Finding Frequent Pattern

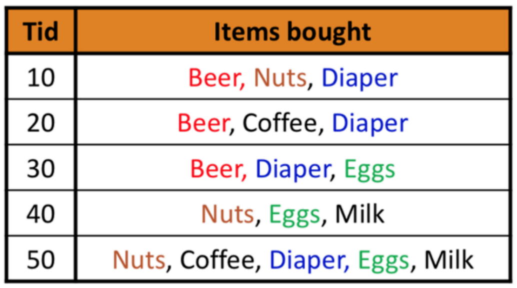
The FP-Growth Algorithm Application

Introduction

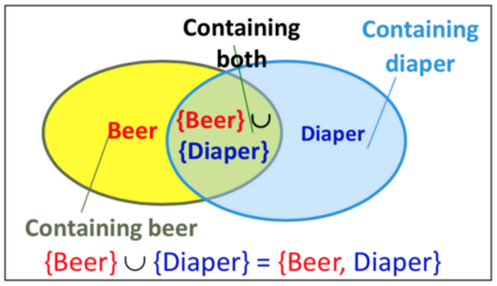
Discovery of association rules is one of important technique in which market basket analysis, cross-marking, catalog design, sale campaign analysis, Web log analysis and biological sequence analysis fields. In this circumstances, frequent pattern mining has become an important data mining task and a focused theme in data mining research.

Frequent patterns(FIG 1) are patterns that appear frequently in a data set. It represents the essential characteristics of the data set and important attributes. A set of items appear frequently together in a transaction data set is a frequent itemset. Pattern discovery is the discovery of patterns from a large set of data.

FIG 1: EXAMPLE of FREQUENT PATTERN



Tid	Items bought
10	Beer, Nuts, Diaper
20	Beer, Coffee, Diaper
30	Beer, Diaper, Eggs
40	Nuts, Eggs, Milk
50	Nuts, Coffee, Diaper, Eggs, Milk



Finding frequent patterns plays an essential role in mining associations, correlations and many other interesting relationships among data. Moreover, it helps in data classification, clustering and other data mining tasks.

Several previous algorithms have been proposed to solve this problem. Most of them require scanning the transactions database repeatedly before the frequent item-sets are to be generated, which incurs huge cost of CPU execution time and memory spaces. The time that costs in Association rules is to generate the frequent itemsets. Since generating frequent itemsets needs to test a lot of alternative sets, so the time complexity reaches O(n^2) if not optimized.

Most of the previous studies such as Lnt eet al.(1997), Ng et al. (1998) and Grahne et al. (2000) adopt an Apriori-like approach which proposed that some itemsets which will not become to the frequent itemsets should be eliminated as soon as possible, thus, to achieve fast mining of frequent itemsets. However, Apriori algorithm base on candidate elimination, it needs to scan all the data records once each time to eliminate. This feature causes it seems powerless in the face of big data sets.

In 2000, Han proposed FP-Growth Algorithm that is an efficient and scalable method for mining the complete set of frequent patterns by pattern fragment growth, using an extended prefix-tree structure for storing compressed and crucial information about frequent patterns named FP- tree. This study shows that FP-growth is about an order of magnitude faster than Apriori, especially when the data set is dense (containing many patterns) and/or when the frequent patterns are long.

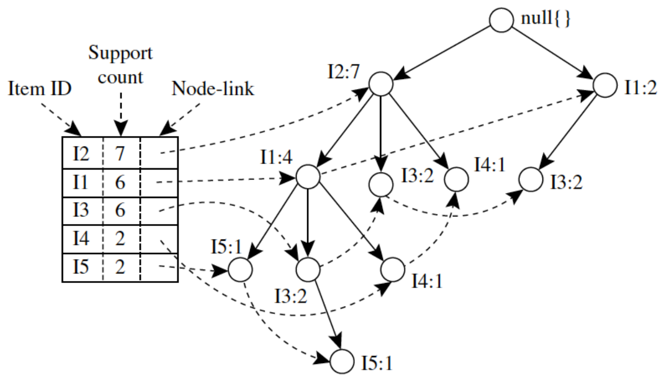
Experimental Design

FP-Growth algorithm compresses the data record by constructing a tree structure, which makes it only need to scan the data record twice, and the algorithm does not need to generate the candidate set, and make it higher efficiency.

- Frequent-pattern Tree Construction

The frequent-pattern tree(FP-tree)(FIG 2) is a compact structure that stores quantitative information about frequent patterns in a database. Let I = {a1, a2,……, am} be a set of items, and a transaction database DB={T1, T2,….,Tn}, where Ti(i∈ [1…n]) is a transaction which contains a set of items, is the number of transactions containing A in DB.

FIG 2 : An FP-Tree registers compressed, frequent pattern information



Step 1: Scan data records（Table 1), generate a frequent itemsets in an order of the number of occurrences from more to less, for example(Table 2):

Table 1 : Data Records

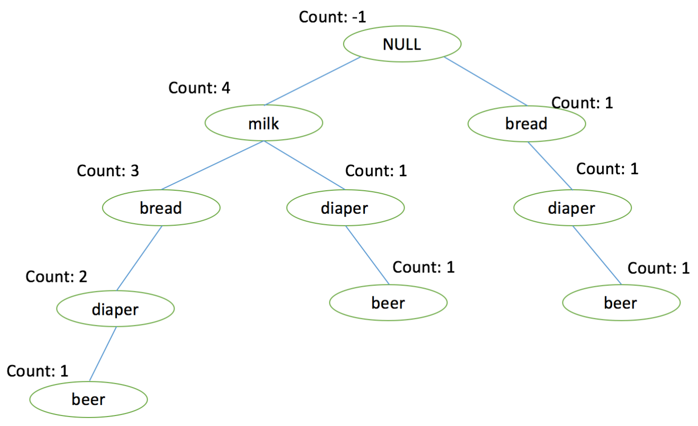
TID	Items
T1	{milk, bread}
T2	{bread, diaper, beer, egg}
T3	{milk, diaper, beer, coke}
T4	{bread, milk, diaper, beer}
T5	{bread, milk, diaper, coke}

Table 2 : Frequent Itemsets

Item	Count
Milk	4
Bread	4
Diaper	4
Beer	3

Step 2: Scan the data record again, sorting the entries in the table that are generated in step 1 for each record according the order in the table, see the result below (FIG 3)

FIG 3 : Frequent Pattern Tree



2. Mining Frequent Pattern using FP-Tree

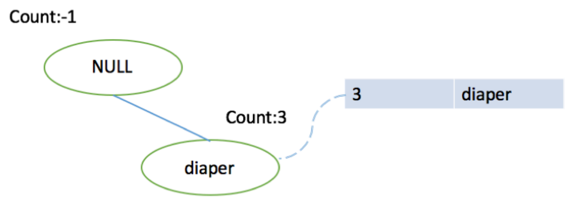
Once FP-Tree is completed, the mainly task is exploring the compact information stored in an FP-Tree, developing the principles of frequent pattern growth and finding a way to perform further optimization when there exists a single prefix path in the FP-Tree.

By following the example above, first, consider the last item in header table. Follow the FP-Tree above, from the node {beer} can find branch as {milk, bread, diaper, beer : 1}, {milk, diaper, beer : 1} , {bread, diaper, beer : 1}.The number 1 presents the frequency of occurrence. Even though the count of milk is 4, {milk, bread, diaper, beer} only occurs once, so the count of the branch depend on the count of the suffix node {beer}. According to the prefix path to generate a condition FP-Tree by the step listed before, thus, the records become to (Table 3), and since the support count still is 3, so the sub-FP-Tree is constructed as below(FIG 4).

Table 3: Data Records

TID	Items
T1	{ milk, bread, diaper}
T2	{milk, diaper}
T3	{bread, diaper}

FIG 4 : Sub-FP-Tree



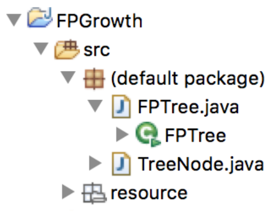
Repeat the above steps to get the entire frequent itemsets by digging each item of the header tables.

Implementation and Results

1. Implementation

The FP-Growth algorithm implementation (FIG 6) contains two classes which deal with TreeNode and mining the FP-Tree.

FIG 6 : Implementation of FP-Growth algorithm

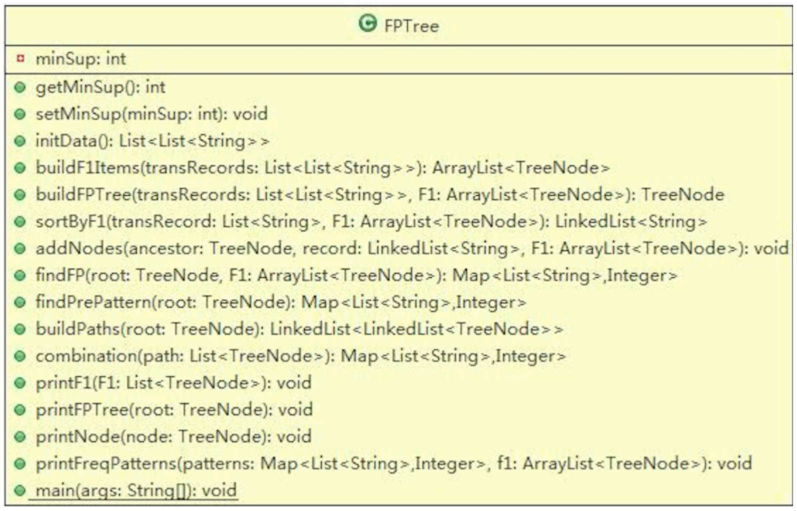


In the TreeNode.java, build the treeNode (FIG 7) with methods such as getParent, getChildren, addChild, getNextHomonym and so on.
It contains one root as null, a set of item-prefix subtrees as the children of the root and a frequent itemsets header table. Each node in the item-prefix subtree consists of 3 fields: item-name, count and node-link, where item-name registers which item this node represents, cunt 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 empty.
Each entry in the frequent itemsets header table has item-name and head of node-link.

```
public class TreeNode implements Comparable<TreeNode> {
   private String name;
   private int count;
   private TreeNode parent;
   private List<TreeNode> children;
   private TreeNode nextHomonym;
   }
```

In the FPTree.java(FIG 7), dealing with initialize data records, construction frequent itemsets, building the frequent tree and finding the FP in the tree.

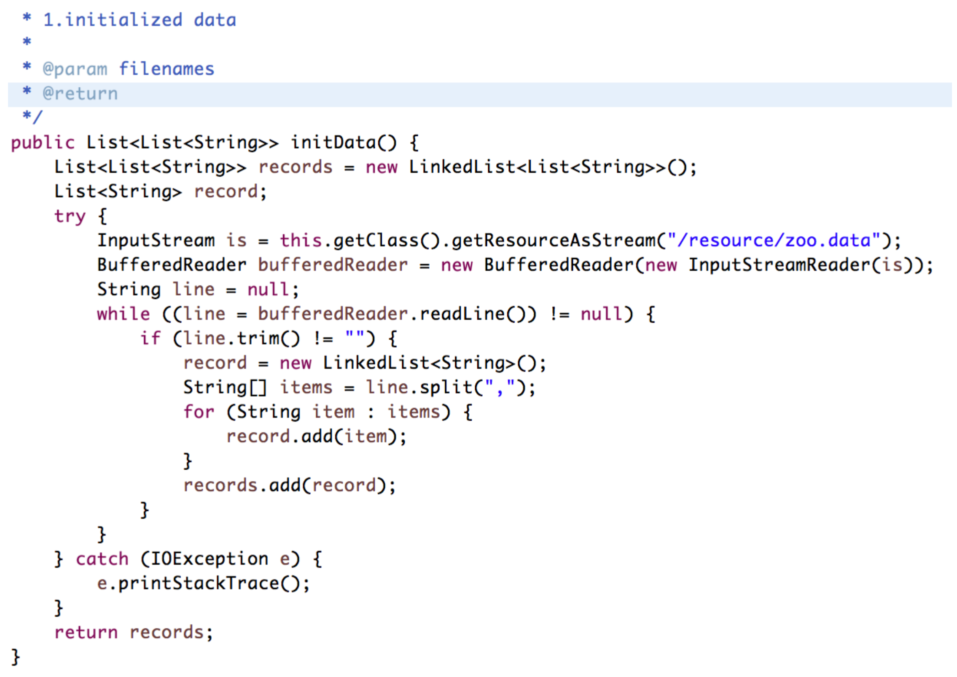
FIG 7 : Implementation of FP-Tree



```
FPTree
minSup: int
getMinSup(): int
setMinSup(minSup: int): void
initData(): List<List<String>>
buildF1Items(transRecords: List<List<String>>): ArrayList<TreeNode>
buildFPTree(transRecords: List<List<String>>, F1: ArrayList<TreeNode>): TreeNode
sortByF1(transRecord: List<String>, F1: ArrayList<TreeNode>): LinkedList<String>
addNodes(ancestor: TreeNode, record: LinkedList<String>, F1: ArrayList<TreeNode>): void
findFP(root: TreeNode, F1: ArrayList<TreeNode>): Map<List<String>,Integer>
findPrePattern(root: TreeNode): Map<List<String>,Integer>
buildPaths(root: TreeNode): LinkedList<LinkedList<TreeNode>>
combination(path: List<TreeNode>): Map<List<String>,Integer>
printF1(F1: List<TreeNode>): void
printFPTree(root: TreeNode): void
printNode(node: TreeNode): void
printFreqPatterns(patterns: Map<List<String>,Integer>, f1: ArrayList<TreeNode>): void
main(args: String[]): void
```

The first step is initialize the data records, see the implementation(FIG 8) below:

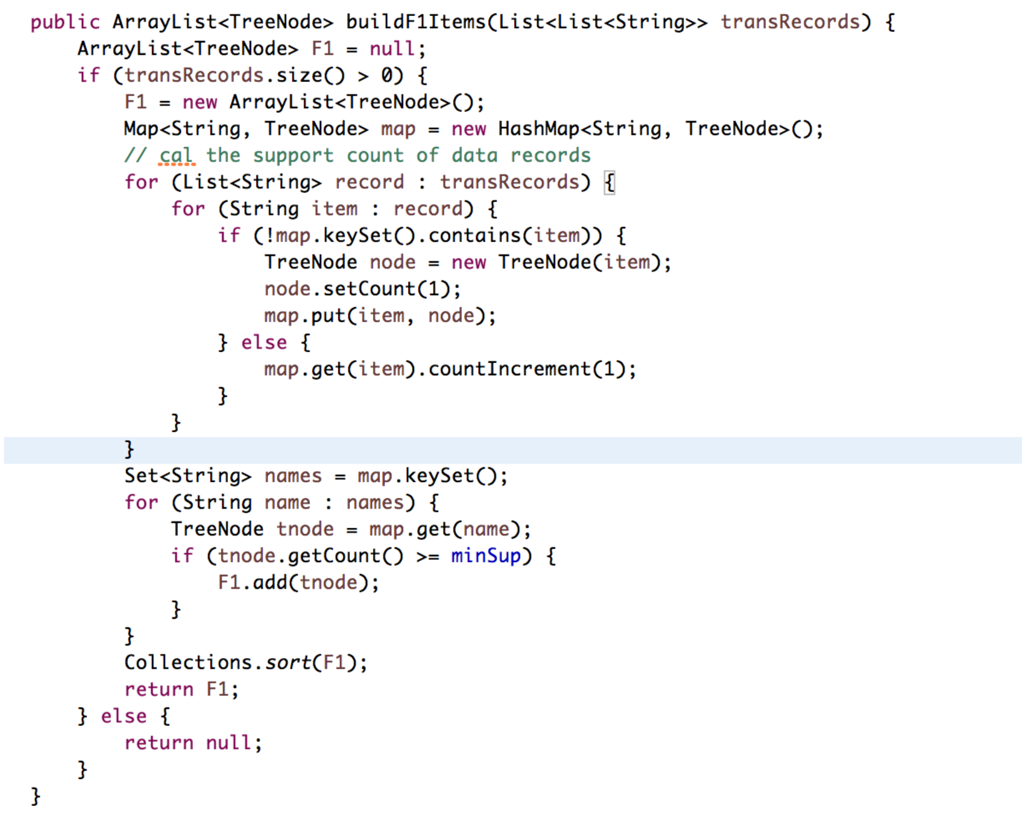
FIG 8： The Data Records Initialization



```
 * 1.initialized data
 *
 * @param filenames
 * @return
 */
public List<List<String>> initData() {
    List<List<String>> records = new LinkedList<List<String>>();
    List<String> record;
    try {
        InputStream is = this.getClass().getResourceAsStream("/resource/zoo.data");
        BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(is));
        String line = null;
        while ((line = bufferedReader.readLine()) != null) {
            if (line.trim() != "") {
                record = new LinkedList<String>();
                String[] items = line.split(",");
                for (String item : items) {
                    record.add(item);
                }
                records.add(record);
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return records;
}
```

The second step: constructing frequent itemsets (FIG 9)

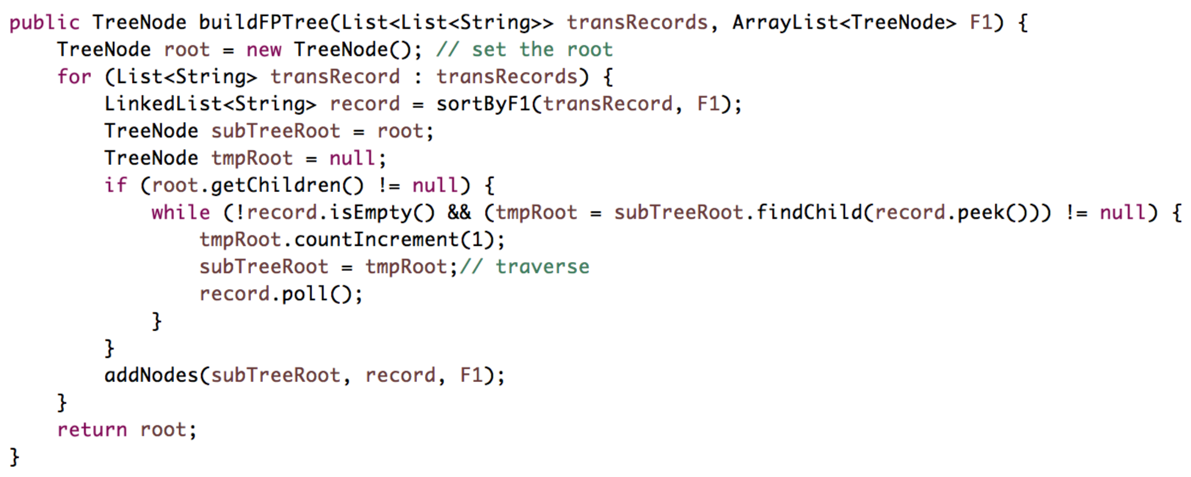
FIG 9： Constructing Frequent Itemsets



```
public ArrayList<TreeNode> buildF1Items(List<List<String>> transRecords) {
    ArrayList<TreeNode> F1 = null;
    if (transRecords.size() > 0) {
        F1 = new ArrayList<TreeNode>();
        Map<String, TreeNode> map = new HashMap<String, TreeNode>();
        // cal the support count of data records
        for (List<String> record : transRecords) {
            for (String item : record) {
                if (!map.keySet().contains(item)) {
                    TreeNode node = new TreeNode(item);
                    node.setCount(1);
                    map.put(item, node);
                } else {
                    map.get(item).countIncrement(1);
                }
            }
        }
        Set<String> names = map.keySet();
        for (String name : names) {
            TreeNode tnode = map.get(name);
            if (tnode.getCount() >= minSup) {
                F1.add(tnode);
            }
        }
        Collections.sort(F1);
        return F1;
    } else {
        return null;
    }
}
```

The third step: Building FP-Tree (FIG 10)

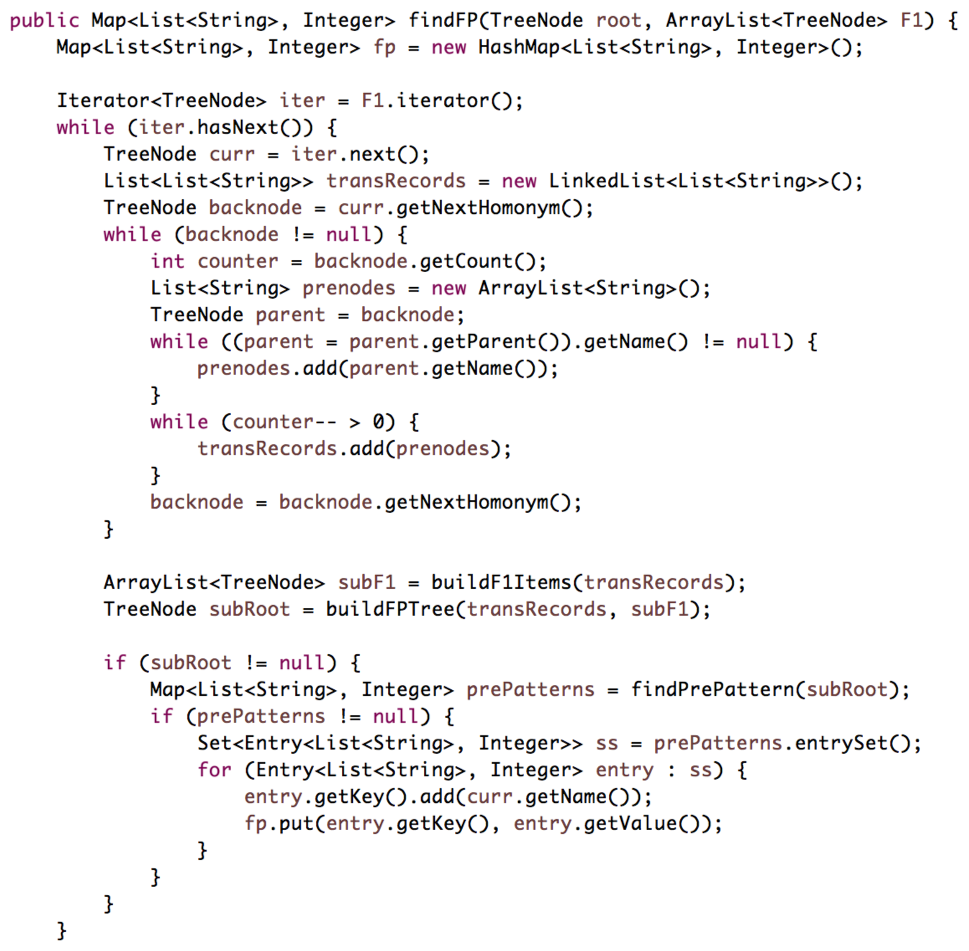
FIG 10 :　Building FP-Tree



```
public TreeNode buildFPTree(List<List<String>> transRecords, ArrayList<TreeNode> F1) {
    TreeNode root = new TreeNode(); // set the root
    for (List<String> transRecord : transRecords) {
        LinkedList<String> record = sortByF1(transRecord, F1);
        TreeNode subTreeRoot = root;
        TreeNode tmpRoot = null;
        if (root.getChildren() != null) {
            while (!record.isEmpty() && (tmpRoot = subTreeRoot.findChild(record.peek())) != null) {
                tmpRoot.countIncrement(1);
                subTreeRoot = tmpRoot;// traverse
                record.poll();
            }
        }
        addNodes(subTreeRoot, record, F1);
    }
    return root;
}
```

The last step: Finding the frequent pattern from a FP-Tree (FIG 11)

FIG 11 :　Finding the FP

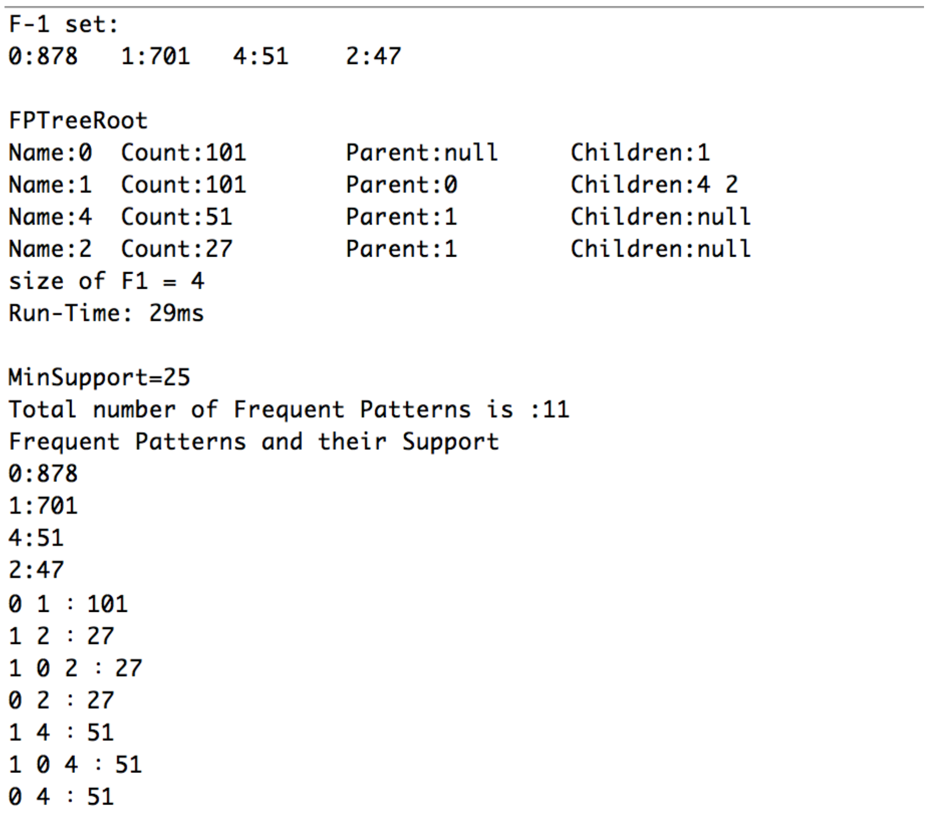


```
public Map<List<String>, Integer> findFP(TreeNode root, ArrayList<TreeNode> F1) {
    Map<List<String>, Integer> fp = new HashMap<List<String>, Integer>();

    Iterator<TreeNode> iter = F1.iterator();
    while (iter.hasNext()) {
        TreeNode curr = iter.next();
        List<List<String>> transRecords = new LinkedList<List<String>>();
        TreeNode backnode = curr.getNextHomonym();
        while (backnode != null) {
            int counter = backnode.getCount();
            List<String> prenodes = new ArrayList<String>();
            TreeNode parent = backnode;
            while ((parent = parent.getParent()).getName() != null) {
                prenodes.add(parent.getName());
            }
            while (counter-- > 0) {
                transRecords.add(prenodes);
            }
            backnode = backnode.getNextHomonym();
        }

        ArrayList<TreeNode> subF1 = buildF1Items(transRecords);
        TreeNode subRoot = buildFPTree(transRecords, subF1);

        if (subRoot != null) {
            Map<List<String>, Integer> prePatterns = findPrePattern(subRoot);
            if (prePatterns != null) {
                Set<Entry<List<String>, Integer>> ss = prePatterns.entrySet();
                for (Entry<List<String>, Integer> entry : ss) {
                    entry.getKey().add(curr.getName());
                    fp.put(entry.getKey(), entry.getValue());
                }
            }
        }
    }
```

2. Results

By using the zoo data from the specific data sets, below is the final output (FIG 12).

FIG 12 : Output



```
F-1 set:
0:878   1:701   4:51    2:47

FPTreeRoot
Name:0  Count:101       Parent:null     Children:1
Name:1  Count:101       Parent:0        Children:4 2
Name:4  Count:51        Parent:1        Children:null
Name:2  Count:27        Parent:1        Children:null
size of F1 = 4
Run-Time: 29ms

MinSupport=25
Total number of Frequent Patterns is :11
Frequent Patterns and their Support
0:878
1:701
4:51
2:47
0 1 : 101
1 2 : 27
1 0 2 : 27
0 2 : 27
1 4 : 51
1 0 4 : 51
0 4 : 51
```

Conclusion

The program successfully applied the FP-Growth Algorithm to find the frequent pattern. Since it hasn't been compared with other algorithm, so it is hard to say FP-Growth is really better than other algorithm by this project. But pervious studies already showed the advantages of FP-Growth, so, I am believe it is a right approach to find the frequent pattern from a huge data set.