Project – 2: FP Growth Algorithm

NAME: VARUNSRIVATHSA VENKATESHA
SBU ID: 110457727

Table of Contents

Abstract:

Data mining plays an essential role for extracting knowledge from large databases from enterprises operational databases. Everyday organizations collect huge amount of data from several resource. Frequent item-set mining is a useful tool for discovering frequently co-occurrent items. Since its inception, a number of significant Frequent item-set mining algorithms have been developed to speed up mining performance. The FP-growth algorithm is currently one of the fastest approaches to frequent item set mining. This project is to describe a java implementation of this algorithm, which contains computing a projection of an FP-tree.

Introduction:

One of the currently fastest and most popular algorithms for frequent item set mining is the FP-growth algorithm. It is based on a prefix tree representation of the given database of transactions called an FP-tree, which can save considerable amounts of memory for storing the transactions.
The basic idea of the FP-growth algorithm can be described as a recursive elimination scheme: in a preprocessing step delete all items from the transactions that are not frequent individually, i.e., do not appear in a user-specified minimum number of transactions. Then select all transactions that contain the least frequent item and delete this item from them. Recurse to process the obtained reduced database, remembering that the item sets found in the recursion share the deleted item as a prefix. On return, remove the processed item also from the database of all transactions and start over, i.e., process the second frequent item etc. In these processing steps the prefix tree, which is enhanced by links between the branches, is exploited to quickly find the transactions containing a given item and also to remove this item from the transactions after it has been processed.
Frequent item-set mining is a useful tool for discovering frequently co-occurrent items. Existing frequent item-set mining algorithms such as Apriori and FP Growth can be resource intensive when a mined dataset is huge. Parallel algorithms were developed for reducing memory use and computational cost on each machine. Early efforts focused on speeding up the Apriori algorithm. Since the FP-Growth algorithm has been shown to run much faster than the Apriori, it is logical to parallelize the FP-Growth algorithm to enjoy even faster speedup. Recent work in parallelizing FP-Growth suffers from high communication cost, and hence constrains the percentage of computation that can be parallelized.
In simple words, this algorithm works as follows: first it compresses the input database creating an FP-tree instance to represent frequent items. After this first step it divides the compressed database into a set of conditional databases, each one associated with one frequent pattern. Finally, each such database is mined separately. Using this strategy, the FP-Growth reduces the search costs looking for short patterns recursively and then concatenating them in the long frequent patterns, offering good selectivity.
In large databases, it's not possible to hold the FP-tree in the main memory. A strategy to cope with this problem is to firstly partition the database into a set of smaller databases and then construct an FP-tree from each of these smaller databases.

Related Work:

1. Apriori Algorithm: Apriori is used to find all frequent item-sets in a given database DB. The key idea of Apriori algorithm is to make multiple passes over the database. It employs an iterative approach/breadth-first search through the search space, where k-item-sets are used to explore (k+1)-item-sets. The working of Apriori algorithm is fairly depends upon the Apriori property which states that All nonempty subsets of a frequent item-sets must be frequent. It also described the anti-monotonic property which says if the system cannot pass the minimum support test, all its supersets will fail to pass the test. Therefore, if the one set is infrequent then all its supersets are also infrequent and vice versa. This property is used to prune the infrequent candidate elements. In the beginning, the set of frequent 1-itemset is found. The set of that contains one item, which satisfy the support threshold, is denoted by L. In each subsequent pass, we begin with a seed set of item-sets found to be large in the previous pass. This seed set is used for generating new potentially large item-sets, called candidate item-sets, and count the actual support for these candidate item-sets during the pass over the data. At the end of the pass, we determine which of the candidate item-sets are actually large/frequent, and they become the seed for the next pass. Therefore, L is used to find L!, the set of frequent 2-itemsets, which is used to find L , and so on, until no more frequent k-item-sets can be found.

2. Eclat Algorithm: An advantage of Eclat algorithm is that it scans the database only two times. A first time to build the 2-itemsets and a second time to transform it into a vertical form. Eclat algorithm has 3 steps:
   i) The initialization phase: construction of the global counts for the frequent 2-itemsets
   ii) The transformation phase: partitioning of the frequent 2-itemsets and scheduling of partitions over the processors. Vertical transformation of the database.
   iii) The Asynchronous phase: construction of the frequent k-item-sets.

   This algorithm uses an equivalence class partitioning schema of the database. The equivalence class is based on common prefix assuming that item-sets are lexicographically sorted. Then candidate item-sets can be generated by joining the members of the same equivalence class. The transformation phase is known to be the most expensive step of the algorithm. In fact, the processors have to broadcast to all other processors the local list corresponding to transaction identifier, for the item-sets.

3. Recursive Elimination Algorithm: The Recursive Elimination algorithm employs a basically horizontal transaction representation, but separates the transactions according to their leading item, thus introducing a vertical representation aspect. The transaction database to mine is preprocessed in 5 steps,
   i) The transaction database is shown in its original form
   ii) The frequencies of individual items are determined from this input in order to be able to discard infrequent items immediately
   iii) The items in each transaction are sorted according to their frequency in the transaction database, since it is well known that processing the items in the order of increasing frequency usually leads to the shortest execution times
   iv) The transactions are sorted lexicographically into descending order, with item comparisons again being decided by the item frequencies, although here the item with the higher frequency precedes the item with the lower frequency.
   v) All transactions are grouped according to their leading item. In addition, the transactions are organized as lists, even though, in principle, using arrays would also be possible. These lists are sorted in descending order w.r.t. the frequency of their associated items in the transaction database.

4. Split and Merge Algorithm: The Split and Merge algorithm can be seen as a simplification of the already fairly simple Recursive Elimination algorithm. While Recursive Elimination represents a conditional database by storing one transaction list for each item which is partially vertical representation, the split and merge algorithm employs only a single transaction list purely horizontal representation, stored as an array. This array is processed with a simple split and merge scheme, which computes a conditional database, processes this conditional database recursively, and finally eliminates the split item from the original conditional database. Split and Merge Algorithm preprocesses a given transaction database in a way that is very similar to the Recursive Elimination algorithm. Except that in the step 5, the data structure on which this algorithm operates is built by combining equal transactions and setting up an array, in which each element consists of two fields: an occurrence counter and a pointer to the sorted transaction. This data structure is then processed recursively to find the frequent item sets.

Proposed Solution:

1. Pseudo Code

```
Parse the input files.
Store all the items in map<Integer, List<String>> dataSet.
Compute frequency of all the elements and store it is map<String, Integer> frequency.
Take minSup condition as input.
If (frequency of item < minSup) {
  Remove that item from dataSet.
}
Sort the items in the dataSet based on the frequency as priority.
Create a root and add child by accessing the all the lists from the map.
After forming FP Tree create conditional FP tree and Pattern base considering minSup.
Output frequent patterns of FP Tree.
```

2. Creating FP Tree and Conditional FP Tree

Step 1:

i. Scan the data and find frequency of each item.
ii. Discard infrequent items.
iii. Sort frequent items in decreasing order based on their frequency.
iv. Construct FP tree by adding the elements from all the lists.

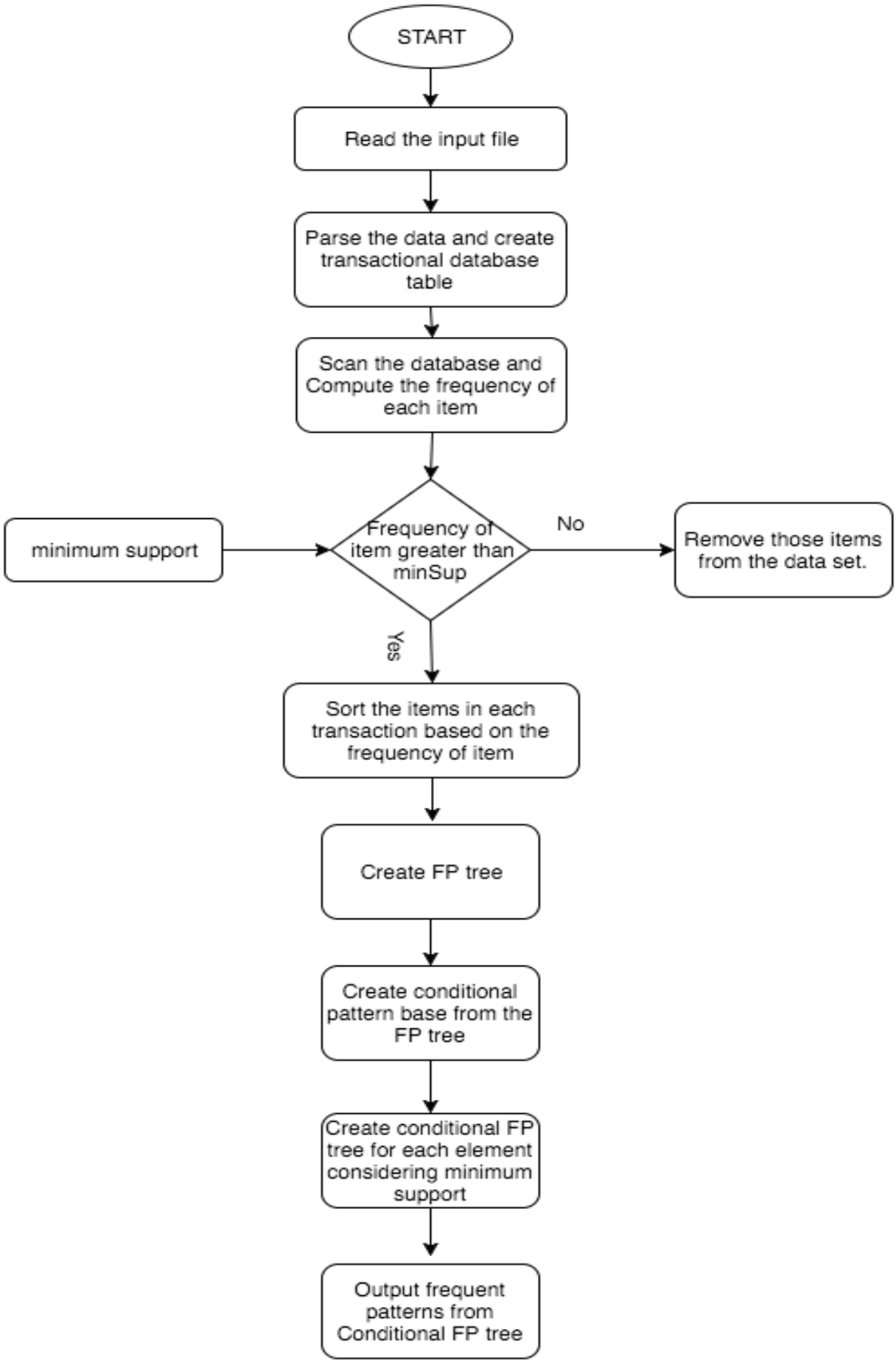
Step 2:

i. FP growth algorithm extracts frequent item sets from the FP tree.
ii. This is a bottom-up algorithm hence we move from leaves to roots.
iii. First extract prefix path subtree ending in an item.
iv. Each prefix path subtree is processed recursively to extract the frequent item sets. Finally, solutions are merged.

Conditional FP Tree:

i. This is the FP Tree that would be built if we only consider transactions containing a particular item set
ii. Update the support counts along the prefix paths to reflect the number of tractions containing that particular item.
iii. Remove the nodes containing information about the selected node.
iv. Remove infrequent items from the prefix paths.

3. Flow Chart:


START
Read the input file
Parse the data and create transactional database table
Scan the database and Compute the frequency of each item
minimum support
Frequency of item greater than minSup
No
Remove those items from the data set.
Yes
Sort the items in each transaction based on the frequency of item
Create FP tree
Create conditional pattern base from the FP tree
Create conditional FP tree for each element considering minimum support
Output frequent patterns from Conditional FP tree

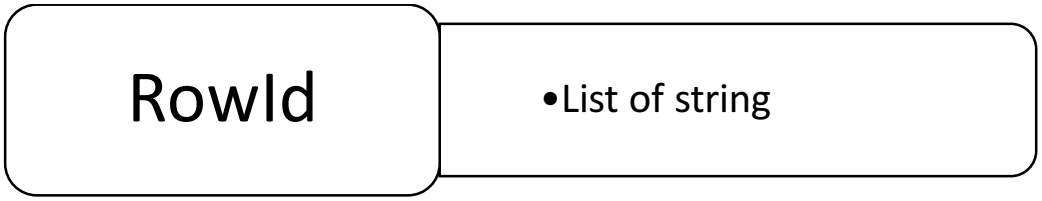
Implementation Details:

1. FPTree.java

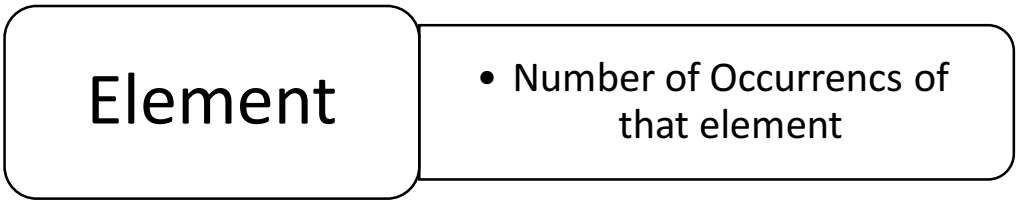
    A. Class FPTree: Class FPTree has following attributes.
        a. value: This is to store the value of every object of the tree.
        b. count: count is an integer to have track of how many time that node is repeated.
        c. hasSibling: This is a boolean value to identify if the tree node has sibling.
        d. isLeaf: This is a boolean value to identify if the tree node is a leaf.
        e. List<FPTree> parentList: This is a list of type FPTree nodes to store parents of each node.
        f. List<FPTree> chidList: This is a list of type FPTree nodes to store children of each node.

2. Main.java

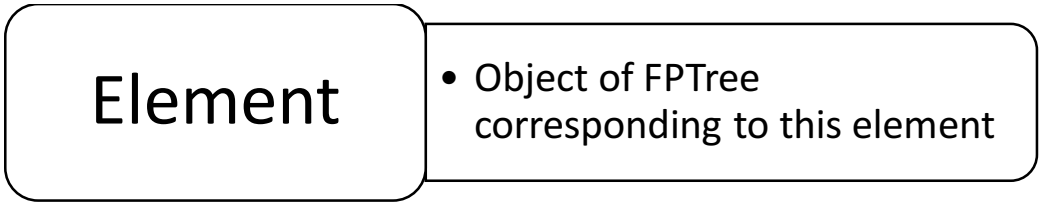
    A. Class Main: Class Main has following attributes and functions.
        a. HashMap<Integer, List<String>> dataset: This is a map to store the dataset with key as row Id and List of string.



        b. HashMap<String, Integer> frequency: This map is to store the number of occurrences of every element in the data set. Here the key is the element and the value is the number of times the element is repeated.



        c. HashMap<String, FPTree> elementObjectMap: This map is to have track of all the tree nodes with its corresponding values.



        d. Functions Present in this class are
            i. void rdFile() : this function read the dataset and stores every row in the dataset map with key as rowId.
            ii. void calcFrequency(): this function read the dataset and stores number of occurrences of each element in frequency map.
            iii. void minSup(int percent): this function takes in the input value of percentage minimum support and calculate the actual minimum support for that dataset.

iv. void removeElements(): this function remove all the elements in the data set with frequency less than minSup condition.
v. void prioritizeList(): A function to arrange all the elements in each row according to decreasing order of frequency.
vi. void createFPTree(): this function creates the root and all its children by accessing every list from dataset. Also add the nodes created to element to object map.
vii. void conditionalFPTree(): this function traverses the dataset from least frequent node and we track all the node pointers of each, extract all suffix tree or parent pointers and check if they satisfy minimum support condition, if they do, then make it to be a part of the conditional FP tree else prune it.

Execution Traces:

- Data Set

```
1: 1.0 2.8 1.04 0.28 2.43 1.71 2.29 1065.0 14.23 15.6 127.0 3.06 5.64 3.92
2: 1.0 2.65 0.26 3.4 13.2 2.14 100.0 2.76 1.28 1.05 1.78 11.2 4.38 1050.0
3: 1.0 2.8 2.36 101.0 0.3 3.17 18.6 13.16 2.67 2.81 1.03 3.24 5.68 1185.0
4: 1.0 1.95 2.5 16.8 0.24 14.37 113.0 3.85 3.49 2.18 7.8 0.86 3.45 1480.0
5: 1.0 2.87 2.8 13.24 2.59 1.04 21.0 118.0 2.69 0.39 1.82 4.32 2.93 735.0
6: 1.0 2.45 0.34 1.97 1.76 15.2 112.0 14.2 3.27 3.39 6.75 2.85 1.05 1450.0
7: 1.0 2.45 1.87 96.0 14.39 14.6 2.5 2.52 0.3 1.98 3.58 1.02 5.25 1290.0
8: 1.0 1.25 2.61 2.6 2.51 14.06 2.15 17.6 121.0 0.31 5.05 1.06 3.58 1295.0
9: 1.0 1.64 14.83 2.17 14.0 97.0 0.29 2.8 2.98 2.85 1.98 1045.0 5.2 1.08
10: 1.0 16.0 1.35 98.0 13.86 2.27 2.98 0.22 3.15 1.85 7.22 1.01 3.55 1045.0
11: 1.0 2.95 1.25 18.0 14.1 2.16 2.3 105.0 3.32 0.22 2.38 3.17 5.75 1510.0
12: 1.0 1.48 14.12 2.32 0.26 16.8 95.0 2.2 2.43 1.57 2.82 1280.0 5.0 1.17
13: 1.0 2.41 1.73 16.0 13.75 89.0 2.6 0.29 1.81 5.6 1.15 2.9 1320.0 2.76
14: 1.0 14.75 1.73 3.1 2.39 11.4 91.0 3.69 0.43 1.25 2.73 1150.0 2.81 5.4
15: 1.0 14.38 3.0 0.29 1.87 2.38 102.0 3.3 2.96 7.5 1.2 1547.0 12.0 3.64
16: 1.0 13.63 2.7 1.81 112.0 2.85 17.2 0.3 1.46 2.88 2.91 7.3 1.28 1310.0
17: 1.0 20.0 1.92 14.3 2.72 120.0 1.97 2.8 2.65 6.2 1.07 3.14 0.33 1280.0
18: 1.0 20.0 13.83 115.0 2.95 3.4 1.57 2.62 1.72 1.13 0.4 6.6 2.57 1130.0
19: 1.0 2.48 14.19 1.59 16.5 108.0 1.86 3.3 3.93 0.32 1.23 8.7 2.82 1680.0
20: 1.0 13.64 3.1 2.56 2.7 15.2 1.66 116.0 5.1 0.17 845.0 3.03 0.96 3.36
21: 1.0 16.0 3.0 126.0 14.06 2.28 1.63 1.09 2.1 3.17 0.24 5.65 3.71 780.0
22: 1.0 3.8 12.93 2.41 2.41 0.25 2.65 102.0 1.98 18.6 4.5 1.03 3.52 770.0
23: 1.0 1035.0 13.71 3.8 1.11 4.0 2.61 1.86 2.36 101.0 0.27 2.88 16.6 1.69
24: 1.0 12.85 1.6 2.48 2.37 1.09 0.26 2.52 95.0 3.93 17.8 1.46 3.63 1015.0
25: 1.0 13.5 20.0 2.61 2.61 96.0 1.81 2.53 1.66 0.28 845.0 3.52 1.12 3.82
26: 1.0 13.05 2.68 3.22 2.63 2.05 25.0 124.0 0.47 1.92 3.58 1.13 3.2 830.0
27: 1.0 1.77 13.39 3.22 1195.0 0.34 2.85 1.45 4.8 0.92 2.94 2.62 16.1 93.0
28: 1.0 13.3 1.35 3.95 0.27 1.72 2.14 17.0 2.4 2.19 1.02 1285.0 94.0 2.77
29: 1.0 13.87 1.9 2.95 1.25 2.97 0.37 2.8 107.0 1.76 19.4 4.5 3.4 915.0
30: 1.0 14.02 1.68 2.21 1035.0 16.0 96.0 2.65 0.26 1.04 1.98 2.33 4.7 3.59
31: 1.0 1.5 13.73 3.0 0.29 2.7 3.25 101.0 2.38 22.5 5.7 1.19 2.71 1285.0
32: 1.0 13.58 1.66 1.09 2.36 0.22 1.95 19.1 106.0 2.86 3.19 6.9 2.88 1515.0
33: 1.0 13.68 1.83 1.97 3.84 2.87 2.36 17.2 2.69 1.23 990.0 104.0 2.42 0.42
34: 1.0 13.76 1.53 2.95 2.74 0.5 1.25 3.0 1.35 2.7 1235.0 19.5 132.0 5.4
35: 1.0 13.51 1.8 0.29 2.65 2.53 1.54 4.2 1.1 2.87 1095.0 110.0 19.0 2.35
36: 1.0 13.48 2.41 20.5 2.7 1.81 2.98 0.26 1.86 5.1 1.04 3.47 920.0 100.0
37: 1.0 13.28 2.68 1.64 15.5 2.6 2.84 0.34 1.36 4.6 1.09 110.0 2.78 880.0
38: 1.0 13.05 1.65 2.55 18.0 98.0 2.51 2.45 0.29 1.44 4.25 1105.0 2.43 1.12
39: 1.0 13.07 1.5 98.0 15.5 2.1 0.28 2.69 1020.0 2.4 2.64 1.37 3.7 1.18
40: 1.0 14.22 3.99 3.0 2.51 3.04 0.2 2.08 5.1 0.89 3.53 760.0 13.2 128.0
41: 1.0 13.56 0.34 2.34 6.13 0.95 3.38 795.0 3.15 3.29 1.71 2.31 16.2 117.0
42: 1.0 13.41 2.68 1035.0 1.48 3.0 4.28 0.91 2.45 3.84 2.12 18.8 90.0 0.27
43: 1.0 13.88 1.89 3.25 3.56 3.56 1095.0 2.59 101.0 15.0 0.17 0.88 1.7 5.43
44: 1.0 2.63 3.0 680.0 1.66 13.24 4.36 0.82 3.98 0.32 2.29 2.64 17.5 103.0
```

- Frequency of items and Data set after pruning using minSup condition

```
1.0: 47
2.44: 45
18.9: 45
4.04: 45
14.21: 45
111.0: 45
1.89: 42
13.88: 42
13.41: 41
13.56: 40
14.22: 39
3.99: 39
13.07: 38
13.28: 36
13.48: 35
1.8: 34
13.51: 34
13.76: 33
1.53: 33
13.68: 32
1.83: 32
1.5: 31
13.58: 31
```

```
1: 1.0 2.8 1.04 0.28
2: 1.0 2.65 0.26
3: 1.0 2.8 2.36 101.0 0.3 3.17
4: 1.0 1.95 2.5 16.8 0.24
5: 1.0 2.87 2.8 13.24 2.59 1.04 21.0 118.0 2.69
6: 1.0 2.45 0.34 1.97 1.76 15.2 112.0 14.2 3.27 3.39 6.75 2.85
7: 1.0 2.45 1.87 96.0 14.39 14.6 2.5 2.52 0.3 1.98
8: 1.0 1.25 2.61 2.6 2.51 14.06 2.15 17.6 121.0 0.31 5.05 1.06
9: 1.0 1.64 14.83 2.17 14.0 97.0 0.29 2.8 2.98 2.85 1.98
10: 1.0 16.0 1.35 98.0 13.86 2.27 2.98 0.22 3.15
11: 1.0 2.95 1.25 18.0 14.1 2.16 2.3 105.0 3.32 0.22 2.38 3.17
12: 1.0 1.48 14.12 2.32 0.26 16.8 95.0
13: 1.0 2.41 1.73 16.0 13.75 89.0 2.6 0.29 1.81
14: 1.0 14.75 1.73 3.1 2.39 11.4 91.0 3.69 0.43 1.25 2.73 1150.0
15: 1.0 14.38 3.0 0.29 1.87 2.38 102.0 3.3 2.96 7.5 1.2 1547.0
16: 1.0 13.63 2.7 1.81 112.0 2.85 17.2 0.3
17: 1.0 20.0 1.92 14.3 2.72 120.0 1.97 2.8 2.65 6.2 1.07 3.14 0.33
18: 1.0 20.0 13.83 115.0 2.95
19: 1.0 2.48 14.19 1.59 16.5 108.0 1.86 3.3 3.93 0.32 1.23
20: 1.0 13.64 3.1 2.56 2.7 15.2 1.66 116.0 5.1 0.17 845.0
21: 1.0 16.0 3.0 126.0 14.06 2.28 1.63 1.09 2.1 3.17 0.24
22: 1.0 3.8 12.93 2.41 2.41 0.25 2.65 102.0 1.98
23: 1.0 1035.0 13.71 3.8 1.11 4.0 2.61 1.86 2.36 101.0 0.27
24: 1.0 12.85 1.6 2.48 2.37 1.09 0.26 2.52 95.0 3.93
25: 1.0 13.5 20.0 2.61 2.61 96.0 1.81 2.53 1.66 0.28 845.0
26: 1.0 13.05 2.68 3.22 2.63 2.05 25.0 124.0 0.47 1.92
27: 1.0 1.77 13.39 3.22 1195.0 0.34 2.85 1.45 4.8 0.92
28: 1.0 13.3 1.35 3.95 0.27
29: 1.0 13.87 1.9 2.95 1.25 2.97 0.37 2.8 107.0 1.76 19.4
30: 1.0 14.02 1.68 2.21 1035.0 16.0 96.0 2.65 0.26 1.04 1.98
31: 1.0 1.5 13.73 3.0 0.29 2.7 3.25 101.0 2.38 22.5 5.7 1.19 2.71
32: 1.0 13.58 1.66 1.09 2.36 0.22 1.95
33: 1.0 13.68 1.83 1.97 3.84 2.87 2.36 17.2 2.69 1.23 990.0
34: 1.0 13.76 1.53 2.95 2.74 0.5 1.25 3.0 1.35 2.7 1235.0 19.5 132.0
35: 1.0 13.51 1.8 0.29 2.65 2.53 1.54 4.2 1.1 2.87 1095.0 110.0 19.0 2.35
36: 1.0 13.48 2.41 20.5 2.7 1.81 2.98 0.26 1.86 5.1 1.04
37: 1.0 13.28 2.68 1.64 15.5 2.6 2.84 0.34 1.36 4.6 1.09 110.0 2.78 880.0
38: 1.0 13.05 1.65 2.55 18.0 98.0 2.51 2.45 0.29 1.44 4.25 1105.0
39: 1.0 13.07 1.5 98.0 15.5 2.1 0.28 2.69
40: 1.0 14.22 3.99 3.0 2.51 3.04 0.2 2.08 5.1 0.89 3.53 760.0
41: 1.0 13.56 0.34 2.34 6.13 0.95 3.38 795.0 3.15
42: 1.0 13.41 2.68 1035.0 1.48 3.0 4.28 0.91 2.45 3.84 2.12 18.8 90.0 0.27
43: 1.0 13.88 1.89 3.25 3.56 3.56 1095.0 2.59 101.0 15.0 0.17 0.88
44: 1.0 2.63 3.0 680.0 1.66 13.24 4.36 0.82 3.98 0.32
45: 1.0 13.05 1.77 3.0 3.0 107.0 2.1 0.28 0.88
46: 1.0 14.21 4.04 2.44 18.9 111.0 1.25 5.24 0.87 3.33 1080.0 2.65 2.85 0.3
```

- FP Tree Creation

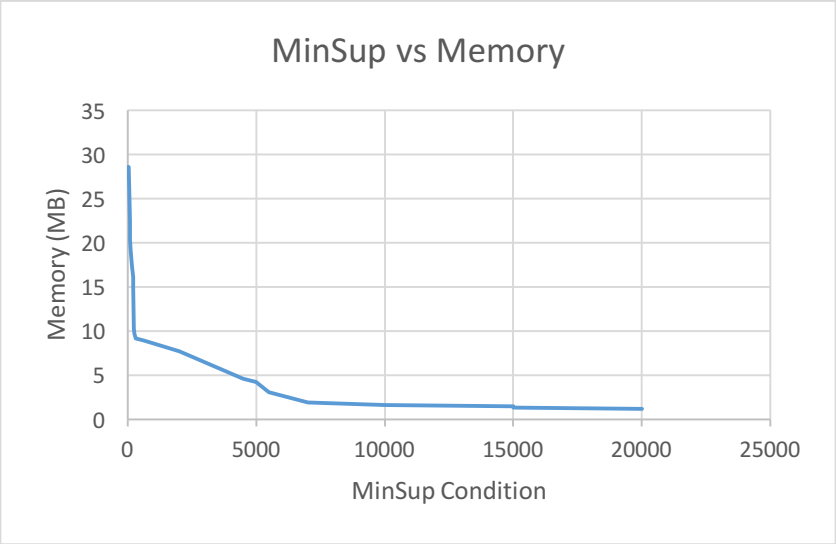
```
Child added With Value 1.0
Parent of This Child 1.0 is null,
Child added With Value 2.8
Parent of This Child 2.8 is null, 1.0,
Child added With Value 1.04
Parent of This Child 1.04 is null, 1.0, 2.8,
Child added With Value 0.28
Parent of This Child 0.28 is null, 1.0, 2.8, 1.04,
Child added With Value 2.43
Parent of This Child 2.43 is null, 1.0, 2.8, 1.04, 0.28,
Child added With Value 1.71
Parent of This Child 1.71 is null, 1.0, 2.8, 1.04, 0.28, 2.43,
Child added With Value 2.29
Parent of This Child 2.29 is null, 1.0, 2.8, 1.04, 0.28, 2.43, 1.71,
Child added With Value 1065.0
Parent of This Child 1065.0 is null, 1.0, 2.8, 1.04, 0.28, 2.43, 1.71, 2.29,
Child added With Value 14.23
Parent of This Child 14.23 is null, 1.0, 2.8, 1.04, 0.28, 2.43, 1.71, 2.29, 1065.0,
Child added With Value 15.6
Parent of This Child 15.6 is null, 1.0, 2.8, 1.04, 0.28, 2.43, 1.71, 2.29, 1065.0, 14.23,
Child added With Value 127.0
Parent of This Child 127.0 is null, 1.0, 2.8, 1.04, 0.28, 2.43, 1.71, 2.29, 1065.0, 14.23, 15.6,
Child added With Value 3.06
Parent of This Child 3.06 is null, 1.0, 2.8, 1.04, 0.28, 2.43, 1.71, 2.29, 1065.0, 14.23, 15.6, 127.0,
Child added With Value 5.64
Parent of This Child 5.64 is null, 1.0, 2.8, 1.04, 0.28, 2.43, 1.71, 2.29, 1065.0, 14.23, 15.6, 127.0, 3.06,
Child added With Value 3.92
Parent of This Child 3.92 is null, 1.0, 2.8, 1.04, 0.28, 2.43, 1.71, 2.29, 1065.0, 14.23, 15.6, 127.0, 3.06, 5.64,
-----------------------------------------------
<----------Child Count Increased to 2 ---------->
Child added With Value 2.65
Parent of This Child 2.65 is null, 1.0,
Child added With Value 0.26
Parent of This Child 0.26 is null, 1.0, 2.65,
Child added With Value 3.4
Parent of This Child 3.4 is null, 1.0, 2.65, 0.26,
Child added With Value 13.2
Parent of This Child 13.2 is null, 1.0, 2.65, 0.26, 3.4,
Child added With Value 2.14
Parent of This Child 2.14 is null, 1.0, 2.65, 0.26, 3.4, 13.2,
Child added With Value 100.0
Parent of This Child 100.0 is null, 1.0, 2.65, 0.26, 3.4, 13.2, 2.14,
Child added With Value 2.76
Parent of This Child 2.76 is null, 1.0, 2.65, 0.26, 3.4, 13.2, 2.14, 100.0,
```

Result:

Data Set	minSup Condition	Number of Dimensions	Number of Tuples	Time (S)	Memory(MB)
	7000			21.3	33.21
	15000			23.7	31.56
Letter Recognition	20000	16	20000	20.9	29.45
	5000			40.2	20.34
	10000			38.8	19.12
Poker	15000	11	25010	35.6	16.07
	10			10.3	3.54
	50			7.7	2.92
Fertility	90	10	100	6.5	1.63
	125			6.11	4.12
	250			4.3	2.18
Wines	300	14	345	2.6	1.13
	100			6.6	5.15
	200			3.3	2.89
Dermatology	350	34	366	2.2	1.87
	100			2.5	5.67
	225			1.4	2.66
Ionosphere	300	35	351	1.9	1.45
	200			3.1	7.23
	400			1.7	3.88
Meta Data	600	22	528	1.3	2.06
	2000			3.5	21.56
	4500			3.6	17.41
Robot Movements	5500	3	5456	3.2	14.95
	150			6	3.86
	300			4.1	2.57
Housing	450	14	506	2	1.34

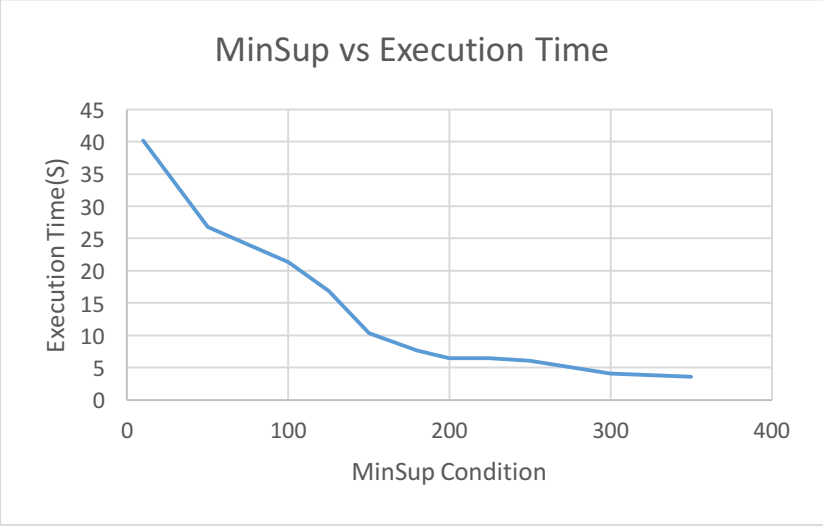
Analysis of the Result:

A. MinSup versus Memory



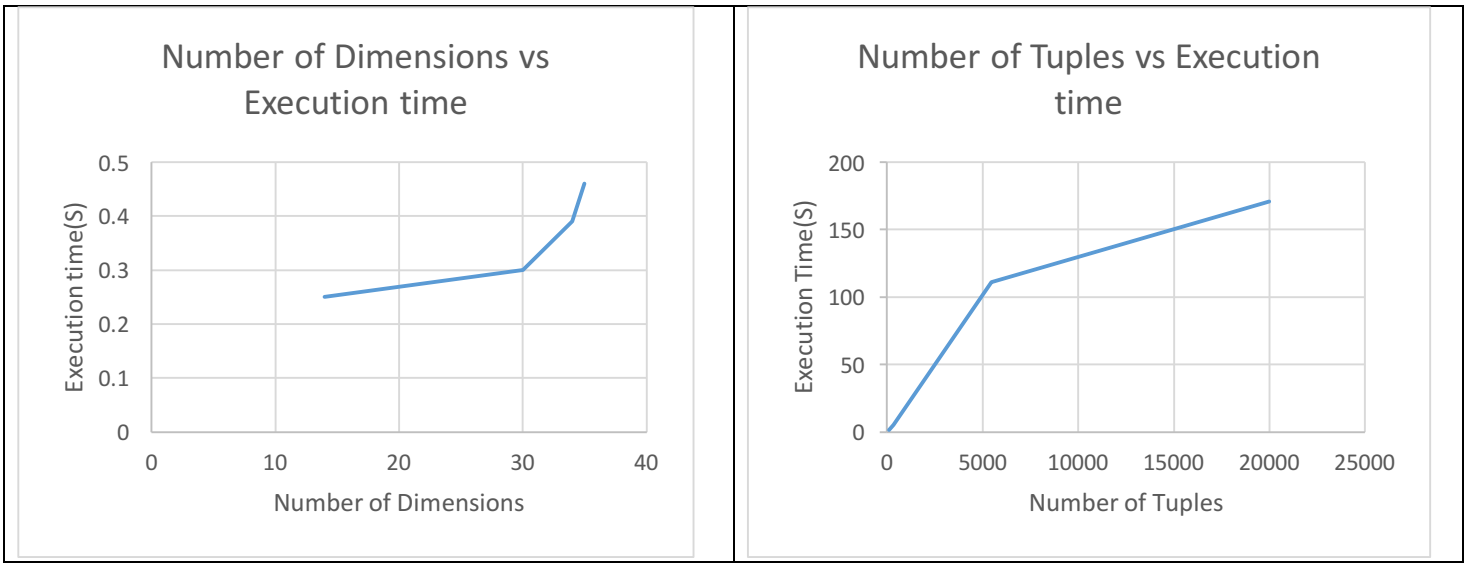
- From the chart we can observe that Memory required for data sets reduces with increase in Minimum Support condition.
- This is because as Minimum Support condition increases amount of frequent patterns to be computed reduces i.e. values less than Minimum Support condition are not processed hence memory required for data-set reduces.

B. MinSup versus Execution time



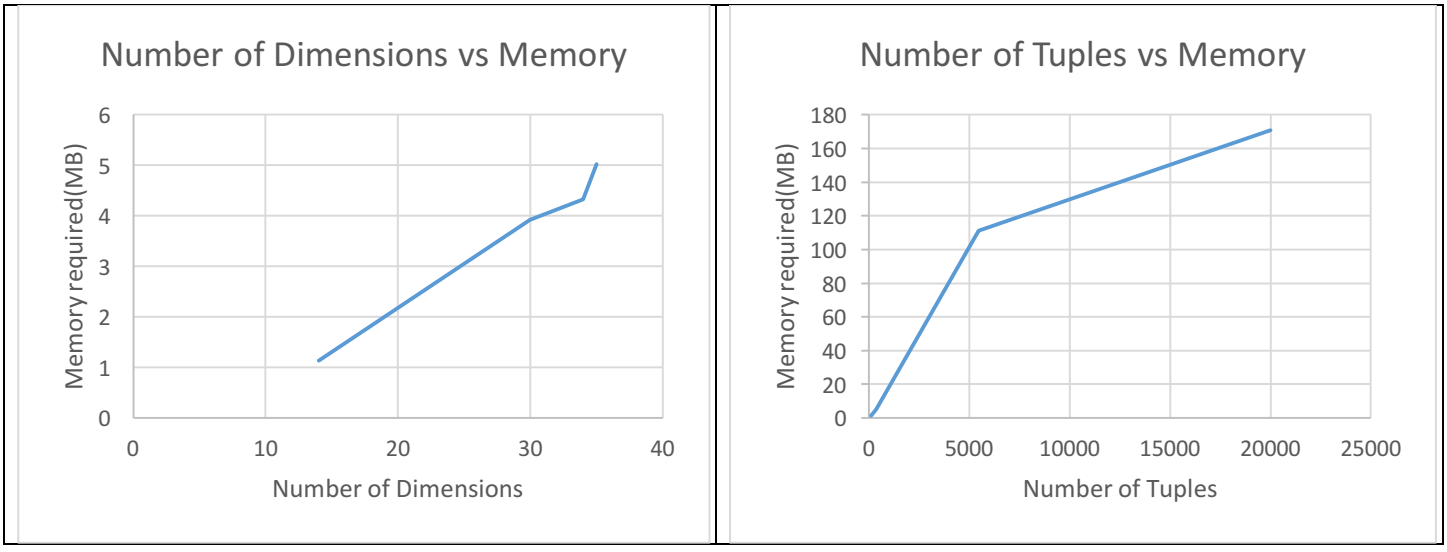
- From the chart we can observe that Execution time reduces with increase in MinSup condition.
- This is because as MinSup condition increases number of frequent patterns to be computed reduces i.e. values less than MinSup condition are not processed hence time required to compute frequent patterns is less with results in low execution time.

A. Variation of Execution time with Number of dimensions and Number of tuples.



- From the chart we can observe that as the number of dimension increases with constant number of tuples in each data-set, Execution increases as Number of dimensions' increases.
- This is observed because as the number of dimension increases amount of data to be processed increases hence the execution time increases.
- Similarly, execution time increases with increase in Number of tuples as the data to be processes increases.

B. Variation of Memory with Number of dimensions and Number of tuples.



- From the chart we can observe that as the number of dimension increases with constant number of tuples in each data-set, Memory required for data sets increases with increase in Number of dimensions.
- This is observed because as the number of dimension increases amount of data to be processed increases hence memory required for computation increases.

- Similarly, Memory required for computing data increases with increase in Number of tuples as the data to be processes increases.

Conclusion:

Hence from all the observation we noticed that Minimum Support condition is the key for the Execution time and memory required for processing the data. Also the size of that is directly proportional the execution time and memory required for processing data as large data needs more amount of data to processed.

Q. Discuss how the algorithm must be modified to run on a grid network of embedded sensors as the one discussed in the paper.

A. In a grid based network, the embedding cells are divided into cells independent of the distribution of input objects. The object cells are divided into a finite number of cells that form a grid structure on which all the operations are performed. We know that the timestamp of each sensor reading is locally dumped in the microcontrollers cache.
FP growth can be applied as follows: The FP growth algorithm can be applied to the sensor data to get to know about the most frequent sensor data in each node. All frequent data in each node is aggregated later on and FP growth algorithm can be applied to the aggregated values to get the most useful sensor node from the microcontroller.

Result:

FP Growth Algorithm is successfully implemented and Analysis of execution time and memory required for processing the data is done for various Minimum Support condition values.

Bibliography:

- http://www.borgelt.net/papers/fpgrowth.pdf
- http://www.philippe-fournier-viger.com/spmf/index.php?link=algorithms.php
- http://arxiv.org/pdf/1403.3948.pdf
- http://www.ijcsonline.com/IJCS/IJCS_2014_0103002.pdf
- https://pdfs.semanticscholar.org/cb3e/76d1773d08545f21daf28cc87b051604aa95.pdf
- http://www.borgelt.net/papers/ifsa_09.pdf
- Lecture notes by Prof. Alexa Doboli.

Apendix:

- FPTreee. Java

```
class FPTree{
	Double value;
	int count;
	boolean hasSibling;
	boolean isLeaf;
	LinkedList<FPTree> parent = new LinkedList<FPTree>();
	LinkedList<FPTree> children = new LinkedList<FPTree>();

	public FPTree(Double value){
		this.value = value;
		this.count = 1;
	}
}
```

- Main.java

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Comparator;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.Map;
import java.util.Scanner;
import java.util.TreeMap;

class Main{

	HashMap<Integer, LinkedList<Double>> wine = new HashMap<Integer,
LinkedList<Double>>();
	HashMap<Double, Integer> element = new HashMap<Double, Integer>();
	TreeMap<Double, Integer> sortedMap = new TreeMap<Double, Integer>();
	FPTree root = new FPTree(null);
	HashMap<Double, FPTree> objMap = new HashMap<Double, FPTree>();
	int minSup;
	private static Scanner sc;

	public void rdFile(){
		String fileName =
"/Users/Varun/Documents/workspace/FPgrowth/wine.data";
	   String line = null;
```

```
    try {
        FileReader fileReader = new FileReader(fileName);

        BufferedReader bufferedReader = new BufferedReader(fileReader);

        int index = 0;

        while((line = bufferedReader.readLine()) != null) {
            index++;
            String[] arr = line.split(",");
            LinkedList<Double> list = new LinkedList<Double>();

            for(int i=0; i<arr.length; i++){
                list.add(Double.valueOf(arr[i]));
            }

            wine.put(index, list);
        }
        bufferedReader.close();
    }
    catch(FileNotFoundException ex) {
        System.out.println(
            "Unable to open file '" +
            fileName + "'");
    }
    catch(IOException ex) {
        System.out.println(
            "Error reading file '"
            + fileName + "'");
    }
}

public void calculateFrequency(){
    int count = 1;
    LinkedList<Double> temp = new LinkedList<Double>();


    for(Map.Entry<Integer, LinkedList<Double>> i : wine.entrySet()){
        temp = i.getValue();

        for(int j=0; j<temp.size(); j++){
            if(!element.containsKey(temp.get(j))){
                element.put(temp.get(j), count);
            }
            else{
                count = element.get(temp.get(j));
                element.put(temp.get(j),count+1);
            }
        }
```

```
            }
      }

      public void minSup(int percentage){
            int cnt=0;
            for(int i=0; i<wine.size(); i++){
                  cnt++;
            }
            minSup = (cnt*percentage)/100;
//          System.out.println("-----------------------------------------------");
//          System.out.println("MinSup Condition is" + minSup);
//          System.out.println("-----------------------------------------------");
      }

      public void removeElements(){
            for(Map.Entry<Integer, LinkedList<Double>> i : wine.entrySet()){
                  LinkedList<Double> temp = i.getValue();
                  LinkedList<Double> newtemp = new LinkedList<Double>();
                  int keyWine = i.getKey();

                  for(Double j: temp){
                        if(element.get(j) >= minSup){
                              newtemp.add(j);
                        }
                  }

                  wine.put(keyWine, newtemp);
            }
      }


      public void sortPriority(){
            sortedMap = sortMapByValue(element);

//          System.out.println("--------------------Sorted----------------");
//
//          for(Map.Entry<Double, Integer> i : sortedMap.entrySet()){
//                System.out.print(i.getKey() + ": ");
//                System.out.println(i.getValue());
//                System.out.println();
//          }
      }

      public static TreeMap<Double, Integer> sortMapByValue(HashMap<Double,
Integer> map){
            Comparator<Double> comparator = new ValueComparator(map);
            TreeMap<Double, Integer> result = new TreeMap<Double,
Integer>(comparator);
            result.putAll(map);
```

```
            return result;
      }

      public void prioritizeList(){
            for(Map.Entry<Integer, LinkedList<Double>> i : wine.entrySet()){
                  LinkedList<Double> temp = new LinkedList<Double>();
                  int key = i.getKey();
                  temp = i.getValue();

                  for(int j=0; j<temp.size(); j++){
                        for(int k = 1; k < (temp.size() - j); k++){
                              if(element.get(temp.get(k-1)) <
element.get(temp.get(k))){
                                    double var = temp.get(k-1);
                                    temp.set(k-1, temp.get(k));
                                    temp.set(k, var);
                              }
                        }
                  }

                  wine.put(key, temp);
            }
      }

      public void getRow(){
            for(Map.Entry<Integer, LinkedList<Double>> i : wine.entrySet()){
                  createFPTree(i.getValue());
            }
      }


      public void createFPTree(LinkedList<Double> list){
            FPTree currentNode = root;

            //System.out.println("-----------------------------------------
");
            for(int i=0; i<list.size(); i++){
                  FPTree childExists = checkChild(currentNode, list.get(i));
                  if(childExists == null){
                        FPTree newNode = new FPTree(list.get(i));
                        objMap.put(list.get(i), newNode);
                        if(i == list.size()-1){
                              newNode.isLeaf = true;
                        }
                        currentNode.children.add(newNode);
                        newNode.parent.addAll(currentNode.parent);
                        newNode.parent.add(currentNode);
                        if(currentNode.children.size() >1){
                              currentNode.hasSibling = true;
```

```
                        }
                        currentNode = newNode;
                        System.out.println("Child added With Value " +
newNode.value);
                        System.out.print("Parent of This Child " + newNode.value +
" is ");
                        for(int j=0; j<newNode.parent.size(); j++){
                              System.out.print(newNode.parent.get(j).value + ", ");
                        }
                        System.out.println();
                  }
                  else{
                        currentNode = childExists;
                        currentNode.count = currentNode.count+1;
                        System.out.println("<----------Child Count Increased to "+
currentNode.count+" ---------->");
                  }
            }
            System.out.println("-----------------------------------------------");
      }

      public FPTree checkChild(FPTree root, Double child){

            LinkedList<FPTree> children = root.children;
            FPTree temp;

            if(!children.isEmpty()){
                  for(int i=0; i<children.size(); i++){
                        temp = children.get(i);
                        if(temp.value.equals(child)){
                              return temp;
                        }
                  }
            }

            return null;
      }

      public static void main(String[] args){

            Main m = new Main();
            System.out.println("Enter the percentage of tuples to be considered");
            sc = new Scanner(System.in);
            int percent = sc.nextInt();
            m.rdFile();
            m.calculateFrequency();
            m.sortPriority();
            m.prioritizeList();
            m.getRow();
```

```
            m.minSup(percent);
            m.removeElements();

            for(Map.Entry<Integer, LinkedList<Double>> i : m.wine.entrySet()){
                  System.out.print(i.getKey() + ": ");
                  LinkedList<Double> t = i.getValue();
                  for(int j=0; j<t.size(); j++){
                        System.out.print(t.get(j)+ " ");
                  }
                  System.out.println();
            }
            System.out.println("-----------------------------------------------");

            for(Map.Entry<Double, FPTree> i : m.objMap.entrySet()){
                  System.out.print(i.getKey() + "-->");
                  LinkedList<FPTree> lst = i.getValue().parent;
                  for(FPTree n : lst){
                        System.out.print(n.value + " ");
                  }
                  System.out.println();
            }
            System.out.println("-----------------------------------------------");

            //System.out.println("--------------------Not Sorted----------------
");

            for(Map.Entry<Double, Integer> i : m.element.entrySet()){
                  System.out.print(i.getKey() + ": ");
                  System.out.println(i.getValue());
                  System.out.println();
            }
      }

}
```