A Frequent Pattern Mining Algorithm Based on FP-growth without Generating Tree

Hossein Tohidi[1], **Hamidah Ibrahim**[2]

Faculty of Computer Science and Information Technology
Universiti Putra Malaysia
Serdang, MALAYSIA
[1]*tohidi.h@gmail.com*, [2]*hamidah@fsktm.upm.edu.my*

ABSTRACT

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. 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. For a large database, constructing a large tree in the memory is a time consuming task and increase the time of execution. In this paper 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 Frequent Pattern-Prime Factorization (FPPF).

Keywords
Data Mining, Frequent Pattern Mining, Association Rule Mining

1.0 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.

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."

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:

i. It may need to generate a huge number of candidate sets.

ii. It may need to repeatedly scan the database and check a large set of candidates by pattern matching.

Can we design a method that mines 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.

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 *Frequent Pattern-Prime Factorization (FPPF)* is similar to FP-growth where the least frequent item is candidate as a suffix then generates all frequent patterns which end with the given suffix. The FPPF is based on the prime factorization from the number theory and does not require the creation of a tree structure.

This paper is organized as follows. Section 2 presents the related works and it also explains the FP-growth algorithm. Section 3 presents our proposed approach while section 4 presents the result. Conclusion is given in the final section.

2.0 RELATED WORK

This section consists of two parts. The first part focuses on some previous works related to this study while the second part focuses on the FP-growth algorithm and explains the algorithm through example.

2.1 Previous Works

FP-growth (Han, Pei, & Yin, 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, Han, Lu, Nishio, Tang, & Yang, 2001). It uses array-based and trie-based data structures to deal with sparse and dense datasets, respectively. Patricia Mine (Zandolin, 2009) employs a compressed Patricia trie to store the datasets. FP-growth (Grahne & 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, Parthasarathy, Ogihara, Li, & W., 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, Parthasarathy, Ogihara, Li, & W., 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 (Zaki, Parthasarathy, Ogihara, Li, & W., 1997). However, when the database is sparse, diffset will lose its advantage over tidset.

VIPER (Shenoy, Haritsa, Sudarshan, Bhalotia, Bawa, & Shah, 2000) and Mafia (Burdick, Calimlim, & Gehrke, 2001) 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 (Zaki & Gouda, Fast Vertical Mining using Diffsets, 2003).

2.2 FP-growth Algorithm

In this section 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 & 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).Le t 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}

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

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.

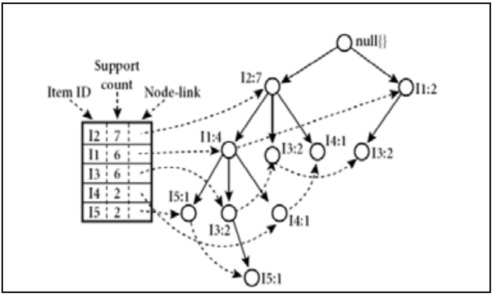

Figure 1: An FP-tree registers compressed, frequent pattern information.

The tree obtained after scanning all of the transactions is shown in Figure 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.

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. Mining of the FP-tree is summarized in Table 2.

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}

3.0 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 the FPPF for mining of all frequent patterns end to a particular suffix like "I5", does not create entire of the tree and just focuses on prefixes related to that particular suffix.

Without generating a tree, our algorithm called *Frequent Pattern-Prime Factor* (FPPF) extracts the frequent prefixes and generates the frequent itemset which ends with that suffix. In Table 3 all of the used symbols and acronyms which are used in this section are presented.

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

Definition 2.1: "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" is a frequent itemset with length 1.
- "SUP (Ii)" is a support count of itemset "Ii" which is greater than minimum support count.
- "L" is 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}.

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.

Definition 2.2: 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 2.3: 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 2.4: A *frequent pattern* "FP" is a pattern like T = {I1, I2, …, Ik} such that the "SUP(T)" is greater than minimum support count.

Definition 2.5: 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 2.6: The set "F" is a set of all possible frequent patterns (Definition 2.5) over the set *M* (Definition 2.3). It is clear that we can partition all of the frequent patterns or set "F" by their last item such as Definition 2.5. Therefore set "F" is represented as F = {F1, F2, …, Fm} such that:

- m ≤ number of items = $|L|$.
- Fi $\cap$ Fj = $\emptyset$.
- Fi = {T1,T2, …, Tk} such as:
  - "Fi" is a set of all frequent patterns ends with "Ii" (Definition 2.6).
  - "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 following the FPPF algorithm is explained. The Figure 2 presents the first phase of the algorithm.

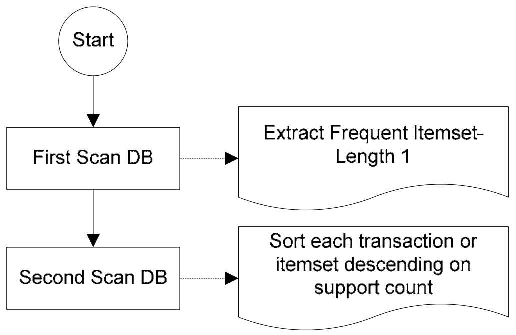
Start
First Scan DB
Extract Frequent Itemset-Length 1
Second Scan DB
Sort each transaction or itemset descending on support count

Figure 2: The First Phase of FPPF (Data Pre-processing)

The first phase of FPPF is similar to the FP-growth. In this phase FPPF 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}.

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.

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

Figure 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 FPPF 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" FPPF 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, FPPF uses prime numbers and prime factorization. Each item in "L" is assigned a *prime 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.

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

GENE (P_i) = $\prod_{j=1}^{k} H(P_{ij})$ *(F1)*

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 as follows:

Table 5: Function H(x) Structure

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

$$Genome(M, I_k) = \prod_{i=1}^{n} \left(\prod_{j=1}^{Len(P_i)} H(P_{ij}) \right) \qquad (F2)$$

where "n" is the total number of patterns and the "Len (P_i)" is 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.

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 $2^2 * 3^2 * 5$ while 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.

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 FPPF 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 FPPF 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.

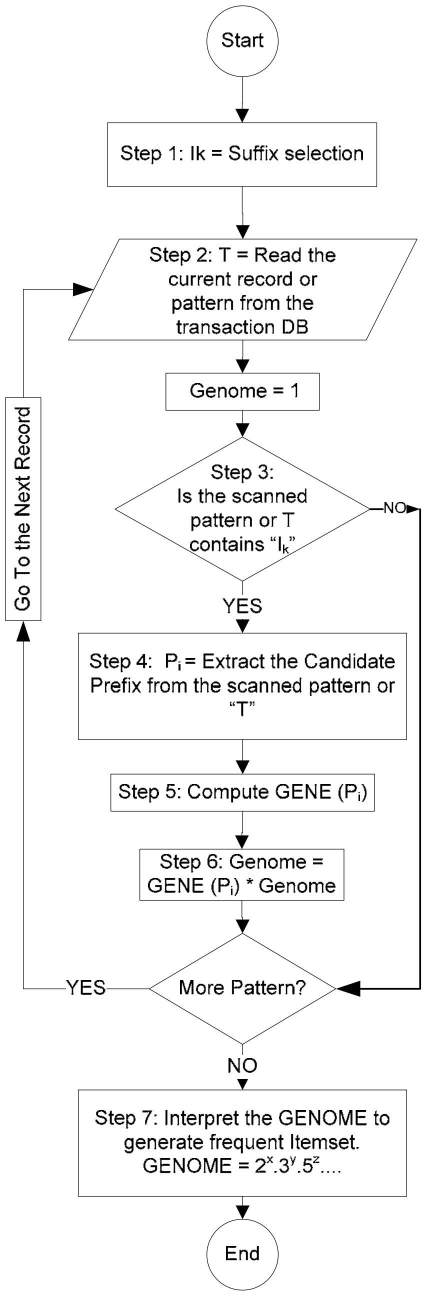
Start
Step 1: Ik = Suffix selection
Step 2: T = Read the current record or pattern from the transaction DB
Genome = 1
Step 3: Is the scanned pattern or T contains "I_k"
NO
YES
Step 4: P_i = Extract the Candidate Prefix from the scanned pattern or "T"
Step 5: Compute GENE (P_i)
Step 6: Genome = GENE (P_i) * Genome
More Pattern?
YES
Go To the Next Record
NO
Step 7: Interpret the GENOME to generate frequent Itemset. GENOME = $2^x.3^y.5^z$....
End

Figure 3: The second phase of FPPF (Generate drequent itemset)

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.

Table 6: FPPF 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~~	

4.0 RESULT AND DISCUSSION

Our evaluation for FPPF 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 FPPF 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"

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

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.

5.0 CONCLUSION

The main aim of FPPF is to reduce the memory and time complexity. Without generating any tree FPPF 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 FPPF algorithm suitable for real time transactional frequent pattern mining where modifications and frequent pattern mining are common.

REFERENCES

Burdick, Calimlim, M., & Gehrke, J. (2001). MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases. *International Conference on Data Engineering*, pp. 443-452. Heidelberg, Germany.

Grahne, G., & Zhu, J. (2003). Efficiently using Prefix-Trees in Mining Frequent Itemsets. *ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, (p. n.p.). Melbourne.

Han, J., & Kamber., M. (2006). *Data-Mining Concepts and Techniques*. Morgan Kaufmann Publishers Elsevier.

Han, J., Pei, J., & Yin, Y. (2000). Mining Frequent Patterns without Candidate Generation. *ACM SIGMOD International Conference on Management of Data*, pp. 1-12. Dallas, Texas: ACM Press.

Pei, J., Han, J., Lu, H., Nishio, S., Tang, S., & Yang, D. (2001.). Hmine: Hyper-Structure Mining of Frequent Patterns in Large Databases. *IEEE International Conference on Data Mining*, pp. 441-448. IEEE.

Shenoy, P., Haritsa, J. R., Sudarshan, S., Bhalotia, G., Bawa, M., & Shah, D. (2000). Turbo-Charging Vertical Mining of Large Databases. *ACM SIGMOD International Conference on Management of Data*, pp. 22-23. Dallas, Texas: ACM Press.

Zaki, M. J., & Gouda, K. (2003). Fast Vertical Mining using Diffsets. *The Nineth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* pp. 326-335. Washington, D.C.: ACM Press.

Zaki, M. J., Parthasarathy, S., Ogihara, M., Li, & W. (1997). New Algorithms for Fast Discovery of Association Rules. *Third International Conference on Knowledge Discovery and Data Mining*, pp. 283-286. AAAI Press.

Zandolin, P. a. (2009). Mining Frequent Itemsets using Patricia Tries. *ICDM 2003 Workshop on Frequent Itemset Mining Implementations* (p. n.p.). Melbourne: FIMI.