itemset and its prefix k-1 frequent itemsets, instead of the tids intersection set. They compute the support by subtracting the cardinality of diffset from the support of its prefix k-1 frequent itemset. This algorithm has been shown to gain significant performance improvements over Eclat (Grahne and Zhu, 2003). However, when the database is sparse, diffset will lose its advantage over tidset.

VIPER (Shenoy *et al*., 2000) and Mafia (Burdick *et al*., 2005) also use the vertical database layout and the intersection to achieve a good performance. The only difference is that they use the compressed bitmaps to represent the transaction list of each itemset. However, their compression scheme has limitations especially when tids are uniformly distributed. The search strategy of the algorithm integrates a depth-first traversal of the itemset lattice with effective pruning mechanisms that significantly improve mining performance.

The dEclat algorithm (Zaki and Gouda, 2003) makes use of the vertical database representation where each item maintains a set of transaction ids where this item is contained. They store the difference of ids, called the diffset, between the candidate itemset and its prefix frequent itemsets, instead of the ids intersection set. They compute the support by subtracting the cardinality of diffset from the support of its prefix frequent itemset.

Table 3: Variable and their definition

Symbol	List of items IDs
L	Set of all frequent itemsets with length 1.
SUP	Support count of an itemset like "T" or an item like "I".
T	A pattern or itemset like {a, b, c}.
M	Set of all possible patterns or itemsets.
FP	A frequent pattern like "T" which SUP (T) > minimum support.
Fj	Set of all frequent patterns which end with "Ij".
F	Set of all possible frequent patterns (Definition 2.5) over the set "M" (Definition 2.3).
Fi	Set of all frequent patterns which their last item is "Ij ∈ L".
Ij	An item

The proposed approach: The fundamental theorem of arithmetic says that every positive integer has a unique prime factorization. What the FP-growth does is getting a common suffix and then extracts all possible prefixes and after joining them to the suffix a frequent pattern is created. In the FP-growth algorithm it is not important that we are looking for all frequent patterns end to a particular suffix like "I5" or we want to extract all of the frequent patterns. In contrast with FP-growth our algorithm for mining of all frequent patterns end to a particular suffix like "I5", does not create entire of the tree but just focuse on prefixes related to that particular suffix.

Without generating a tree, our algorithm called Prime Factor Miner (PFM) extracts the frequent prefixes and generates the frequent itemset which end with that suffix. In Table 3 all of the used symbols and acronyms which are used are presented.

The following provides some primitive definitions which are necessary to clarify the frequent pattern mining problem.

Definition 1: "L" is defined as a set of all frequent itemsets with length 1 and is denoted as follows:

L = {I1: SUP (I1), I2: SUP (I2), …, In: SUP (In)}

Where:

Ii = A frequent itemset with length 1
"SUP(Ii)" = A support count of itemset
"Ii" = Greater than minimum support count
"L" = Sorted descending based on support count, which means SUP (Ii) > SUP (Ii+1)

For instance referring to Table 1 the L set is {I2:7, I1:6, I3:6, I4:2, I5:2}.

Definition 2: A pattern or itemset "T" with length m is represented as T = {I1, I2, …, Im} such that "Ij" represents the item in "j_{th}" position of "T". For example

if T = {a, b, c} then "I1" is the item "a". All of the patterns "Ti" is sorted in "L" order which means SUP (Ii) > SUP (I (i+1)).

Definition 3: Set "M" is defined as a set of all patterns or itemsets which is also called the transaction table and is represented as:

M = {T1, T2, …, Tn}

Where, "T" is a pattern or itemset (Definition 2.2).

Definition 4: A frequent pattern "FP" is a pattern like T = {I1, I2, …, Ik} such that the "SUP (T)" is greater than minimum support count.

Definition 5: The set "Fj" is defined as a set of all frequent patterns where their last item is "Ij" that "Ij∈ L". It means "Ij" is a suffix for all of the patterns in "Fj" set. For example if "I3" is "h" then "F3" is set of all frequent patterns like "abh" or "asdfh" where the last item is "h". Note that when "i ≠ j" then "Fj∩Fi = ∅" which means there is no frequent pattern like "T" that at the same time ends with two different items "Ii" and "Ij".

Definition 6: The set "F" is a set of all possible frequent patterns (Definition 5) over the set M (Definition 3). It is clear that we can partition all of the frequent patterns or set "F" by their last item such as Definition 5. Therefore set "F" is represented as F = {F1, F2, …, Fm} such that:

- m ≤ number of items = |L|
- Fi ∩ Fj =∅.
- Fi = {T1,T2, …, Tk} such as
  - "Fi" is a set of all frequent patterns ends with "Ii" (Definition 6)
  - "Ti" is a frequent pattern
  - "Ti" = {I1, I2, …, Ii}

Frequent pattern mining problem: The problem of mining the frequent patterns of set "M" is reduced to the problem of mining "Fj" sets. Frequent pattern mining for "Fj" is achieved by extracting all prefixes (subpattern) such that if joining the prefixes to the related suffix "Ij" the result pattern is a frequent pattern. In the following the PFM algorithm is explained. The Fig. 2 presents the first phase of the algorithm.

The first phase of PFM is similar to the FP-growth. In this phase PFM derives the set of frequent items (1-itemsets) and their support counts (frequencies) which are greater than the minimum support count. This set is called "L" and is sorted in the order of descending support count. For example by considering Table 1 the result is L = {I2: 7, I1: 6, I3: 6, I4: 2, I5: 2}.

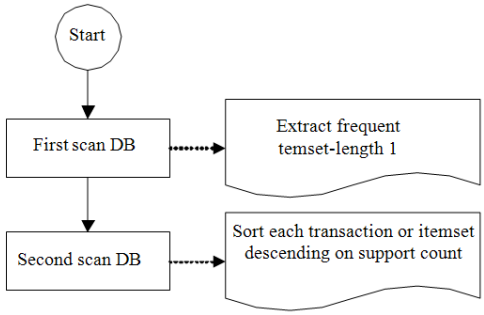

Fig. 2: The first phase of PFM (data pre-processing)

Table 4: Sorted transactional data based on "L" set order (descending on support count)

TID	List of items Ids
T100	I2, I1, I5
T200	I2, I4
T300	I2, I3
T400	I2, I1, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I2, I1, I3, I5
T900	I2, I1, I3

In addition in the scanning process, each transaction record is sorted based on the "L" set order. For example in Table 1 the transaction "T100" is "I1, I2, I5" thus according to the "L" set order it is sorted to "I2, I1, I5". The result of sorting is presented in Table 4.

Fig. 3 presents the flows for the second phase which consists of 7 main steps:

Step 1: In this step the last item or the most minimum support count in the set "L" is selected as the suffix, rather than the first. Then, when PFM finds all of the prefixes for this suffix, the next last item from the "L" is selected and the same process is repeated until there is no more unvisited item in "L".

Steps 2, 3, 4: After selecting a suffix such as "Ik" PFM scans the transaction table (DB) or set "M" (Definition 2.3). From each itemset or pattern that contains "Ik" the related prefix which is called Candidate Prefix (CP) is extracted. For example by considering the transaction "T100" in Table 4 if the "Ik" is "I5" then "I2, I1" is the candidate prefix.

Instead of using a tree for counting the pattern support, PFM uses prime numbers and prime factorization. Each item in "L" is assigned a prime

Table 5: Function H(x) structure

x	I2	I1	I3	I4	I5
H(x)	2	3	5	7	11

number in ascending order. For instance in our example after assigning the prime numbers, the L set becomes {I2 (2), I1 (3), I3 (5), I4 (7), I5 (11)}:

Step 5: When all of the candidate prefixes have been extracted then for each candidate prefix like "P_i" a unique number called "GENE" is generated as follow:

For $P_i = \{P_{i1}, P_{i2}, \ldots, P_{ik}\}$, $P_{ij} \in L$

$$\text{GENE}(P_i) = \prod_{j=1}^{k} H(P_{ij}) \quad (1)$$

The "H(x)" function is just a simple mapping that for a given item like "x" it returns the related prime number for the item. The function H(x) for the example in Table 5 is presented.

According to the fundamental theorem of arithmetic there are no two different rows with the same "GENE" number:

Step 6: The generated "GENE" numbers will be multiplied together. The result is called the "Genome" of the given suffix. The mathematical representation of "Genome" function is follows:

$$\text{Genome}(M, l_k) = \prod_{i=1}^{n} \left(\prod_{j=1}^{\text{Len}(p_i)} H(P_{ij}) \right) \quad (2)$$

Where:
"n" = The total number of patterns
"Len (P_i)" = The number of items for the pattern "P_i"

The processes of steps 2, 3, 4, 5 and 6 are repeated for all of the container rows or patterns and at the end of each cycle the value of "Genome" will be updated and multiplied with new "GENE" value.

Again consider the Table 4. We assume that the given suffix is "I5". We can see there are two container patterns (T100, T800) for "I5". The result of computing the "Genome" is presented in Table 6. For each container row the candidate pattern is marked by underline.

The "Genome" is a multiplication of these "GENE" numbers. In this example it would be (2*3)*(2*3*5) which can be simplified to 2^2*3^2*5 which is a numerical representation for all of the prefixes that by joining to the "Ik" (in this example "I5") the result is a frequent pattern.

Table 6: PFM process over Table 4

TID	Patterns	Gene
T100	I2, I1, I5	H (I2)*H (I1) = 2*3
T200	I2, I4	
T300	I2, I3	
T400	I2, I1, I4	
T500	I1, I3	
T600	I2, I3	
T700	I1, I3	
T800	I2, I1, I3, I5	H (I2) *H (I1) *H (I3) = 2 * 3 * 5
T900	I2, I1, I3	

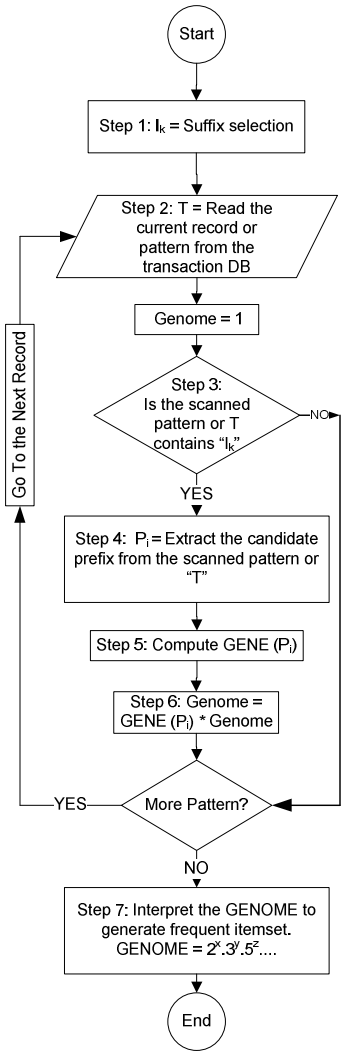

Fig. 3: The second phase of PFM (Generate frequent itemset)

The multiplicity or power of each prime factor in the Genome is the support count of the related item to that prime factor. This support count is just among the container patterns which contain "Ik". Also all of the prime factors with multiplicity lower than minimum support must be removed.

According to the computed "Genome" for the Table 6 the power of prime factor 3 which is for item "I3" is 1 where it is lower than minimum support thus the prime factor 5 must be removed. Finally the result of "Genome" for "Ik" after removing 5 is equal to 2^2*3^2. The multiplicity of prime factor 2 which is for item "I2" shows that "I2" is repeated two times as part of prefix for the patterns that have "I5" as their suffix:

Step 7: Finally PFM maps the prime factors to their related item. Thus from 2^2*3^2 we have {I2:2, I1:2} and this is known in FP-growth as Conditional FP-tree and we call it Frequent Prefix. Finally PFM generates all of the subsets for this set and add the given suffix to the end of each subset, the same as in FP-growth.

In this example the subsets are {I2}, {I1}, {I2, I1},{} and by adding "Ik" which is I5 three frequent patterns {I2, I5}, {I1, I5}, {I2, I1, I5} are generated. For the support count it is clear that for each frequent itemset like {I2, I1, I5} the support count is the minimum support count between items. For instance the pattern {I2, I1, I5} has the support equal to 2.

RESULTS

Our evaluation for PFM is done by computing the time and memory complexity. For the purpose of the evaluation, the algorithm is evaluated starting from the step where a suffix is given to the PFM algorithm. Given "Ik" all of the transaction rows or patterns must be checked to extract all of the container patterns, therefore in the worst case all of the rows must be checked. For each row or pattern which includes the "Ik" the "GENE" for that pattern must be computed. In the worst case we assume that the length of each pattern is "m" and it is the length of the longest pattern. According to equation (F2), we should change the "Len (P_i)" to "m":

$$\text{Genome}(M, l_k) = \prod_{i=1}^{n}\left(\prod_{j=1}^{m} H(P_{ij})\right)$$

Therefore the time complexity for this algorithm is O (n^2).

For memory complexity it is clear that the maximum data we should keep in the memory is just a simple integer number for the Genome. But in contrast for FP-growth because of the FP-tree, for a transaction database with n records and maximum m items for each record we need a memory from O (n^2), whereas as mentioned earlier, for PFM the memory order is O (1). It means we do not need to keep a bunch of data in a particular data structure like tree or array.

DISCUSSION

Our result confirms that, the significant objective of this study is satisfied. This objective is achieving an algorithm with low memory consumption, which can be considered as the main benefit in compare with FP-Growth.

As mentioned through introduction chapter, when the database is large, it is sometimes unrealistic to construct a main memory based FP-tree. Especially if we are interested in long frequent patterns, the memory consumption becomes a critical problem.

PFM is based on number theory which means in this study a numerical approach to present and extract frequent patterns is devised. Hence, the maximum needed memory space is equal to the memory which is needed for numerical computations.

Majority of numerical approaches, have this benefit, that they are free from high memory consumption, as well as PFM.

Moreover, must of previous studies, did not focus on a numerical approach, and this is one of the difficulties to decrease the memory consumption.

Therefore, it would be expected result to have O(1) as memory order and this result is proved in the result section.

CONCLUSION

The main aim of PFM is to reduce the memory and time complexity. Without generating any tree PFM is able to extract all of the frequent patterns. Thus for a large database no tree data structure is required in the memory. Removing the tree generation step has definitely increases the speed of the approach.

FP-growth is a noble approach that allows frequent patterns to be identified without generating candidate. But for large database and frequently changing or real time database, creating this tree can be a time consuming process.

Frequent pattern mining using prime factorization is a fast and simple approach. Also when the database is changed, only the rows that have been changed are considered. This makes PFM algorithm suitable for real time transactional frequent pattern mining where modifications and frequent pattern mining are common.

REFERENCES

Burdick, D., M. Calimlim, J. Flannick, J. Gehrke and T. Yiu, 2005. MAFIA: A maximal frequent itemset algorithm. IEEE Trans. Know. Data Engineer., 17: 1490-1504. DOI: 10.1109/TKDE.2005.183

Grahne, G. and J. Zhu, 2003. Efficiently using prefix-trees in mining frequent itemsets. Proceeding of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations, 19 December 2003, Melbourne, Florida, USA, pp: 90. DOI: 10.1.1.3.6241

Han, J. and M. Kamber, 2006. Data Mining Concept and Techniques. 2nd Edn., Morgan Kaufman, ISBN: 978-1558609013, pp: 242-245.

Han, J., J. Pei and Y. Yin, 2000. Mining frequent patterns without candidate generation. Proceeding of the ACM SIGMOD International Conference on Management of Data, ACM Press, Dallas, Texas, 2000, pp: 1-12. DOI: 10.1145/342009.335372

Patel, R., S.S. Rana and K.R. Pardasani, 2005. Model for load balancing on processors in parallel mining of frequent itemsets. Am. J. Applied Sci., 2: 926-931. ISSN: 1546-9239

Ren, J., X. Zhang and H. Peng, 2006. MFTPM: Maximum frequent traversal pattern mining with bidirectional constraints. J. Comput. Sci., 2: 704-709. DOI: 10.3844/jcssp.2006.704.709

Shenoy, P., J.R. Haritsa, S. Sudarshan, G. Bhalotia and M. Bawa *et al*., 2000. Turbo-charging vertical mining of large databases. Proceeding of the ACM SIGMOD International Conference on Management of Data, ACM Press, Dallas, Texas, 2000, pp: 22-23. DOI: 10.1145/335191.335376

Verkhovsky, B.S., 2009. Integer factorization: Solution via algorithm for constrained discrete logarithm problem. J. Comput. Sci., 5: 674-679. ISSN: 1549-3636

Zaki, M.J. and K. Gouda, 2003. Fast vertical mining using diffsets. Proceeding of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, D.C., ACM Press, New York, pp: 326-335. DOI: 10.1145/956750.956788

Zaki, M.J., S. Parthasarathy, M. Ogihara and W. Li, 1997. New algorithms for fast discovery of association-rules. Proceeding of the 3rd International Conference on Knowledge Discovery and Data Mining, AAAI Press, pp: 283-286. DOI: 10.1.1.42.3283

Zubair Rahman, A.M.J.M. and P. Balasubramanie, 2008. An efficient algorithm for mining maximal frequent item sets. J. Comput. Sci., 4: 638-645. ISSN: 1549-3636