CHAPTER 4

FREQUENT ITEMSET DISCOVERY USING LLT ALGORITHM

4.1 Introduction

After preprocessing, the transactional data takes a ready form for mining. A new and efficient algorithm is developed and implemented in this work, for finding the frequent itemsets, which is the resource consuming, and heavily researched area of Association rule mining. Various algorithms were devised in this area [14] [17] [41] [45][47] [56][60] [71][85][86] [98]. Frequent itemset discovery is the second module of the model and is highlighted in the figure 4.1.

The new algorithm developed is a tree based algorithm having three levels. Therefore it is named as Limited Level Tree –LLT algorithm. The main objective of developing this Limited Level Tree algorithm is to reduce the repeated database scans and the cost of computation involved in generating frequent itemsets. Another major objective in developing the LLT algorithms is that it should support real time applications.

LLT algorithm is composed of two steps. First step is creation of tree data structure. Second step is mining the frequent itemsets from the tree structure.
Figure 4.1 Frequent Itemset Discovery module
4.2 Constructing LLTree structure.

First step of LLT algorithm is creation of data structure. This data structure is a three level tree named Limited Level Tree (LLTree). LLTree structure is compact and memory efficient. Only one scan of the given transactional database is enough to create this tree. LLT algorithm first generates the itemsets for every transaction in the input database. Then these itemsets are stored in the appropriate nodes of the tree.

4.2.1 Candidateset generation

Input to the LLT algorithm is the $m \times n$ transaction table $T$ containing the information transaction id, item1, item2... Itemn. $m$ is number of transaction in the table and $n$ is the number of items. Any transaction can hold at the most $n$ number of items. Transactions in the database are read one by one. After reading a transaction from the database all possible $k$ cardinality itemsets of that transaction are generated, where $1 \leq k \leq n$. The size of an itemset is defined as its cardinality; an itemset containing $k$ items is called a $k$-itemset. A unique and efficient algorithm is devised and used for generating all the $k$-itemsets. It takes transaction as the input and produces all the $k$-itemsets of that transaction as the result. These itemsets are the candidatesets here.

Maximum combinations of itemsets that can be generated for any length $k$ is $nC_k$. For any transaction the maximum number of itemsets generated could be
\[ nc_1 + nc_2 + \ldots + nc_n \]  \hspace{1cm} (4.1)

Which can be expressed as
\[ \sum_{r=1}^{n} nc_r \]  \hspace{1cm} (4.2)

Expression (4.2) can be equated to \( 2^n - 1 \). So all the \( k \)-itemsets \( I_1, I_2, \ldots, I_k \) where \( 1 \leq k \leq n \), obtained for any transaction \( T \) is
\[ \sum_{r=1}^{n} nc_r = 2^n - 1. \]  \hspace{1cm} (4.3)

\( 2^n - 1 \) is the maximum number of itemsets for any transaction \( T \).

**4.2.2 Tree structure**

Structure of LLTree is shown in the figure 4.2. First level of the tree is a single header node. It is just a pointer node to its children. Second level of the tree has \( n \) subtrees labeled from 1 to \( n \). The label represents the length of the itemsets that particular subtree stores. If the label is some \( k \) then all \( k \)-itemsets are stored in that subtree as children in the third level. Maximum length of an itemset in an \( n \) transactional data is \( n \). Therefore, the labels range from 1 to \( n \) and the maximum number of subtrees in level two is \( n \). The leaf nodes at the level three contain two fields. The itemsets itself is stored in one field and next field stores the occurrence count of that itemset. The storage format is

\[ <\text{itemset}><\text{occurrence count}> \]

When an itemset is stored for the first time, a new node is created and the occurrence count is set to 1. On its repeated arrivals, only the count gets incremented.
4.2.3 Storage process

After generating itemset, it must be stored in the appropriate subtrees. The storages process is as follows.

1. To store a k-length itemset, reach the subtree with label k.
2. Search for the existence of that itemset.
3. If exists, increment the occurrence count. Go to Step 6.
4. If not, create a new child node for the subtree labeled k.
5. Store the itemset in that child node. Set the occurrence count to 1.
6. Repeat step 1 to 5 for all the itemsets generated for a transaction.
In one scan, the given transactional database is stored in the tree structure. It is a compressed storage of the entire transactional database. Once the tree is constructed, the original web log database is no longer required for subsequent processing.

**4.2.4 Size of the tree**

From expression (4.3), it is clear that when all \( n \) items are present in a transaction, the total number of all \( k \)-itemsets generated is

\[
T_{\text{trans}} = \sum_{r=1}^{n} nC_r = 2^n - 1. \tag{4.4}
\]

All the itemsets are stored in the level three of LLTree. So the maximum nodes needed at level three is

\[
L_3 = 2^n - 1. \tag{4.5}
\]

Level two of LLTree needs \( n \) nodes and at level one number of nodes is one.

\[
L_2 = n \tag{4.6}
\]

\[
L_1 = 1 \tag{4.7}
\]

The maximum nodes needed for an LLTree structure for an \( m \times n \) transactional database can be computed from (4.5), (4.6) and (4.7).

\[
\text{Tot nodes} \leq 1 + n + 2^n - 1
\]

\[
\leq 2^n + n. \tag{4.8}
\]

\( 2^n + n \) are the maximum nodes needed for an \( m \times n \) transactional database. Here the size of the tree remains the same irrespective of number of transactions.
The algorithm for LLTree construction is given below.

**Algorithm 4.1. LLTree construction**

ALGORITHM 1: Limited Level Tree (Transaction)

*Input*: Transactions

*Output*: Limited Level Tree

*T*: Transaction

*Ci*: Candidate set's of *T*

1. Begin
2. FOR each element in *Ci* DO
3. `CandidateLength`: Number of item count in *Ci*
4. Look for the subTree with label `CandidateLength` for process
5. IF no subTree with label `CandidateLength` THEN
   1. Create a new subTree and label it as `candidateLength`
   2. Insert the candidate item *Ci* as node;
   3. Initiate the occurrence count
6. Select the tree with label `CandidateLength`
   1. Update the occurrence count of *Ci* node
7. End IF
8. End
4.2.5 Complexity analysis for LLTree storage

Total cost \( C_{\text{ins}} \) of inserting an itemset \( I \) of length \( k \) in the LLTree structure is

\[
C_{\text{ins}} = O(|I_k|) + O(|t_{\text{ser}}|) \tag{4.9}
\]

\[
= O(|I|)
\]

\[
C_{\text{ins}} = O(|k|) \tag{4.10}
\]

Where \( t_{\text{ser}} \) is the time taken to search the particular node in the tree.

Complexity of constructing the LLTree from the scratch is equivalent to the cumulative cost \( CC_{\text{cum}} \) of inserting all \( k \)-length itemsets \( I_1, I_2, I_3...I_k \) where \( 1 \leq k \leq n \) is

\[
CC_{\text{cum}} = O(|I_1|) + O(|I_2|) + O(|I_3|)+...... + O(|I_k|) \tag{4.11}
\]

\[
= O(n)
\]

Thus

\[
CC_{\text{cum}} = O(n) \tag{4.12}
\]
4.3 Mining frequent itemsets from LLTree.

The second step of LLT algorithm is mining the frequent itemsets from the tree structure. If the support of an itemset is greater than or equal to a given support threshold $\text{minsup}$, then that itemsets are called frequent itemsets. It is a very simple process. The algorithm accepts the minimum support threshold $s$ as input and retrieves all the itemsets whose occurrence count is greater than or equal to the given minimum threshold. The process of mining the frequent itemsets can be expressed as

Step 1. Input the minimum support threshold $s$.
Step 2: for all substrees $k = 1…n$ at level 2 do the following
  i. Visit the nodes of subtree $k$ at level three.
  ii. Compare the occurrence count part of the node with the support threshold $s$.
  iii. If it is greater than equal to $s$ then return the itemset in that node with its occurrence count.
  iv. Repeat until $k = n$.

This process can be repeated any number of times with different threshold levels. There is no need to scan the database again to compute the occurrence count. The algorithm for mining frequent itemsets from the LLTree structure is given below.
Algorithm 4.2. Frequent itemset generation

Input: Minimum support

Output: Frequent Itemsets

FrequentItems List = Φ

1 Begin
2 FOR each subTree Ti DO
3 FOR each CandidateSetsCi in the Tree Ti DO
4 IF Ci occurrence count ≥ MinimumSupport THEN
5 Add Ci to FrequentItems List
6 END FOR
7 END FOR
8 Write FrequentItems List
9 END

4.3.1 Complexity analysis for fetching frequent itemsets from LLTree.

The cost of fetching all the k-length itemsets from the LLTree for the given support level s is the cumulative searching time of all the sub nodes of all the level two nodes. Number of nodes at level two is n and level three is $2^n-1$. Total cost of fetching frequent itemsets is

$$T_c = O(n) + O(2^n-1) \quad (4.13)$$
4.4 Illustration

LLT algorithm can be explained with an example. Let the incoming transaction be with three items

\[
123 \quad (1)
\]

The transaction table is:

<table>
<thead>
<tr>
<th>Transaction no.</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 2 3</td>
</tr>
</tbody>
</table>

The first phase of the algorithm is LLTree construction. Candidate itemsets of incoming transactions are generated. Accordingly all k-itemsets of the transaction (1) are

- 1-itemset - 1, 2, 3
- 2-itemset - 12, 13, 23
- 3-itemset - 123

Since \( n = 3 \) here, from (2), the maximum number of candidate itemsets could be

\[
2^{n-1} = 2^3 - 1 = 7.
\]

The 1-itemset of the transactions are stored in the subtree labeled 1 and 2-itemset in the subtree labeled 2. Similarly, other itemsets are
also stored in their respective sub trees. The insertion procedure in to the LLTree is depicted in the figure 4.3.

Next incoming transaction is

\[ 1 \, 3 \, 4 \]  

(2)

After the arrival of second transaction the resultant table:

<table>
<thead>
<tr>
<th>Transaction no.</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 , 2 , 3</td>
</tr>
<tr>
<td>2</td>
<td>1 , 3 , 4</td>
</tr>
</tbody>
</table>

The candidate itemsets generated for transaction (2) are:

1-itemset - 1 \, 3 \, 4
2-itemset - 13, 14, 34
3-itemset - 134

The same procedure is followed for the transaction (2) also and the resultant tree is shown in the figure4.4.
Figure 4.3 – LLTree construction
Mining frequent itemsets from the tree structure is the next phase of the algorithm. The retrieval procedure is depicted in the Figure 4.5.
Figure 4.5. Mining frequent itemsets from the tree
4.5 Comparative study

Various algorithms for finding frequent itemsets are discussed earlier in the survey chapter. To compare with the LLT algorithm, Apriori and FP-growth algorithms are considered. Apriori is the foremost and popular algorithm for finding frequent itemsets [2]. FP-growth (Frequent Pattern growth) algorithm [44] faster than others in the literature, is regarded as a landmark in mining frequent patterns. FP-growth uses a compact tree structure called FP tree. These algorithms differ in their search strategy, search direction and counting strategy.

4.5.1 Search strategy

Search strategy refers to the order in which the itemsets are visited. Search strategy can be depth first search or breadth first search. Some proposed algorithms combine both approaches. Apriori and Partition algorithms use breadth first strategy while FP-growth and Eclat use depth first strategy. LLT algorithm uses depth first strategy. However, the levels to search are very much limited, since the depth is at most three levels.

4.5.2 Counting Strategy

Counting Strategy refers to the methods used to count the occurrences of candidate itemsets. Up to date, there are two main approaches: horizontal counting and vertical intersection. The horizontal counting determines the support value of a candidate itemset by scanning transaction one by one, and increasing the counter of the
itemset if it is a subset of the transaction. This is costly for candidates of large size. Vertical intersection, on the other hand, is employed when the database is represented as a vertical format such that each record is associated with an item to store the identifiers of the transactions containing that item, called tidlist. Though the vertical intersection scheme eliminates the I/O cost for database scan, it has the following deficiency: When the support count of a candidate itemset is quite less than the number of transactions, there occur a large amount of unnecessary intersections.

Apriori and FP-growth employ horizontal counting strategies while Eclat follows vertical intersection. LLT does a single scan of the database and generates candidate itemsets, which are inserted in the tree. It is vertical insertion procedure. The I/O cost for data base scan is eliminated here. The unnecessary insertions as in the other vertical insertion algorithms are not present in the LLT.

4.5.3Search direction

Search direction guides the way the search space is exploited. This can be top down or bottom up. Apriori approach and FP-growth adopt bottom-up traversal of the search space, starting from all frequent 1-itemsets upward to the longest frequent itemsets. Top down strategy is traditionally adopted for discovering maximal frequent itemsets [88]. LLT algorithms search frequent 1–itemsets first then proceed to the next longest frequent itemsets. Therefore, LLT algorithm adopts bottom up traversal.
4.5.4 An overview of Apriori

Most of the algorithms for finding frequent items are based on the Apriori algorithm. To achieve an efficient frequent pattern mining, an anti-monotonic property of frequent itemsets, called the Apriori heuristic, was formulated. The basic intuition of this property is that any subset of a frequent itemset must be frequent. Apriori is a breadth first search algorithm that iteratively generates two kinds of sets: $C_k$ and $L_k$. The set $L_k$ contains the frequent k-itemsets. Meanwhile, $C_k$ is the set of candidate k-itemsets, representing a superset of $L_k$. This process continues until a null set $L_k$ is generated.

The set $L_k$ is obtained by scanning the dataset and determining the support for each candidate k-itemset in $C_k$. The set $C_k$ is generated from $L_{k-1}$ following the procedure.

$$C_k = \{c \mid \text{Join}(c, L_{k-1}) \land \text{Prune}(c, L_{k-1})\} \quad (4.14)$$

where:

$$\text{join} \left(\{i_1, i_2, \ldots, i_{k-1}, i_k\}, L_{k-1}\right) \equiv$$

$$\langle i_1, i_2, \ldots, i_{k-1} \rangle \in L_{k-1} \land \langle i_1, \ldots, i_k \rangle \in L_{k-1}, \quad (4.15)$$

$$\text{Prune}(c, L_{k-1}) \equiv$$

$$\langle \forall s[(s \subseteq c \mid |s| = k-1) \rightarrow s \in L_{k-1}]\rangle \quad (4.16)$$
The main problem about the computation of frequent itemsets is the support counting, that is, computing the number of times that an itemset appears in the dataset.

**The bottlenecks of Apriori**

In Apriori the time consuming steps are Join, create candidate and add support. In add support step the algorithm will scan all transactions to find the existence of each k pattern, and then add the support number according to scan result. Since the candidate set can be very huge for k-itemset, if the candidate set is land there are n transactions of average length m the worst time complexity is

\[ O(n.l(\binom{m}{k})) \]  \hspace{1cm} (4.17)

**Huge candidate sets:**

104 frequent 1-itemsets will generate more than 107 candidate 2-itemsets. To discover a frequent pattern of size 100, e.g., \{a1, a2, ..., a100\}, one needs to generate at least 2100 \(\approx\) 1030 candidates.

**Multiple scans of database:**

Needs n or n+1 scans, n is the length of the longest frequent pattern which is a time consuming task.

**4.5.5 A revisit of FP-growth**

The main bottleneck of the Apriori-like methods is at the candidate set generation and test. This problem was dealt with by introducing a novel, compact data structure, called frequent pattern tree, or FP-tree.
Based on this structure an FP-tree-based pattern growth method was developed. FP-growth approach for mining frequent itemsets without candidate generation was proposed by Han in [44]. Its scalable frequent patterns mining method has been proposed as an alternative to the Apriori-based approach. This algorithm creates a compact tree-structure, FP-Tree, representing frequent patterns, which moderates the multi-scan problem. This algorithm is faster than others in the literature as reported by the authors of this algorithm.

FP-growth algorithm mines all frequent itemsets without candidate’s generation. The algorithm mines the frequent itemsets by using a divide-and-conquer strategy as follows: FP-growth first compresses the database representing frequent itemset into a frequent-pattern tree, or FP-tree, which retains the itemset association information as well. The next step is to divide a compressed database into set of conditional databases (a special kind of projected database), each associated with one frequent item. Finally, mine each such database separately. Particularly, the construction of FP-tree and the mining of FP-tree are the main steps in FP-growth algorithm.

A frequent pattern tree

1. It consists of one root labeled as “root”, a set of item prefix sub-trees as the children of the root, and a frequent-item header table. To ease tree traversal, header table is built so that each item points to its occurrences in the tree via chain of node-link.
2. Each node in the item prefix sub-tree consists of three fields: item-name, count, and node-link, where item-name registers which item this node represents, count registers the number of transactions represented by the portion of the path reaching this node, and node-link links to the next node in the FP-tree carrying the same item-name, or null if there is none.

3. Each entry in the frequent-item header table consists of two fields, (1) item-name and (2) head of node-link, which points to the first node in the FP-tree carrying the item-name.

The FP-tree is constructed in the following steps:

1. Scan the transaction database DB once. Collect the set of frequent items F and their supports. Sort F in support descending order as L, the list of frequent items.
2. Create the root of an FP-tree, T, and label it as “root”. For each transaction Trans in database do the following.
   a. Select and sort the frequent items in Trans according to the order of L. Let the sorted frequent item list in Trans be [p | P], where p is the first element and P is the remaining list. Call insert_tree([p | P], T).
   b. The function insert_tree([p | P], T) is performed as follows. If T has a child N such that N.item-name = p.item-name, then increment N’s count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its node-link be linked to the nodes with the same item-name via the node-link structure. If P is nonempty, call insert_tree(P, N) recursively.
4.5.6 Theoretical evaluation

When comparison is done between Apriori and LLT algorithms following observations are made.

- Unlike Apriori LLT algorithm does not do multiple scans of database to generate candidate itemsets. A single scan of database is only done in LLT.
- Join and Prune operations, which add up the time complexity of Apriori is not needed for LLT.
- In Apriori, whenever user specified minimum support threshold changes for finding frequent itemsets, the entire process has to start from the core. However, in LLT, for any user specified minimum support, no need to turn to database again, rather just fetching the frequent itemsets from the LLTree data structure is only executed.
- Performance of Apriori is not perfect on weblogs either due to its candidate generation or due to test strategy, which needs big memory and repeated scanning of database. For online or real time applications, Apriori is not suitable. However, LLT is designed to highly support real time data flow thus suitable for weblogs and online systems.

Similarly when compared with FP growth algorithm with LLT, the observations made are:
Like LLT, FP is also a compact tree structure, but recursive construction of FP growth affects algorithm performance. LLTree construction is a simple process when compared to FP tree.

FP tree can become very large and it is expensive to generate also. However, irrespective of number of transactions, the size of LLTree is $2^n+n$, where $n$ is the maximum number of items any transactions can hold.

FP algorithm need two scans of the given database while LLT needs only one scan.

FP growth algorithm when thrown online raises a severe delay issue. User’s real feedback or any parameter change in the algorithm is not possible.

4.5.7 Performance Evaluation

Performance of LLT algorithm was compared to Apriori and FP-growth, the implementations of which were downloaded from http://www.cs.helsinki.fi/u/goethals/software, implemented in C++. LLT Algorithm was written in Java and performed on a Pentium core 2 duo 2.8GHz PC with 2GB RAM running on windows XP. Datasets are generated using synthetic data generation program [40] of the IBM Almaden Quest research group available at http://fimi.cs.helsinki.fi/data/T. T10I4D100k and T25I10D10k datasets are used. All the three algorithms were run over the same data set. Characteristics of the datasets are shown in the table 4.1.
Table 4.1 Characteristics of datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>No. of items</th>
<th>Avg. transaction length</th>
<th>No. of transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>T10I4D100k</td>
<td>1000</td>
<td>10</td>
<td>100,000</td>
</tr>
<tr>
<td>T25I10D100k</td>
<td>1000</td>
<td>25</td>
<td>9,219</td>
</tr>
</tbody>
</table>

Table 4.2 shows the running time of all the three algorithms on T10I4D100k data set with different minimum supports. All times shown include time for outputting all the frequent itemsets. The minimum supports are represented by percentage of total transactions.

Table 4.2 Execution time (S) for T10I4D100k data

<table>
<thead>
<tr>
<th>Support(%)</th>
<th>APRIORI Time(s)</th>
<th>FP growth Time(s)</th>
<th>LLT Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.9</td>
<td>0.71</td>
<td>0.428</td>
</tr>
<tr>
<td>2</td>
<td>1.2</td>
<td>2.375</td>
<td>0.874</td>
</tr>
<tr>
<td>1</td>
<td>7.5</td>
<td>5.82</td>
<td>1.4</td>
</tr>
<tr>
<td>0.5</td>
<td>19.5</td>
<td>7.484</td>
<td>3.5</td>
</tr>
<tr>
<td>0.2</td>
<td>7.484</td>
<td>2.44</td>
<td>4.8</td>
</tr>
</tbody>
</table>

Under large minimum support, FP-growth runs faster than Apriori, under small minimum supports, it is slower or nearly equal to Apriori, where as LLT algorithm is faster than both algorithms under
almost all minimum support values. On average LLT runs faster than Apriori and FP-growth. The graph in the Figure 4.6 show the performance comparison of the three algorithms for different support levels.

![Graph showing execution time for various support levels]

**Figure 4.6 Execution time for various support levels**

Table 4.3 shows the performance comparison of the three algorithms on T25I10D10k data set. Here also LLT algorithm performs faster than FP growth and Apriori on average for all support levels.
Table 4.3 Execution time (S) for T25I10D10k data

<table>
<thead>
<tr>
<th>Support (%)</th>
<th>APRIORI Time(s)</th>
<th>FP growth Time(s)</th>
<th>LLT Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.2</td>
<td>0.25</td>
<td>0.17</td>
</tr>
<tr>
<td>2</td>
<td>2.23</td>
<td>3.093</td>
<td>1.27</td>
</tr>
<tr>
<td>1</td>
<td>6.27</td>
<td>4.406</td>
<td>2.819</td>
</tr>
<tr>
<td>0.5</td>
<td>16.5</td>
<td>5.187</td>
<td>4.348</td>
</tr>
<tr>
<td>0.2</td>
<td>19.73</td>
<td>10.328</td>
<td>7.9</td>
</tr>
</tbody>
</table>

Figure 4.7 is employed for displaying performance comparison on T25I10D10k data set.

Figure 4.7. Execution time for various support levels
While computing total memory consumption for each algorithm, LLT algorithm is stable over the whole range of support values on T10I4D100k when compared to Apriori. FP growth also shows stability in memory consumption but not better than LLT. The graph in the Figure 4.8 shows the performance comparison of the three algorithms in terms of memory usage.

![Figure 4.8. Memory usage on T10I4D100k](image)

The screen shot of frequent itemsets generated by LLT algorithm for T10I4D100k data set with support value 2 is shown in the figure 4.9.
Figure 4.9. Frequent itemsets for T10I4D100k using LLT Algorithm

4.6 Discussion

A new and efficient three level tree based algorithm LLT for finding frequent itemsets is discussed in this chapter. Functioning of the algorithm along with an illustration is presented. The algorithm is compared for speed and memory efficiency to two other existing algorithms. Output of this algorithm, the frequent itemsets are the input to the next module. Next chapter deals with the generation of association rules, which take the frequent itemsets as inputs.