ISSN 2007-9737

Recent Developments in Frequent Itemset Mining: A Survey

O. Jamsheela*, Raju G

EMEA College of Arts and Science,
India

Kannur University,
India

ojamshi@gmail.com

Abstract. Frequent Pattern Mining is an essential part of Association Rule Mining process and have been used in different applications. Apriori and FP-Growth are the two major algorithms in Frequent Pattern Mining. FP-tree is an efficient tree structure introduced with FP-Growth algorithm by Agarwal et al. FP-Growth outperformed all earlier frequent itemset mining methods. This paper introduces FP-Growth, concept of FP-Tree and recent improvements on FP-tree.

Keywords. Frequent itemsets, mining, trees.

1 Introduction

Data mining is the process of discovering previously unknown and useful information from large databases. Data mining techniques are widely used by organizations to extract unknown knowledge. Association Rule Mining is one of the most widely used data mining technique. Association rule mining finds all rules in the database that satisfy some minimum support and minimum confidence constraints [3].

Frequent pattern mining is the first and most time consuming step in association rule mining and hence a potential research area in data mining. Frequent patterns are patterns that appear in a data set most frequently. Finding frequent patterns plays an essential role in data mining and knowledge discovery techniques, such as association rules, classification and clustering [25]. In 1993 Agrawal et al. [2] first proposed the concept of frequent itemsets in their association rule mining model.

1.1 Frequent Itemset Mining

A set of items that frequently appear together in a transactional data set is considered as a frequent itemset. A frequently occurring subsequence, such as the pattern that customers tend to purchase first a laptop, followed by an external hard disc, and then a mouse, is a (frequent) sequential pattern. A substructure can refer to different structural forms, such as graphs, trees, or lattices, which may be combined with itemsets or subsequences [10]. A frequent structured pattern is a substructure, which occurs frequently. Mining frequent patterns leads to the discovery of interesting associations and correlations within data [12].

Frequent itemset mining is first introduced by Agrawal et al. [2] in the context of transaction databases. The frequent itemset mining can be defined as follows.

- A transaction database is a database containing a bag of transactions and each transaction is associated with a unique transaction id.
- Let D={ t_1 , t_2, ..., $t_N\}$ be a transaction database and *I*={ i_1,i_2,....i_n } be the set of distinct items appearing in D, where t_i (i$\in$ [1,N]) is a transaction and $t_i \subseteq$ *I*.
- Each subset of *I* is called an itemset.
- An itemset with k items is called a k-itemset.
- The support of itemset X in database D is defined as the percentage of transactions in D containing X:

$$support_D(X) = |\{t|t \in D \; and \; X \subseteq t\}|/|D|, \quad (1)$$

- If $\text{support}_D(X) \geq \text{min_sup}$, where min_sup is a user-specified minimum support threshold, then X is called a frequent itemset in D [18].

Given a transaction database and a minimum support threshold, the task of frequent itemset mining is to find all frequent itemsets in the transaction database [18].

An association rule is a conditional implication among itemsets, X →Y , where X⊆I, Y⊆ I, X ≠ ∅, Y≠ ∅ and X∩Y=∅. The confidence of the association rule, given as sup(X∪Y)/sup(X), is the conditional probability among X and Y, such that the appearance of X in t_i implies the appearance of Y in t_i. The problem of association rule mining is the discovery of association rules that have support and confidence greater than the min_sup and the user-defined minimum confidence.

Let us see an example. Consider the transaction database in Table 1. The data base contains 15 transactions. The minimum support is set as 3, then the frequent pattern mining problem is to find frequent itemsets with support not less than 3. Table 2 shows all the frequent Itemsets mined from Table 1 along with their frequency.

Association Rule Mining can be divided into two parts: 'find all frequent itemsets', and 'generate reliable association rules directly from all frequent itemsets' [19]. The step of rule construction is direct and less expensive. Therefore, most researchers concentrate on the complex task of the frequent itemset mining. In addition to the problem of the large number of candidates, the frequent itemset mining algorithm demands an efficient data structure to store frequent itemsets for further processing.

Table 1. Transaction database

TID	Transactions
1	B,D,F
2	B,C,J P,Q
3	A,B,G,H
4	B,C,D,E,G
5	C,D,E,F,J
6	B,C,F,J, R, T
7	A ,D, E ,S
8	C,F,L ,U
9	D,F,H,I ,S
10	C,F, R
11	K,L,M
12	A,K,M, Q
13	E,K,T
14	H,L,U
15	M,N,O

Table 2. The Frequent Itemsets extracted from Table 1

ID	Frequent Itemsets	Frequency
1	C	6
2	F	6
3	B	5
4	D	5
5	E	4
6	A	3
7	H	3
8	J	3
9	K	3
10	L	3
11	M	3
12	F C	4
13	B C	3
14	D F	3
15	E D	3
16	J C	3

2 Various Approaches to Frequent Itemsets Mining

Frequent pattern mining extracts specific patterns with supports higher than or equal to a minimum support threshold. Apriori [2] and FP-growth [13] methods are two basic approaches to find frequent itemsets. Many new algorithms and improvements on the basic algorithms have been suggested to improve the efficiency of frequent itemset mining, but Apriori [2] and FP-growth [13] are still regarded as the baseline algorithms.

2.1 Apriori

Agrawal and Srikant [2] introduced the Apriori algorithm for mining frequent itemsets. Apriori

is the oldest algorithm and it is the candidate generation and test approach such that the frequent k-itemsets in one iteration can be used to construct candidate (k + 1)-itemsets for the next iteration. This is the first technique to find frequent patterns based on the Apriori principle and the anti-monotone property, i.e. if a pattern is found to be frequent then all of its non-empty subsets will be frequent. In other words, if a pattern is not frequent, then none of its super sets can be frequent. The Apriori technique calculates the frequent patterns of length k from the set of already generated candidate patterns of length k-1. Apriori terminates its process when no new candidate itemsets can be generated. Therefore, this algorithm demands multiple database scans. The number of scans are equal to the size of the maximum length of frequent patterns in the worst case. It requires a large amount of memory to handle the candidate patterns when the number of potential frequent patterns is reasonably large. Various methods have been proposed by researchers as an improvement of Apriori.It is still used in recent research to enhance recommender systems by addressing the limitations of collaborative filtering in sparse data. It helps build user profiles based on item ratings and categorical attributes. Studies on the MovieLens dataset show that this approach improves prediction accuracy compared to traditional methods [24]. However, those methods could not avoid scanning the database several times to find frequent itemsets.

2.2 FP-growth

Han et al. [13] introduced the FP-growth algorithm to mine frequent itemsets without generating candidate itemsets. Apriori-based algorithms consume more time and memory due to the generation of candidate patterns. FP-growth-based algorithms generally have better performance than that of Apriori-based algorithms [19]. FP-growth uses a Frequent Pattern tree (FP-tree) for efficiently mining association rules. In this method the FP-tree captures the content of the transaction database and stores all the transactions (only frequent items) in a compressed form. It scans the database only twice. In addition to FP-tree, another data structure, a header table, is used to traverse the tree and find frequent itemsets quickly. The header table stores frequent 1-itemsets in decreasing order of their frequencies. Each item in the header table points to the first occurrence of the corresponding node in the FP-tree. All nodes with similar items in the FP-tree are connected by using a link. After completing the FP-tree construction, the next step of the FP-growth algorithm is to mine frequent patterns from the FP-tree. It starts the mining from the least frequent item to the most frequent item. It traverses from the leaf nodes of the FP-tree to the root. Paths with the same prefix item in the FP-tree are used to construct conditional FP-tree. Using the conditional FP-tree, the algorithm can generate frequent itemsets with the same prefix. It is proved theoretically and empirically that FP-growth is faster than Apriori algorithm [15, 6, 4].

2.2.1 Example

Consider the transaction database in Table 1. The database contains 15 transactions with 21 items. The minimum support value is set as 3. The procedure of FP-growth algorithm begins by scanning the database to find frequent 1-itemsets. The result of the first scan is displayed in table 3. The frequent items sorted by the order of the frequency are shown in Table 4. Only 11 items are frequent. Scan the database again for constructing the FP-tree. During the second scan consider each transaction. After removing the infrequent items from the transaction, sort the remaining frequent items in frequency descending order and insert the transaction in to the FP-tree.

In the above example ‘B D F’ is the first transaction. The items (B,D,F) in the transaction are frequent items and hence sort the items according to the frequency count. The sorted transactions are displayed in the third column of Table 5. The sorted transaction is used to construct the FP-tree.

Before inserting the first transaction, a root should be created with the label as null. Insert the first transaction ‘F B D’ to the tree. Set the node with label F as the first child of the root with support 1. Node ‘B’ will be the child of ‘F’ and ‘D’ will be

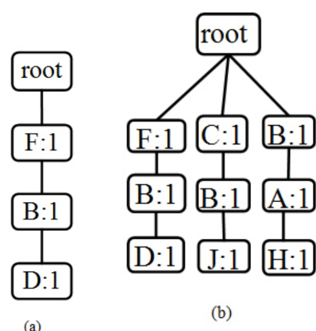

Fig. 1. (a) FP-tree after inserting transaction-1, (b) FP-tree after inserting transaction-3

the child of ‘B’ and ‘F B D’ is formed as a branch from root.

The support of all three nodes is 1. The tree after inserting the first transaction is displayed in Figure 1(a).

Insert the second transaction ‘C B J’. Search node C among the children of root. After inserting the first transaction the root of the FP-tree has only one child which is node F, so create a new node with label C and set the support as 1. Insert the new node as the child of root. Form ‘B’ and ‘J’ as branch from C with support 1. Insert the third transaction as same as transaction 2. Now root has three children. The FP-tree after inserting third transaction is shown in figure 1(b) .

During the insertion of fourth transaction ‘C B D E’ search among the children of root for C. C is a child of root. Increase the support count of C by adding 1. Now the support of C is 2. B is the second item of the current transaction. Search B among the children of current node C. B is the only child of C, increase the support of B by 1. Next item to be inserted is D. Search D among the children of current item B. B has only one child J. Create a branch from B with the remaining nodes D and E with support 1.

Figure 2 is the FP-tree with 4 transactions. The complete FP-tree after inserting all the transactions in table 1 is displayed in Figure 3.

Table 3. List of items with frequency

item	frequency
A	3
B	5
C	6
D	5
E	4
F	6
G	2
H	3
I	1
J	3
K	3
L	3
M	3
N	1
O	1
P	1
Q	2
R	2
S	2
T	2
U	2

Table 4. List of frequent items

	item	frequency
1	C	6
2	F	6
3	B	5
4	D	5
5	E	4
6	A	3
7	H	3
8	J	3
9	K	3
10	L	3
11	M	3

3 A Detailed Review of Selected FP-tree Based Frequent Itemset Mining Algorithms

Apriori [2] and FP-growth [13] are two basic frequent itemset mining approaches. FP-Tree

ISSN 2007-9737

Table 5. Original transactions and sorted transactions after removing infrequent items

TID	Transactions	Sorted transactions after removing infrequent items
1	B,D,F	F,B,D
2	B,C,J, P, Q	C, B, J
3	A,B,G,H	B, A, H
4	B,C,D,E,G	C,B,D ,E
5	C,D,E,F,J	C,F, D, E, J
6	B,C,F,J, R, T	C,F, B ,J
7	A ,D, E ,S	D, E, A
8	C,F,L ,U	C,F,L
9	D,F,H,I ,S	F,D, H
10	C,F, R	C,F
11	K, L,M	K, L, M
12	A,K,M, Q	A,K,M
13	E,K, T	E, K
14	H,L,U	H, L
15	M,N, O	M

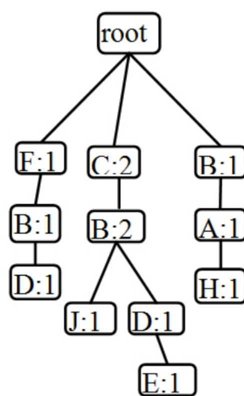

Fig. 2. FP-tree after inserting transaction-4

algorithm is better than the Apriori for large data sets for finding associations [5]

There are hundreds of modifications suggested as the improvement of FP-growth which constitutes a large family of FP-tree based algorithms.

Incorporating an elaborate discussion about all the FP-tree based algorithms is out of scope but to get a clear picture about the problems associated with FP-tree based algorithms seven prominent recent algorithms which covers basic modifications of FP-tree, have been considered.

3.1 LP-tree

[19] recommended a new method to construct the frequent pattern tree by using arrays. The tree structure is same as FP-tree and called **Linear Prefix Tree (LP-Tree)**. LP-Tree is composed of array forms to minimize pointers between nodes. LP-tree is highly effective when compared to the conventional methods because of pointers are avoided. In the FP tree structure introduced by Jai Wai Han et al.,[13] separate nodes are used to store each items in a single transaction. But in LP-tree separate arrays are used to store each transaction. They have explored the simplicity of array while inserting transactions and traversing through LP- tree. The authors created the exact FP-tree structure in LP-tree by using arrays and few pointers.

Multiple arrays are used to create the tree since single array cannot express transactions as a tree structure with many branches. Another extension named BNL(branched node list) have been used to keep track of the branched nodes in LP-tree method. BNL contains the branched node table and child node list. The branched node table stores pointers of all branched nodes. Each node stored in it has one child node list.

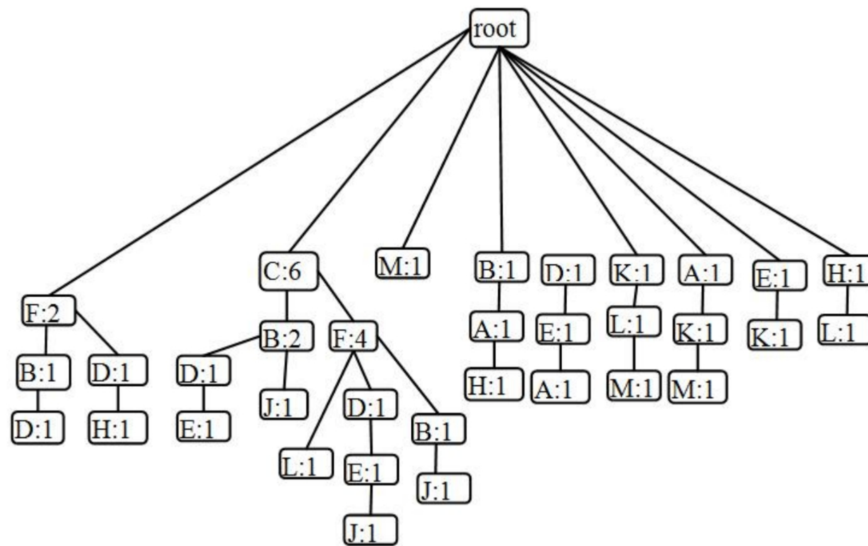

Fig. 3. FP-tree after inserting all transactions

The child node list has child node pointers of a corresponding branched node. When a new branch other than root is created in LP-tree, a new branch node is inserted to the BNL with its child list. The structure of LP-tree is as follows: LP-tree=$\{$Headerlist,BNL,LPN$_1$,LPN$_2$,...,LPN$_n\}$.

The header list is same as the header table of FP-tree, it contains the sorted frequent items, frequency count of each item and the pointer to the first node of the specific item in the LP-tree. Here the pointer is pointed to the array location in which the specific item is stored.

The process starts by scanning the database to find the frequency of each item, then create the header list and sort it in descending order of the frequency.

In second scan each transaction is being considered, prune it by removing infrequent items and sort in descending order of the frequency. The processed transaction will be stored in a newly created LPN which is a combination of arrays . The length of the array is equal to the number of items in the processed transaction. The structure of the LPN with n items is as follows:

$$LPN = \{(Parent_Link), (i_1, S, L, b), (i_2, S, L, b), ..., (i_n, S, L, b)\}. \quad (2)$$

Parent_Link is the header of the LPN to connect to its parent or to root and which is the first node of the LPN. i$_n$ represents the nth item, ‘S’ denotes the support/frequency of the item, ‘L’ is the link to the next node of the same item and ‘b’ is the flag to represent whether the node is a branch or not. If the first item of the transaction is not found on the root, then the Parent_Link of a newly created LPN is connected to the root.

Since array can hold only one child at a time(next of the current location), nodes with more than one child (branch nodes) are represented by BNL. A node becomes a branch node if it has more than one child. BNL is a linked list or any other pointer linear data structure and each node represents one branch node. The children of each branch node are represented by using another child node list and connected to its parent branch node. BNL is used to keep track of the children of each branch node. The total number of branch nodes and number of children of each branch cannot be predicted in advance, hence BNL should be a dynamic linear data structure.

3.1.1 LP-tree Insertion

During second scan each processed transaction has to be inserted into the tree. Search the

ISSN 2007-9737

children of the root to find the first item of the new transaction, if a match is found, increase the support of the item, otherwise create a new LPN and store the items of the transaction in it and header of the new LPN is pointed to the root. Root is entered in to BNL as a branched node. Add the first node of the newly created LPN into the child node list of the root in BNL. If all the items of the current transaction are same as an already inserted transaction, the support of all the items in the existing LPN is increased by one. If one or more items are same with an already existing LPN but remaining items are different, a new branch is created (new LPN) with the remaining items and its header will be pointed to the existing LPN as the parent. In this situation, if the new item (new LPN) is the second child of the branch node, a new node is added to the BNL and its child list is initialized with two children. If the branch node is already in the BNL, the new LPN header is added into the child list of the corresponding branch node.

The authors have proved that the exact FP-tree structure can be created by using only arrays but they have used some pointers during the construction of LP-tree as BNL. Each node of a basic FP-tree contains item name, support count, parent pointer, child pointer and node link. To represent one transaction FP-tree creates one or more branches with nodes and each node represents each item in the transaction. But in LP-tree one or more LPN represents one transaction and multiple arrays are used to represent the transactions in the database. An array contains a group of items in a transaction.

The mining process is same as FP-tree. Mining starts from the last item of the header list and mine the frequent items by traversing from bottom to top (root) of the LP-tree. During mining process (LP-growth) only one traversal is carried on, bottom to top, the header pointers are used to get the parent of current LPNs. Here the authors specifically mentioned that the BNL can be deleted after completing the creation of LP-tree because to traverse from bottom to top only the header pointer of each LPN is enough.

Even though the LP-tree removes the complexity of pointers, during the insertion process the LP-tree have some disadvantages. Let LPN_1 contains only 5 items. Suppose, a transaction with 10 items have to be inserted and the first 5 items are already there in LPN_1. To insert the transaction the support count of the 5 items in the LPN_1 is increased by one. A new LPN have to be created to represent the remaining 5 items and the header of the newly created LPN should be pointed to the LPN_1. In such situations a transaction may be stored in multiple LPNs even though it can be stored in only one LPN sufficiently. Further, BNL have to be used to track the diversions on the paths. To simplify these difficulties, the authors introduced an integrating method to store a newly added transaction with an existing one and also recommended to delete BNL to free up the extra space.

Six existing algorithms have been compared with LP-growth with six different datasets. A detailed experimental analysis have been conducted to prove the efficiency of LP-growth. In the experimental results LP-growth showed better performance than other algorithms with less memory requirement [19].

3.2 Frequent Items Ultrametric Trees (FIUT)

Another tree structure called the **Frequent Items Ultrametric Trees(FIUT)** is suggested by Yuh-Jiuan Tsay et al. [26] for mining frequent itemset. The FIU-tree structure is used to mine the frequent itemsets directly from the FIU-tree without creating the conditional trees. It consists of two main phases. Two database scans are conducted in first phase. First scan is used to find the frequency of all 1-itemsets. During the second scan the transactions are pruned (deleting the infrequent items), count the number of remaining items in each transaction and group the transactions in to separate clusters based on the number of items remaining in the transaction after pruning. The transactions with k frequent items are grouped into k-cluster. All the transactions in one group should have the same number of frequent items in it.

In the second phase the transactions are taken from the clusters, not from the the original database. The repetitive construction of FIU-tree and mining process are conducted in the second

phase. The next step is the construction of k-FIU tree where k is from M down to 2 and M denotes the maximal value of k among the transactions. The group of transactions with highest number of items (Transactions with M items) are considered first to create the first k-FIU tree (ie M-FIU tree). Mine the frequent M-itemsets form the constructed M-FIU tree.

The procedure to construct the k-FIU tree is as follows. At the beginning the value of k=M. The tree construction is started by creating a root labeled with null. Then take the first transaction from k-group(k-cluster), the root will be the parent of the first item of the transaction. Then repeatedly add all the remaining items as a branch from the last added item. Then consider next transaction from the same cluster. Compare the first item of the current transaction among the children of root. If exist, it will be the parent of next item. Then search second item among the children of current node. If found, it will be the parent of next item. If found a mismatch create a new node as the child of the current parent and add all remaining items as a new branch. The frequency of the transaction appeared in the cluster should be added as count in the leaf node. After finishing the construction of k-FIU-tree with all the transaction in the *k*-cluster, mining of frequent *k*-itemsets is started. At any given time only one *k*-FIU tree is present in the main memory. The FIU tree is a balanced tree ie. all leaves are at same level *k*. 'Count value' of leaf nodes in *k*-FIU tree is used for mining. If count$\geq$ min_support then the items from leaf to root (items in the path) is considered as frequent *k*-itemset. After mining all *k*-frequent itemsets by checking the leaf nodes next step is the construction of (k-1)-FIU tree. The *k*-FIU tree is decomposed to (k-1)-FIU tree and add all original (k-1)-itemsets from (k-1)cluster. Repeat the above process till all the groups(clusters) become empty.

In this approach an alternative mining method, called FIU-tree mining is introduced. The FIU-tree mining method is suggested as an efficient method for reducing the search space by considering only one k-itemset at a time. The method suggests the tree creation and mining process together. After creating the k-FIU-tree the mining of frequent k-itemsets is started. The conditional databases and conditional FP-trees need not be created here. In FP-growth method the complete FP-tree have to be loaded to the memory but in the FIUT method at a time only one k-FUI-tree is maintained in memory. Running time is also reduced here because the items in the transactions are kept in its lexicographical order and the transactions are not sorted before inserting in to the tree.

In order to evaluate the efficiency of this method the authors used two transactional databases. Several experiments were conducted to compare the new method with existing FP-growth [13] method. While processing 20000 transactions with different support threshold, a slight increase in runtime in the case of FUI-tree is visible but in FP-tree when support decreased from 0.1 to 0.08 runtime is increased by more than 5 seconds.

3.3 Improved Frequent Pattern (IFP) growth

Ke-Chung Lin et al. [17] suggested an **Improved Frequent Pattern (IFP) growth** method for mining association rules. Authors of the IFP-growth method pointed out that the algorithm requires less memory and shows better performance in comparison with FP-tree based algorithms. To lower the complexity of searching in each node the IFP-growth employs an address-table with each node of the tree. It also uses a hybrid mining method to reduce the need for rebuilding conditional FP-trees.

The authors pointed out that the task of constructing the FP-tree and the conditional FP-trees affects the mining performance of FP-growth. To add a new transaction with n items, the FP-tree have to be searched n times and to insert each item the search should be conducted 'm-i' times(in worst case), m denotes the total number of items in the transaction database, i is the position of the current item in the transaction. To insert first item, we have to search m-1 times, to insert second item m-2 times and so on. Therefore, in the worst-case scenario, the complexity of constructing a new path is calculated as (m + (m - 1) +.... + (m - (n - 1))). So to reduce the complexity of tree construction an effective data structure called address-table is introduced. Each address-table contains a set of items and pointers.

The pointer of an item points to the corresponding node of that item at the next level of IFP-tree. The address table makes the searching easy. Each node checks its address-table to confirm whether a specific child exists or not.

In IFP-growth method, the authors included one address table with each node except the nodes of last item in FP-tree. The address table of the root node includes all the frequent items and pointers to the actual nodes in first level. Each nodes in the first level is attached with a separate address table and the address table contains those items which are stored after the current item in the header table. The items are arranged in header table in the order of the frequency. Suppose the position of item B is 5 in header table and total number of frequent items are 10. The address table of item B contains the items in the header table from 6 to 10.

In mining phase the authors have introduced FP-tree^{+} method to avoid the conditional tree creation. The FP-tree^{+} mining procedure is same as the conditional FP-trees, but the direction of mining is opposite to that of mining a conditional FP-tree. FP-tree^{+} traverses the items from the top to the bottom of a header table. FP-tree^{+} can be built on the original FP-tree. FP-tree^{+} contain infrequent items because during mining the infrequent itemsets are not eliminated. It may cause potential performance problems in traversing FP-tree^{+}. To avoid this issue the authors suggested a hybrid mining method by combining the FP-tree^{+} technique and the conditional FP-tree technique.

The items in the header table is divided in to two parts by using a tree level value. An FP-tree^{+} is constructed if the item belongs to the top half to discover its original conditional base. Otherwise, a conditional FP-tree is used.

The IFP-growth is compared with two existing frequent itemset mining methods to evaluate the efficiency. The compared algorithms are FP-growth [13] and nonordfp [20]. The experimental results shows that the runtime of FP-growth is more than the other two methods but in memory consumption, nonordfp require more memory than the other two. The authors states that IFP-growth has a better performance than FP-growth and nonordfp even if the dataset is dense.

3.4 Adaptive Mechanism

For an efficient frequent itemset mining Fan-Chen Tseng [27] presents an **Adaptive mechanism** to select a suitable data structure among two pattern list structures. The Frequent Pattern List (FPL) [28] and the Transaction Pattern List (TPL) [29] are the two suggested pattern lists. The database density will change during the mining process. Therefore, the FPL data structure is suggested for sparse databases and the TPL structure is suggested for dense databases. The authors introduced a method to calculate the database density.

Numerous frequent itemset mining algorithms are introduced and each algorithm is suitable for a specific kind of database. It is hard to find an algorithm which is optimum for all kind of datasets. The authors of the paper **Adaptive mechanism** state that the features of real databases are diversified, no single data structure can be claimed to be the optimal solution, and the features of local databases in the mining process will also change during the mining process [27]. The authors suggested an adaptive mechanism to select a suitable data structure for different types of databases. In the beginning stage the database will be sparse and the FPL is used. As the mining process goes on, the conditional (local) databases become dense, in such a situation the Transaction Pattern List is used [27]. The authors [27] explained the two methods FPL and TPL by using an example.

FPL is a linear list for representing transaction databases [30]. The frequent 1-itemsets are calculated in the first scan. The frequent 1-itemsets are sorted in the descending order of the frequency and the global FPL is created with the sorted items. FPL is a data structure of item nodes to store the frequent items with support counts. FPL have one row with frequent item names and another row with the same length contains the frequency count of each item. During the second scan each transaction is processed by removing infrequent items from the transaction followed by sorting the remaining items in descending order of the frequency. The processed transaction is converted to a bit string format called transaction

string. The length of the string and the length of the total frequent item in the FPL should be the same.

For 10 frequent items, the length of the bit string should be 10. Each bit string represents one transaction. The bit 1 represents the presence of the item in the transaction. If a transaction contains 3 items, the corresponding bit string contains three 1s and 7 zeros. The bits from MSB to the last 1 bit is kept and the trailing 0 bits are trimmed. The transaction bit string is then stored in the item node of the FPL that corresponds to the last (rightmost) 1 bit of the bit string.

Mining starts from the last node of the FPL. Three basic operations are performed during the mining process. The bit counting operation, which is the first operation, is used to count the number of 1 bits in each bit position in the transaction signatures to find the frequent items.

Put the frequent items in separate groups based on the bit count. Put all the items with highest count into group one. Combine all the items in the first group with the current item to get the frequent itemsets. Put all the items with second highest count into group two which should satisfy minimum support. Construct the conditional database by using the items in group two and create the conditional FPL. Repeat the procedures (count the bits, put the items in separate groups and combine the current item with items in first group) with the conditional FPL until there are no more items.

Mining of current item, which is the last item in the global FPL, is finished. Remove the least significant 1 bit and all the trailing 0 bits of the transaction signatures from the current node of FPL. This is the second operation called the ‘signature trimming’. Next find the bit position of the least significant bit of the trimmed signatures and put the signature to the corresponding node of the FPL. This is the third process called ‘signature migration’. Now start the mining procedure of the second last item of the FPL by counting, grouping and mining etc. To calculate the database density, the following formula is suggested:

$$density_D(db) = sum/Count, \quad (3)$$

$$sum = freq_f(fp_1) + freq_f(fp_2) + ... + freq_f(fp_n), \quad (4)$$

$$Count = n \times nf, \quad (5)$$

where

$\{fp_1, fp_2, ...fp_n\}$ is the set of frequent items in the transaction DB.

n is the number of transactions in DB,

nf is the number of frequent items in DB.

A density of 78.7% means in the bit string representation of DB, 78.7% of the entries are 1s and the remaining are zeros.

While the conditional database becomes smaller the database becomes denser. Conditional databases contain more similar transactions. The authors suggested another data structure called TPL (Transaction Pattern List) if the database is dense which is better than FPL. Only one transaction signature is used in TPL to represent the identical transactions, with an additional field to indicate the number of identical transactions. If the transaction is sparse the FPL is better than TPL because the construction of TPL takes time to search for identical transactions. So the authors suggested an adaptive method to select the suitable data structure based on the density of the database among the two data structures. Two parameters are used as the criteria for switching from FPL to TPL during the mining process: the number of frequent items in group two after bit counting(*n*), and the density of the conditional database(*d*) [27]. The searching for identical patterns are time consuming if *n* is too large. So the authors set the upper bound of *n* to 60 and the lower bound of *d* to 20. Therefore during mining process, after each bit counting operation, the density is calculated. If *d* ¿= 20 and *n* is between 0 and 60 the process is switched to TPL otherwise FPL is continued.

3.5 PrePost

Deng et al. [7] introduced a new method called **PrePost** in 2012 for mining frequent itemsets from the prefix tree structure. They used the FP-tree structure to construct a 'Pre-Post Code' (PPC) tree. In PrePost algorithm, the initial steps and the tree construction are same as FP-tree. But the node structure is different. The authors have used two additional values in each node to store the pre-order and post-order rank of each node. After finding the frequent 1-itemsets, sort the items in descending order of the frequency. Next step is the construction of PPC-tree. Scan the database again to construct the PPC-tree. In each transaction, the infrequent items are removed and remaining items are sorted in descending order of frequency. This is followed by insertion of the processed transaction in to the PPC-tree.

In FP-tree, a node link is used to connect similar items together. But in PPC-tree the nodes do not have a node link field. The next step is the post-order and pre-order traversal through the entire tree to get the pre-order and post-order ranks of each node. A node of PPC-tree contains pre-order and post-order rank of the node in addition to the name of the item, support count and child links. Each node in the PPC-tree is represented by using the 'Pre order-Post order' (PP) code, which contains the node's pre-order and post-order rank and the support count. After these initial steps the tree is traversed again to create N-lists of each frequent 1-itemset. Node list (N-list) is the collection of the PP-codes of a specific item. The PP-codes are unique and each PP-code have a one to one relationship with each node in the PPC-tree. If 10 nodes of a PPC-tree contains item M, then the N-list of item M contains 10 PP-codes representing each of the nodes with item M. N-lists are used to mine the frequent itemsets without recursive construction of conditional FP-trees.

PrePost follows the Apriori-like approach for mining frequent (¿2)-itemsets with some efficient candidate generation. The bottleneck of Apriori is its huge generation of candidate itemsets. To mine frequent 2-itemsets a pre-post traversal is conducted in PPC-tree. While visiting each node during the traversal, it finds the ancestors of the node and create a 2-itemset by combining the current node with each of its ancestors. If the pair already exists in the list of 2-itemsets, increase the support count, otherwise add the pair in to the list as a new itemset. After completing the traversal compare the support count with minimum support count and remove all infrequent itemsets.

PrePost finds frequent (k+1)-itemsets (k$\geq$2) by intersecting the N-lists of frequent k-itemsets. Intersection here means verifying the ancestor-descendant relationship between frequent itemsets. By using the N-lists of each itemsets, the ancestor-descendant relationships can be found efficiently without traversing PPC-tree. To verify whether a node A is an ancestor of another node B, compare the PP-codes of A and B. If the pre-order rank of A is less than the pre-order rank of B and the post-order rank of A is greater than the post order rank of B, then A is the ancestor of B. To optimize the intersection some properties and lemmas are explained.

To check whether one itemset is the ancestor of another itemset, we need not intersect all the PP-codes in the N-lists each other because of some properties of the N-lists. Let A and B are two frequent itemset, in order to find whether AB is frequent, N-list of A with the N-list of B have to be intersected.

If it is found that by comparing the first PP-codes in each N-list, A is ancestor of B, then skip the intersection of all other PP-codes in N-list of A with the first PP-code of N-list of B, because all the PP-codes in one N-list have the same item name and the same item will not repeat in the same branch.

The frequent itemsets are mined without generating candidates by using the single path property of N-list. The single path property can be explained in a simple manner. Let S_1, S_2, ...S_n are frequent items and S_n has only one PP-code in its N-list. If the result of intersection of each S_i with $S_n(S_i \cap S_n)$ is not null, then the N-list have only one PP-code with the support of the PP-code of S_n. By using this property, the execution time is reduced.

In PrePost the PPC-tree is created to compress the database and to calculate the N-lists. The mining is conducted by using the N-lists except frequent 2-itemsets. The N-list is introduced to

speed up the mining process by avoiding the recursive construction of conditional FP-trees. But after the tree construction, the entire tree have to be traversed many times: to store pre-order rank, post-order rank of each node, to collect N-list and to find frequent 2-itemsets. The authors mentioned that they have combined the last two traversals together (the N-lists creation and mining frequent 2-itemsets).

PrePost uses a new method to mine frequent 2-itemsets without using its new data structure N-list. The N-lists with Apriori-method are used to mine frequent k($\geq$3)-itemsets. The limitation of Apriori is the generation of huge amount of candidates. It is controlled in PrePost by using the N-lists with single path property.

PrePost requires more memory than the basic FP-growth because two extra values are stored in each node. The PrePost method is compared with other algorithms to prove its efficiency. From the outcome of experiments carried out, the authors established that PrePost takes less running time than the other algorithms chosen for comparison.

The authors explained the creation and the structure of the PPC tree by using a simple example. The first algorithm is used to construct the PPC-tree. The PPC-tree is created only to generate PP-codes of the nodes. The second line of the algorithm is a database scan to find the frequent 1-itemsets. The authors did not mention any further details about the database scan.

The PPC-tree creation algorithm is same as the FP-tree construction algorithm.

3.6 FIN

Deng at al. [8] introduced another algorithm called **FIN** in 2013 to mine frequent itemsets. The authors proposed a new and more efficient data structure called Nodesets, instead of N-lists. Nodesets require only the pre-order (or post-order) of each node. The prefix-tree created in FIN is called ‘Pre order code’ -tree (POC-tree). The POC-tree creation procedure is same as PPC-tree but the nodes in POC-tree includes only pre-order (or post-order) rank of the node with all other details. The N-info of each node contains the pre-order rank and the support count of the node. The N-info is unique and has a one to one relationship with the nodes in the POC tree. In PrePost the N-lists are used for mining process but in FIN Nodesets are used. The Nodeset of an item A is the collection of N-infos of nodes with item A in the POC-tree.

After completing the pre-order traversal, scan the POC-tree again to find frequent 2-itemsets and their Nodesets. In PrePost the N-lists are prepared directly from the PPC-tree only for frequent 1-itemsets. The N-lists for frequent k(¿1)-itemsets are prepared by intersecting the N-lists of frequent 1-itemsets. The PP-codes contain both pre-order and post order rank. So it is easy to find the N-list by comparing two PP-codes. But N-info of FIN contains only the pre-order(or post-order rank) of the node. So the Nodesets are collected from the POC-tree during the mining of frequent 2-itemsets. In PrePost the PP-codes of the ancestor node is collected with the count of descendant node to form the N-list. But in FIN the N-info of the descendant node is used to form the Nodeset of a frequent item. Here the intersection is simple and direct. Let A and B are two frequent itemsets. Intersect the Nodeset of A and B to find the Nodeset of itemset AB. If a N-info appears in both of the Nodesets, then that N-info will be added into the Nodeset of itemset AB. FIN used a set-enumeration tree proposed by Rymon [22] in 1992, to represent the search space for mining process.

To reduce the search space a superset equivalence property is used. If a new item A is included into a frequent itemset B and the support of B remains same as before, then the support of $A \cup C$(C is another frequent itemset) is equal to the support of $A \cup B \cup C$. The itemset of A will be put in to the set of promoted items. The set of promoted items of the current itemset need not be included to build the child candidates because the current itemset is enough to represent all the promoted items. By applying this superset equivalence property the search space is pruned and candidates generation is drastically reduced. The Frequent k(¿2)-itemsets are prepared by generating candidates of each frequent 2-itemsets. In each candidate generation step the superset equivalence property is applied to reduce the number of candidates.

ISSN 2007-9737

Three datasets are used in the experimental evaluation. The algorithm FIN is compared with two efficient algorithms, PrePost and FP-growth*. The running time of FIN and PrePost are same with highest minimum support on mushroom dataset. FIN outperforms only when the minimum support becomes small. With connect dataset FIN outperforms the other two algorithms but the growth trend of FIN and PrePost are same. The experimental result with dataset T25I10D100K shows that PrePost is faster than the other two algorithm. The authors claimed that in the case of sparse data sets the promotion strategy was not efficient. They observed that the single path property and the superset equivalence property are disabled here and the time taken to create N-list is less than the time taken to create Nodesets. Because of these reasons PrePost outperforms FIN with the sparse dataset T25110D100K. The memory consumption of FIN is compared only with PrePost. FIN consumes less memory than PrePost.

3.7 PrePost$^+$

Deng et al., proposed another algorithm '**PrePost⁺**' [9] for frequent itemsets mining. PrePost$^+$ is a modification of PrePost. In PrePost$^+$ the N-list data structure and N-list intersection of PrePost method to find frequent k(¿2)itemsets have been applied. To speed up the mining process, the set enumeration tree [21] is used in PrePost$^+$. The authors tried to bring together the advantages of PrePost and the advantages of FIN in PrePost$^+$ to introduce a more efficient algorithm to mine frequent itemsets. The pruning strategy 'Children-Parent Equivalence' is almost same as the pruning strategy 'Superset Equivalence' which is applied in FIN algorithm. If the support of the itemset represented by the child node is equal to the support of the itemset represented by the current node, then the child node will not be generated while creating the compressed frequent itemset tree. In PrePost$^+$ the search space is reduced by applying the Children-Parent Equivalence property.

To evaluate the performance of PrePost$^+$ six real datasets have been used and compared the performance with other three algorithms, i.e., PrePost [7], FIN [8] and FP-growth* [11]. In the experimental analysis the authors have mentioned that the pruning strategy used in PrePost$^+$ is more efficient than the pruning strategies of other methods by comparing the number of visited nodes during the mining process. In performance evaluation section, with the experimental results, the authors proved that PrePost$^+$ out performs the other algorithms. The experimental results of memory usage show that FP-growth uses the lowest amount of memory than PrePost$^+$. PrePost$^+$ consumes less memory than FIN and PrePost.

The authors of FIN, PrePost and PrePost$^+$ introduced a new method to uniquely identify each node in a frequent pattern tree by using the PP-code and N-info. This information can be used to mine frequent k(¿2)itemsets without traversing the tree. But with huge datasets this new method consumes more memory than other methods. The pre-order and post-order traversals are used to assign unique codes to each node. An FP-tree have no left or right subtrees. Hence the traversals, root-left-right and left-right-root, are applied as root-children and children-root without any complexity. The pruning strategies used by N-list and Nodesets with set enumeration tree is proved as an efficient method to reduce the running time.

4 Problems and Directions

The Frequent Itemset Mining algorithms proposed so far have some drawbacks associated with them. Detailed study of the seven prominent Frequent Pattern Mining algorithms lead to the following conclusions:

The data structure array is used to construct the **linear prefix tree** (LP- tree). The recursive creation of the LP-tree is one of the difficulties faced by this algorithm. To overcome this drawback, after creating the LP-tree, any other method can be used for mining process. Theoretically the tree creation process is simple, but certain complexities are inherent in array representation. LPNs are used to store the transactions. Unique strings are used in header fields of each LPN to represent

ISSN 2007-9737

Table 6. Features of the seven algorithms

Algori- thms Author year	Data structures	Concept	Techniques	Algorithms compared with
Pre-Post+ [9]	PPC tree. Set Enumeration Tree. N-list. PPC tree is same as FP-tree but nodes with two extra fields for post-order and pre-order code.	An efficient N-lists-based algorithm for mining frequent itemsets via Children-Parent Equivalence pruning. Used FP-tree structure to construct PPC-tree and Apriori method to mine frequent Items from N-list without generating candidate itemsets. Set Enumeration Tree is used to represent Candidates.	PPC tree is used to construct the N-list Data structure. Upto frequent 2-itemsets PPC-tree is used. N-list is used to mine (¿2)-itemsets. Set enumeration tree is used for candidate pruning.	PrePost. FIN. FP-growth*.
FIN [8]	POC tree, Nodeset and set enumeration tree. POC-tree is same as FP-tree but nodes with one extra fields for post-order or preorder code.	Fast mining frequent itemsets using Nodesets. Used FP-tree structure to construct PPC tree and Apriori method to mine frequent Items from Nodeset.	POC-tree is used to construct the Nodeset and to mine upto frequent 2-itemsets. Nodeset is used to mine (¿2)-itemsets. Set enumeration tree is used for candidate pruning.	PrePost. FP-growth.
PrePost [7]	PPC tree. N-list. PPC tree is same as FP-tree but nodes with two extra fields for postorder and preorder code.	A new algorithm for fast mining frequent itemsets using N-lists. Used FP-tree structure to construct PPC tree and Apriori method to mine frequent Items from N-list without generating candidate itemsets.	PPC tree is used to construct the N-list Data structure. Upto frequent 2-itemsets PPC-tree is used. N-list is used to mine (¿2)-itemsets.	FP-growth. dEclat. FP-growth*. eclat_goethals.

ISSN 2007-9737

Table 7. Features of the seven algorithms

Algorithms Author year	Data structures	Concept	Techniques	Algorithms compared with
LP-Growth [19]	LP-tree: FP-tree structure reconstructed by using arrays. Branch Node List(BNL) based on Linked List structure.	Efficient frequent pattern mining based on linear prefix tree constructed by using arrays. Avoids complex pointers used in FP-tree. Efficient Searching and insertion.	Construct the LP-Tree by using arrays and to keep track of branched nodes another dynamic linear pointer structure like linked list(BNL) is used. After tree construction BNL is deleted. LP-growth is performed on LP-tree.	FP-growth. FP-growth*. MAFIA-F. FP-goethals. FP-growth-tiny. CT-PRO
Adaptive Mining. [27]	Bit matrix. Frequent Pattern List(FPL). Transaction Pattern List(TPL)	An adaptive approach to mine frequent itemsets by switching between two data structures ie. FPL and TPL based on Database density.	After first scan to find frequent 1-itemsets construct the bit string matrix and then use the FPL. During the recursive mining process test the density of the conditional database. If the database is dense switch to FPL data structure.	FPL-Mining. TPL-Mining.
IFP-growth [17]	FP-tree with address table. FP-tree^{+}.	An improved frequent pattern growth method for mining association rules by introducing an address table and new mining methods.	Address table attached with each node of the FP-tree to improve the search of children and also improve the tree construction time. FP-tree+ is constructed to apply the mining method.	FP-growth. nonordfp.
FIUT [26]	FIUT tree- the leaves are at the same height. Clusters to store transactions with same number of items.	A new method for mining frequent itemsets by decomposing the tree. Avoids recursive creation of conditional FP-trees.	After second scan clusters of transactions with same number of items are formed. Next create the first FIUT tree from the cluster with maximum number(m) of transactions. Mine from the FIUT tree, then decompose the tree to smaller branches and insert the transactions from next lowest(m-1) numbered cluster.	FP-growth.

the parent of that LPN. Array can hold a group of data with similar type. Hence in all other fields only strings can be used. To store the details of each item as i_n, S, L, B (item, support, node link, branch information) the authors did not specify how each field of the LPN is used. The details of an item is stored in a single field of the LPN as a single string. Each details (i_n,S,L,B) can be separately represented by using delimiters. But it is time consuming procedure to access each value separately from one single field of an LPN.

The parent of a LPN is accessed by using the string value which is stored in parent field of the LPN. The location cannot be directly accessed. It is easy to access a location by using a pointer. But to access the locations specified by using strings extra processing should be done. LP-tree structure is suggested to avoid the complexity of pointers. But to implement pointers by using strings and arrays, the length of the algorithm will grow to unmanageable size.

The algorithm **FIUT** is very effective while comparing with the FP-tree. But FIUT scans the database three times ie. two database scans and one cluster scan. Comparing with other methods FIUT method have some additional processing such as creating the clusters, decomposing the tree etc. If the number of transactions in each group are very less, the total number of groups will increase. It leads to significant increase in computing time. The mining and FIUT creation are done simultaneously. Hence the size of FIUTree is comparatively smaller than FP-tree. Only one FIUTree is loaded to memory at a time. It will improve the runtime. To speed up the runtime, efficient clustering methods can be applied with FIUT algorithm.

The third method, the **IFP growth**, uses additional memory for holding an address table with each node. Address table contains item name and pointer to its child. Since the items appear more than once in FP-tree, the same address table is repeated many times. Hence the memory consumption will increase. A new improved address table can be suggested here to reduce the memory usage such as replace item names with item codes or store only the children of the nodes in the address table or avoid duplicate address tables etc.

The fourth algorithm, an **Adaptive Method**, uses two different data structures. The database density should be calculated each time while fixing the 'density lower bound' and 'count number upper bound' values.

The last three methods (PrePost [7], FIN [8], PrePost$^+$[9]) used a node identifier to uniquely identify each node. PP-codes are used in PrePost and PrePost$^+$ to identify each node. N-infos are used in FIN to identify each node. To store the PP-codes and N-infos the algorithm demands more memory. The experimental results shows that the three methods consume more memory than FP-growth. To store the pre-order and post-order values in each node, additional memory is used. The prefix-tree used only to mine frequent 2-itemsets and to collect the node identifiers. The tree is not used for mining k(¿2)itemsets. A more simple datastructure can be suggested here to improve the running time and to reduce the memory consumption. PrePost uses a new method to mine frequent 2-itemsets without using its new data structure N-list. This method can also be applied in the basic FP-tree.

Each algorithm introduced a variety of new methods to improve the performance of the basic FP-tree algorithm. The efficiency of each method have been proved by experiments. The basic concepts and techniques used in each algorithm is summarized in table 6 and 7. The basic FP-tree structure is widely used in recent algorithms like FIN, PrePost and PrePost$^+$ without any major modifications but the mining methods have been improved. Many other recent studies have increasingly utilized the FP-tree for frequent pattern mining.

One such study presents an enhanced MFP-growth [23] algorithm that combines FP-tree* and FP-growth methods to improve efficiency. While it outperforms existing techniques in both speed and accuracy across diverse datasets, it still faces memory limitations with very large databases.Researchers can improve the efficiency of the recent algorithms by contributing some improvements in the structure of the FP-tree.

These algorithms can be modified effectively to reduce the run time and memory usage.

Since its inception, the FP-Growth algorithm has undergone numerous enhancements aimed at improving its efficiency, scalability, and adaptability to various data mining contexts. Improvements in tree structures, such as compact FP-trees (CFP-Tree) and enhanced header tables, help reduce memory usage and improve traversal speed. Several mining strategies have evolved, including top-k FP-Growth [14], which eliminates the need for a predefined minimum support threshold, and dynamic FP-Growth, which allows incremental updates as new data becomes available. To handle large-scale datasets, parallel and distributed versions like PFP [16] and DFP-Growth [1] have been introduced using frameworks such as Hadoop and Spark. Moreover, FP-Growth has been extended to support different data types and applications, including weighted FP-Growth for incorporating item importance [32], utility FP-Growth for high-utility itemset mining [31], and temporal FP-Growth for time-sensitive pattern discovery . Privacy-preserving and incremental variants have also emerged to meet the needs of modern data environments. These advancements make FP-Growth a versatile and powerful tool in the field of frequent pattern mining.

5 Conclusion

Frequent pattern mining finds the patterns which are appearing frequently in a transaction database. Association rule mining is used to find the relationship among the frequent patterns.

In this paper a basic idea of Frequent pattern mining and association rule mining are illustrated. A detailed study of the FP-Tree structure and FP-growth algorithm is also presented. An overview of six prominent improvements over FP-growth algorithm Viz.

PrePost [7], FIN [8], PrePost^{+} [9], LP-growth [19] and IFP-growth [17] are given. Further, an algorithm without FP-tree is also presented [27]. Finally, the merits and demerits of the seven algorithms are analyzed.

References

1. **Abdullah, Z., Herawan, T., Noraziah, A., Deris, M. M. (2012).** Dfp-growth: an efficient algorithm for mining frequent patterns in dynamic database. Information Computing and Applications: Third International Conference, ICICA 2012, Chengde, China, September 14-16, 2012. Proceedings 3, Springer, pp. 51–58.
2. **Agrawal, R., Imieliński, T., Swami, A. (1993).** Mining association rules between sets of items in large databases. ACM SIGMOD Record, ACM, Vol. 22, No. 2, pp. 207–216.
3. **Agrawal, R., Srikant, R., et al. (1994).** Fast algorithms for mining association rules. Proc. 20th int. conf. very large data bases, VLDB, Vol. 1215, pp. 487–499.
4. **Aldino, A. A., Pratiwi, E. D., Sintaro, S., Putra, A. D., et al. (2021).** Comparison of market basket analysis to determine consumer purchasing patterns using fp-growth and apriori algorithm. 2021 International Conference on Computer Science, Information Technology, and Electrical Engineering (ICOMITEE), IEEE, pp. 29–34.
5. **Asif, M., Ahmed, J. (2015).** Analysis of effectiveness of apriori and frequent pattern tree algorithm in software engineering data mining. Intelligent Systems, Modelling and Simulation (ISMS), 2015 6th International Conference on, IEEE, pp. 28–33.
6. **Borgelt, C. (2005).** An implementation of the fp-growth algorithm. Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations, pp. 1–5.
7. **Deng, Z., Wang, Z., Jiang, J. (2012).** A new algorithm for fast mining frequent itemsets using n-lists. Science China Information Sciences, Vol. 55, No. 9, pp. 2008–2030.
8. **Deng, Z.-H., Lv, S.-L. (2014).** Fast mining frequent itemsets using nodesets. Expert Systems with Applications, Vol. 41, No. 10, pp. 4505–4512.

9. **Deng, Z.-H., Lv, S.-L. (2015).** Prepost+: An efficient n-lists-based algorithm for mining frequent itemsets via children–parent equivalence pruning. Expert Systems with Applications, Vol. 42, No. 13, pp. 5424–5432.

10. **Ebrahimi, M., Bazyar, M. A., Tahmasbi, M., Boostani, R. (2008).** Benefiting from data mining techniques in a hybrid peer-to-peer network. Advanced Computer Theory and Engineering, 2008. ICACTE'08. International Conference on, IEEE, pp. 499–502.

11. **Grahne, G., Zhu, J. (2005).** Fast algorithms for frequent itemset mining using fp-trees. Knowledge and Data Engineering, IEEE Transactions on, Vol. 17, No. 10, pp. 1347–1362.

12. **Han, J., Kamber, M., Pei, J. (2011).** Data mining: concepts and techniques. Elsevier.

13. **Han, J., Pei, J., Yin, Y. (2000).** Mining frequent patterns without candidate generation. ACM SIGMOD Record, ACM, Vol. 29, No. 2, pp. 1–12.

14. **Han, J., Wang, J., Lu, Y., Tzvetkov, P. (2002).** Mining top-k frequent closed patterns without minimum support. 2002 IEEE International Conference on Data Mining, 2002. Proceedings., IEEE, pp. 211–218.

15. **Heaton, J. (2016).** Comparing dataset characteristics that favor the apriori, eclat or fp-growth frequent itemset mining algorithms. SoutheastCon 2016, IEEE, pp. 1–7.

16. **Li, H., Wang, Y., Zhang, D., Zhang, M., Chang, E. Y. (2008).** Pfp: parallel fp-growth for query recommendation. Proceedings of the 2008 ACM conference on Recommender systems, pp. 107–114.

17. **Lin, K.-C., Liao, I.-E., Chen, Z.-S. (2011).** An improved frequent pattern growth method for mining association rules. Expert Systems with Applications, Vol. 38, No. 5, pp. 5154–5161.

18. **Liu, G., Lu, H., Yu, J. X. (2007).** Cfp-tree: A compact disk-based structure for storing and querying frequent itemsets. Information Systems, Vol. 32, No. 2, pp. 295–319.

19. **Pyun, G., Yun, U., Ryu, K. H. (2014).** Efficient frequent pattern mining based on linear prefix tree. Knowledge-Based Systems, Vol. 55, pp. 125–139.

20. **Rácz, B. (2004).** nonordfp: An fp-growth variation without rebuilding the fp-tree. FIMI.

21. **Ruggieri, S. (2010).** Frequent regular itemset mining. Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, pp. 263–272.

22. **Rymon, R. (1992).** Search through systematic set enumeration. Technical Reports (CIS), pp. 297.

23. **Shawkat, M., Badawi, M., El-ghamrawy, S., Arnous, R., El-desoky, A. (2022).** An optimized fp-growth algorithm for discovery of association rules. The Journal of Supercomputing, Vol. 78, No. 4, pp. 5479–5506.

24. **Singh, P. K., Othman, E., Ahmed, R., Mahmood, A., Dhahri, H., Choudhury, P. (2021).** Optimized recommendations by user profiling using apriori algorithm. Applied Soft Computing, Vol. 106, pp. 107272.

25. **Tanbeer, S. K., Ahmed, C. F., Jeong, B.-S., Lee, Y.-K. (2008).** Cp-tree: a tree structure for single-pass frequent pattern mining. In Advances in Knowledge Discovery and Data Mining. Springer, pp. 1022–1027.

26. **Tsay, Y.-J., Hsu, T.-J., Yu, J.-R. (2009).** Fiut: A new method for mining frequent itemsets. Information Sciences, Vol. 179, No. 11, pp. 1724–1737.

27. **Tseng, F.-C. (2012).** An adaptive approach to mining frequent itemsets efficiently. Expert Systems with Applications, Vol. 39, No. 18, pp. 13166–13172.

28. **Tseng, F.-C., Hsu, C.-C. (2001).** Generating frequent patterns with the frequent pattern list. In Advances in Knowledge Discovery and Data Mining. Springer, pp. 376–386.

29. **Tseng, F.-C., Hsu, C.-C., Fu, K.-S. (2005).** Efficiently mining frequent closed itemsets by eliminating data redundancies. Electronic Commerce Studies, Vol. 3, No. 1, pp. 39–56.

30. **Tseng, F.-C., Hsu, C.-C., Fu, K.-S. (2005).** The frequent pattern list: Another framework for mining frequent patterns.. IJEBM, Vol. 3, No. 2, pp. 104–115.

31. **Tseng, V. S., Wu, C.-W., Shie, B.-E., Yu, P. S. (2010).** Up-growth: an efficient algorithm for high utility itemset mining. Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 253–262.

32. **Wang, W., Yang, J., Yu, P. S. (2000).** Efficient mining of weighted association rules (war). Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 270–274.

Article received on 17/10/2023; accepted on 18/02/2025.
**Corresponding author is O. Jamsheela.*