INDIAN INSTITUTE OF TECHNOLOGY

ROORKEE

Project Report for Programming Laboratory

(Implementation of Apriori and Frequent Pattern Growth Algorithm)

Submitted To:

Ms. Durga Toshniwal

Associate Professor (CSE)

Submitted By:

Amala Thampi	14535003
Nikita Jain	14535031
Nishtha Behal	14535032
Preeti Bansal	14535035
Yashika Jain	14535054

Apriori Algorithm:

Association rules mining is the main task of data mining. An association rule, X => Y, is a statement of the form "for a specified fraction of transactions, a particular value of an attribute set X determines the value of attribute set Y as another particular value under a certain confidence". Thus, association rules aim at discovering the patterns of co-occurrences of attributes in a database. There are many algorithms presented to generate association rules, but all of these algorithms are variations of the *Apriori* algorithm, which is the state of the art.

For generation of association rules Apriori requires two values to be specified:

1. Minimum Support
2. Confidence Threshold

These are defined as:

Support

The support supp(X) of an itemset X is defined as the proportion of transactions in the data set which contain the itemset.

supp(X) = no. of transactions which contain the itemset X / total no. Of transactions

Confidence

The confidence of a rule is defined:

Conf(A=>B)= supp(AUB)/supp(A)

Apriori requires a priori knowledge to generate the next generation of candidate itemsets, i.e., it generates the candidate (k+1)-itemsets from the frequent k-itemsets. In addition, it depends on the *Apriori fact* : "***all the subsets of a frequent itemset are frequent too***".

Hence, Apriori generates candidate itemsets that must pass two exams, to be frequent itemsets. The pruning step depends on the apriori fact. After generating a candidate (k+1)-itemset, it will be degenerated to its k-itemsets subsets. If any one of these subsets is not large, i.e., is not member in L_k , the candidate will be removed and regarded as small or infrequent. The support of the candidate itemset, which delivered from the first pruning, is counted in the second pruning step. If this support exceeds the minimum support, then the candidate will be regarded as frequent itemset. The support counting is accomplished by |D|

database scanning operation where |D| is the number of transactions in the database under consideration.

Example:

Assume the user-specified minimum support is 40%, then generate all frequent itemsets.

Solution:

The transaction database shown below

Pass 1:

C_1

Itemset X	supp(X)
A	?
B	?
C	?
D	?
E	?

L_1

Itemset X	supp(X)
A	100%
B	60%
C	100%
D	80%
E	40%

Pass 2:

C2

Itemset X	supp(X)
A,B	?
A,C	?
A,D	?
A,E	?
B,C	?
B,D	?
B,E	?
C,D	?
C,E	?
D,E	?

- Nothing is pruned since all subsets of these itemsets are frequent

L_2

Itemset *X*	supp(X)
A,B	60%
A,C	100%
A,D	80%
A,E	40%
B,C	60%
B,D	40%
B,E	20%
C,D	80%
C,E	40%
D,E	40%

L_2 after saving only the frequent itemsets

Itemset *X*	supp(X)
A,B	60%
A,C	100%
A,D	80%
A,E	40%
B,C	60%
B,D	40%
C,D	80%
C,E	40%
D,E	40%

Pass 3:

- To create C3 only look at items that have the same first item (in pass k, the first k - 2 items must match)

C_3

	Itemset *X*	*supp(X)*
join AB with AC	A,B,C	?
join AB with AD	A,B,D	?
join AB with AE	A,B,E	?
join AC with AD	A,C,D	?
join AC with AE	A,C,E	?
join AD with AE	A,D,E	?
join BC with BD	B,C,D	?
join CD with CE	C,D,E	?

C_3 after pruning

Itemset *X*	supp(X)
A,B,C	?
A,B,D	?
A,C,D	?
A,C,E	?
A,D,E	?
B,C,D	?
C,D,E	?

- Pruning eliminates ABE since BE is not frequent

- Scan transactions in the database

L3

Itemset *X*	supp(X)
A,B,C	60%
A,B,D	40%
A,C,D	80%
A,C,E	40%
A,D,E	40%
B,C,D	40%
C,D,E	40%

Pass 4:

- First k - 2 = 2 items must match in pass k = 4

C4

	Itemset *X*	*supp(X)*
combine ABC with ABD	A,B,C,D	?
combine ACD with ACE	A,C,D,E	?

- Pruning:
    - For ABCD we check whether ABC, ABD, ACD, BCD are frequent. They are in all cases, so we do not prune ABCD.
    - For ACDE we check whether ACD, ACE, ADE, CDE are frequent. Yes, in all cases, so we do not prune ACDE

L4

Itemset *X*	supp(X)
A,B,C,D	40%
A,C,D,E	40%

- Both are frequent

Pass 5:

For pass 5 we can't form any candidates because there aren't two frequent 4-itemsets beginning with the same 3 items

APRIORI PSEUDOCODE:

Input:

The algorithm reads its input from a text file containing the data set. It also specifies the minimum support and confidence threshold values.

Variables used:

t: number of transactions in the data set.

n: number of distinct data items.

C_k: Candidate itemset of size k

L_k: frequent itemset of size k

Data Structures used:

1. A two dimensional integer matrix trans[t][n] to hold the transaction set. We assign numbers to all the data items starting from one. If transaction i has the data item j in its data set then trans[i-1][j]=1.
2. A three dimensional matrix to hold the frequent item sets.

Algorithm:

1. Input is read into the two dimensional matrix from the input file.
2. Make a call to the function APRIORI(). This function makes generates the frequent itemsets.
3. Call RULES_GENERATE(). This function is used to generate rules from the frequent itemsets generated in step 2.

Methods Used:

APRIORI()

1. The input matrix is used to generate the 1 item frequent itemsets.
2. PAIRING() function is called after this where rest of the k-itemsets are generated. Here k varies from 2 to until there are no new frequent itemsets at a particular level.
3. The initial call is PAIRING(L_1).

PAIRING(L_k itemset)

1. This function uses the (k-1)-itemsets ie L_{k-1} to generate the k-item candidate itemsets ie C_k.
2. $C_k = \phi$
3. for all itemsets *X* in L_{k-1} and Y in L_{k-1} do
4. if $X_1 = Y_1 \cap \ldots \cap Xk\text{-}2 = Yk\text{-}2 \cap Xk\text{-}1 < Yk\text{-}1$
5. then C = X1X2 ... Xk-1Yk-1
6. add C to C_k
7. Calculate count for each of the itemsets generated ie for each itemset in C_k.
8. Remove those itemsets from C_k for which count is below the minimum support.
9. Now we need to prune itemsets from C_k based on the apriri property ie "***all the proper subsets of a frequent itemset are also frequent***".
10. For this we call the SUBSET() function.
11. The subset function returns the k item frequent itemsets ie L_k by taking as input the k item candidate sets.
12. if L_k is not empty then
13. Call PAIRING(L_k)
14. else stop.

SUBSET(C_k)

1. Generate all possible subsets for C_k using a list data structure.

2. Check if these occur in previous itemsets with size k-1 in array ar_4_ap[].

3. If all subsets occur than return 1.

4. else if any one of the subset also doesnt occur return 0 and make the values in ar_4_ap[] for that itemset as all 0's.

RULES_GENERATE()

1. Let R denote the set of rules. Initially R is an empty set.
2. Traverse each frequent itemset generated ie each L_k where k>1.
3. For each itemset X in L_k

4. Generate its proper subsets.
5. for each subset calculate confidence using formula:
   confidence (A ⇒ B) = P(B|A) = support count (A ∪ B)/support count (A)
   here A: set of the items present in this subset.
   B: X-A
6. If the confidence is greater then the threshold then add the rule A=>B to R.
7. Print the set of rules R.

ANALYSIS:

The time needed by apriori apriori algorithm is determined by the number of itemsets that are output and the number of itemsets that are counted but not output. In best case the candidacy test always correctly predicts the result of the frequency test and the amount of time spent by the apriori algorithm is essentially the time spent verifying that all output sets should be output. The worst case time needed by apriori algorithm is polynomial in the sum of the size of input plus output. Since the worst case output is exponenetially larger than the input , the worst case time needed can be exponential in the size of input.

Frequent Pattern Growth Algorithm:

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 database separately. For each “pattern fragment,” only its associated data sets need to be examined. Therefore, this approach may substantially reduce the size of the data sets to be searched, along with the “growth” of patterns being examined.

First, we design a novel data structure, called frequent pattern tree , or FP-tree for short, which is an extended pre x-tree structure storing crucial, quantitative information about frequent patterns. To ensure that the tree structure is compact and informative, only frequent length-1 items will have nodes in the tree. The tree nodes are arranged in such a way that more frequently occurring nodes will have better chances of sharing nodes than less frequently occurring ones. Our experiments show that such a tree is highly compact, usually orders of magnitude smaller than the original database. This o ers an FP-tree -based mining method a much smaller data set to work on.

Second, we develop an FP-tree -based pattern fragment growth mining method, which starts from a frequent length-1 pattern #as an initial su x pattern #, examines only its conditional pattern base #a sub-database" which consists of the set of frequent items co-occurring with the su x pattern#, constructs its # conditional # FP-tree , and performs mining recursively with such a tree. The pattern growth is achieved via concatenation of the su x pattern with the new ones generated from a conditional FP-tree . Since the frequent itemset in any transaction is always encoded in the corresponding path of the frequent pattern trees, pattern growth ensures the completeness of the result. In this context, our method is not Apriori -like restricted generation-and-test but restricted test only . The major operations of mining are count accumulation and prefix path count adjustment, which are usually much less costly than candidate generation and pattern matching operations performed in most Apriori -like algorithms.

Third, the search technique employed in mining is a partitioning-based, divide-and-conquer method rather than Apriori -like bottom-up generation of frequent itemsets combinations .

This dramatically reduces the size of conditional pattern base generated at the subsequent level of search as well as the size of its corresponding conditional FP-tree . Moreover, it transforms the problem of nding long frequent patterns to looking for shorter ones and then concatenating the sufix. It employs the least frequent items as sufix, which o ers good selectivity. All these techniques contribute to the substantial reduction of search costs.

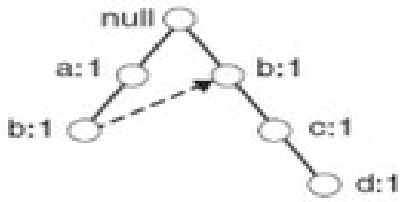
Example:

FP-Tree Construction

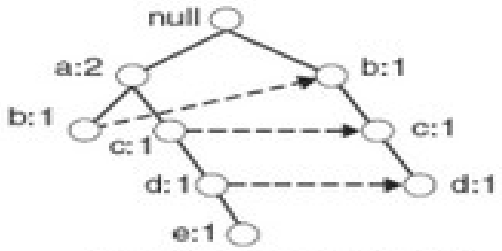
Transaction Data Set

TID	Items
1	{a,b}
2	{b,c,d}
3	{a,c,d,e}
4	{a,d,e}
5	{a,b,c}
6	{a,b,c,d}
7	{a}
8	{a,b,c}
9	{a,b,d}
10	{b,c,e}

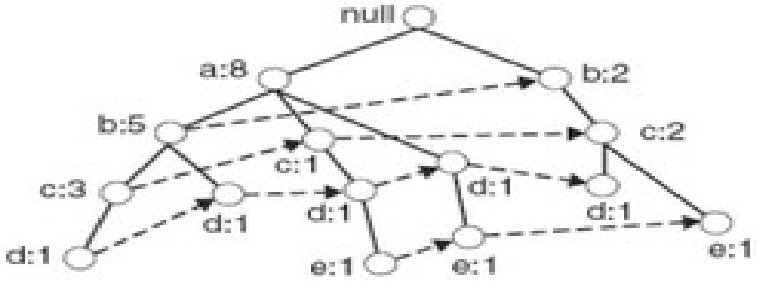
(i) After reading TID=1



(ii) After reading TID=2

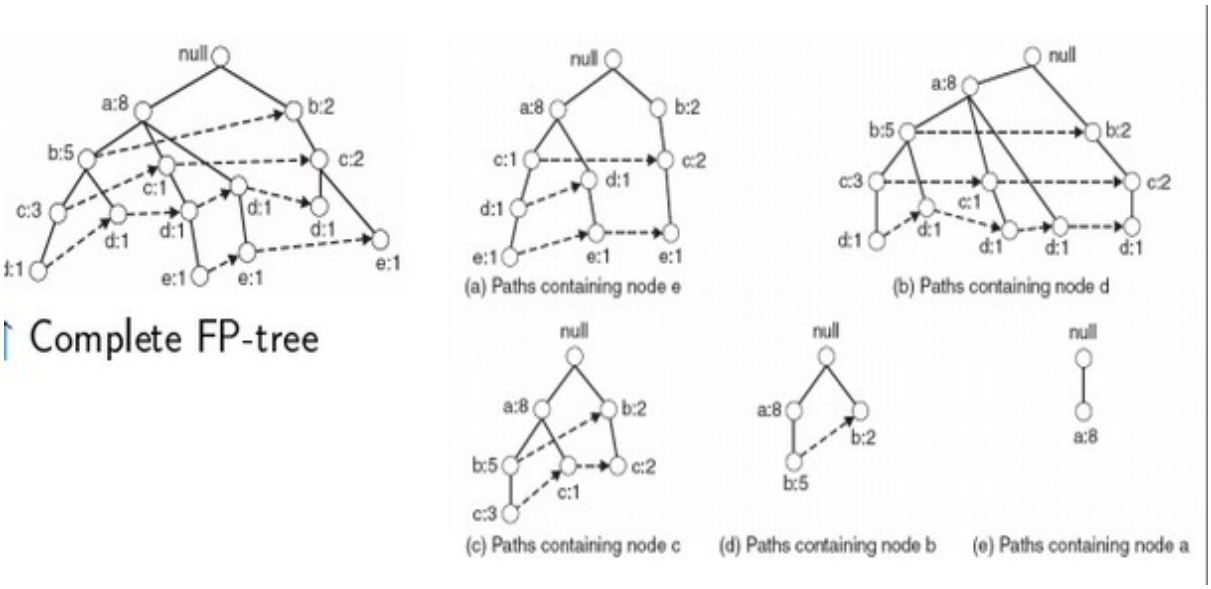


(iii) After reading TID=3

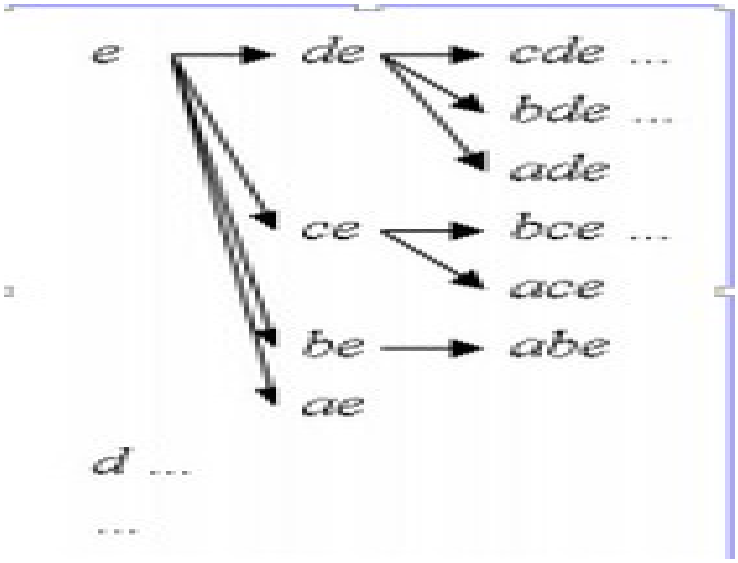


(iv) After reading TID=10

Prefix path sub-trees

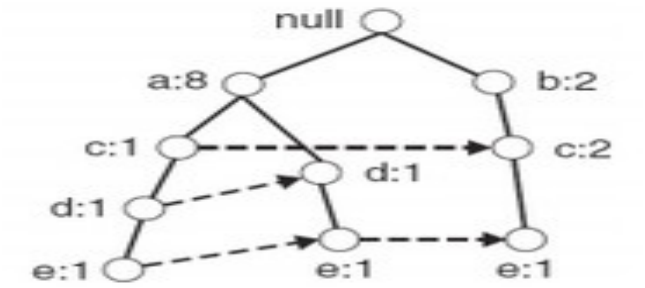


Frequent Itemset Generation



Let minSup = 2 and extract all frequent itemsets containing e.

1. Obtain the prefix path sub-tree for e:



2. Check if e is a frequent item by adding the counts along the linked list (dotted line). If so, extract it.

   Yes, count =3 so {e} is extracted as a frequent itemset.

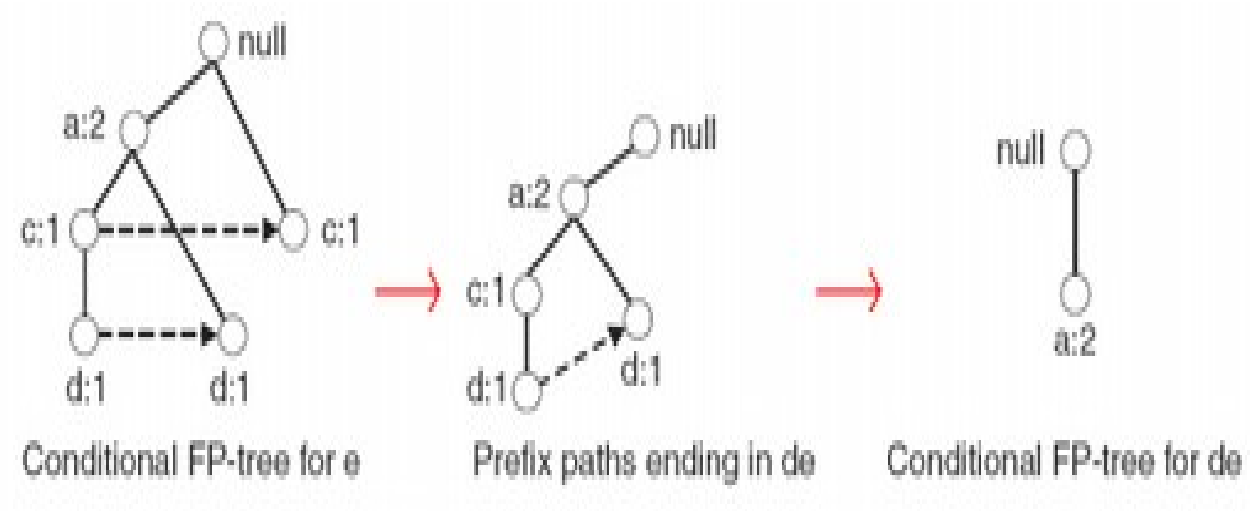
3. As e is frequent, find frequent itemsets ending in e. i.e. de, ce, be and ae.

4. Use the the conditional FP-tree for e to find frequent itemsets ending in de, ce and ae

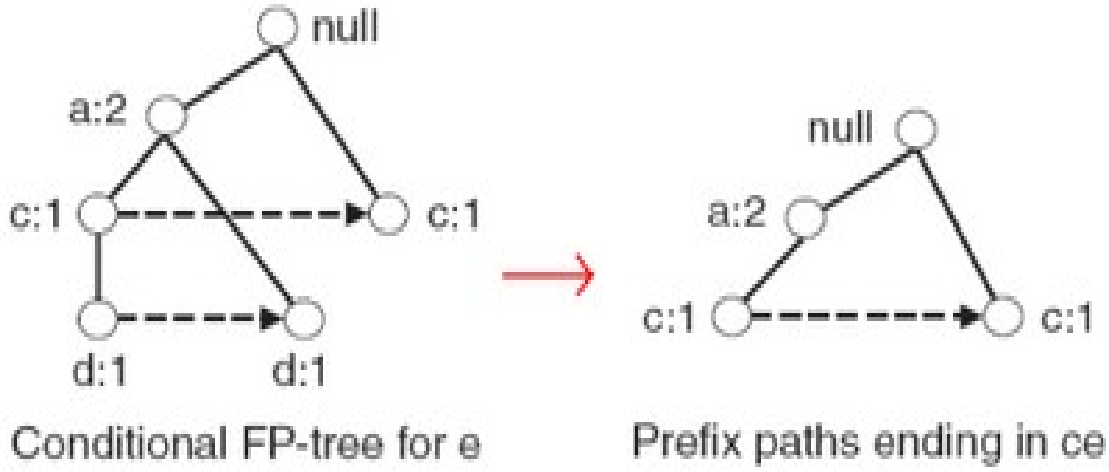
   Note that be is not considered as b is not in the conditional FP-tree for e.

I For each of them (e.g. de), find the prefix paths from the conditional tree for e, extract frequent itemsets, generate conditional FP-tree, etc... (recursive)

_Example: e -> de -> ade ({d,e}, {a,d,e} are found to be frequent)



Example: e -> ce ({c,e} is found to be frequent)



Frequent itemsets found (ordered by sufix and order in which they are found):

Transaction Data Set

TID	Items
1	{a,b}
2	{b,c,d}
3	{a,c,d,e}
4	{a,d,e}
5	{a,b,c}
6	{a,b,c,d}
7	{a}
8	{a,b,c}
9	{a,b,d}
10	{b,c,e}

Suffix	Frequent Itemsets
e	{e}, {d,e}, {a,d,e}, {c,e},{a,e}
d	{d}, {c,d}, {b,c,d}, {a,c,d}, {b,d}, {a,b,d}, {a,d}
c	{c}, {b,c}, {a,b,c}, {a,c}
b	{b}, {a,b}
a	{a}

PSEUDOCODE FOR FP-GROWTH

Algorithm:

FP growth. Mine frequent itemsets using an FP-tree by pattern fragment growth.

Input:

An array ar , a transaction database;

support , the minimum support count threshold.

Output:

The complete set of frequent patterns and association rules.

Method Used:

1. The FP-tree is constructed in the following steps:

(a) Scan the array ar[] once. Collect the set of frequent items, and their support counts in count array. Sort the array in support count descending order as array ar, the list of frequent items.

(b) Create root of FP-tree and assign its data as null. For each transaction in array ar do the following

For each individual transaction Call insert_tree(ar) to insert that transaction into tree.

(c) Call rules_generate(ar,ar1) where ar1 is also a matrix of transactions.

2. insert_tree(char ar[][])

(a) Traverse the complete transaction till value is not equal to '0' .

(b) Call trav_root(val ,par) in which the data will be inserted in the tree as follows If par has a child N such that N.data= par.data, then increment N 's count by 1; else create a new node aux ,and let its count be 1, its parent link be linked to par, and its(aux's) sibling be linked to new node aux.

3. The FP-growth is implemented as:

(a) Select each itemset from descending order from order matrix.

(b) Assign each frequent itemset in integer array of linked list and call sort() on each linked list node by traversing the path from bottom to top until we arrive at root that is null . Sort() function arranges the items of each linked list array as per the order matrix defined in the program.

(c) Append the original itemset into the the frequent itemset.

(d) Calculate Count for each itemset and match it with minimum support if found that calculated count is above minimum support than save it.

Else if delete that itemset

(c) Save these itemsets in ar_4_fp[][] matrix for future association rules generation.

4. rules_generate(char ar[][],int ar1[][])

(a) Traverse each frequent itemset generate its subsets using function subset_4_rules_f()

(b) Call rules_trav for each subset where we calculate count for AU B as c_major_count and count of each subset using array ar1[] . We calculate confidence using formula

$$\text{confidence } (A \Rightarrow B) = P(B|A) = \frac{\text{support count } (A \cup B)}{\text{support count } (A)}$$

here A: set of the items present in this subset.

B: X-A

(c) If the confidence is greater then the threshold then add the rule A=>B to R

(c) Print each association rules.

ANALYSIS

Each path in the tree will be at least partially traversed the number of items existing in that tree path (the depth of the tree path) * the number of items in the header. Complexity of searching through all paths is bounded by $O(header\_count^2 * depth\_of\_tree)$.