Implementing FP Growth Algorithm

Abstract

Mining frequent patterns in transaction databases, time-series databases, and many other kinds of databases has been studied popularly in data mining research. Most of the previous studies adopt an Apriori-like candidate set generation-and-test approach. However, candidate set generation is still costly, especially when there exist prolific patterns and/or long patterns.

In this study, I have implemented a novel frequent pattern tree (FP-tree) structure, which is an extended prefix-tree structure for storing compressed, crucial information about frequent patterns, and develop an efficient FP-tree-based mining method, FP-growth, for mining the complete set of frequent patterns by pattern fragment growth.

Introduction

The **Frequent Pattern Growth (FP-Growth) algorithm** is a widely used technique for mining frequent item-sets in a transaction dataset. The FP-Growth Algorithm proposed by ***Han in***. This 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 frequent-pattern tree (FP-tree).

The algorithm is primarily used for association rule learning and market basket analysis to uncover relationships between items in transactional data. It identifies item-sets that occur together frequently, which is useful in areas like retail, e-commerce, and recommendation systems. Unlike the Apriori algorithm, which generates candidate sets and iteratively reduces them, FP-Growth uses a more efficient method that avoids candidate generation, making it faster and more scalable for large datasets.

Methodology

1. **Construct the FP-Tree**:
    - **Scan the dataset** to calculate the frequency of each item.
    - **Filter out items** that do not meet the minimum support threshold.
    - **Sort items** in descending order of frequency to maintain consistency across transactions.
    - **Build the tree** by inserting transactions, ensuring that items follow the same order at each insertion. Nodes are created for each item, with links to parent nodes and links connecting identical items for traversal.
2. **Mine the FP-Tree**:
    - **Start from the least frequent item** in the header table and form conditional patterns based on paths leading to that item.
    - **Construct conditional FP-trees** for each item using its conditional pattern base (a sub-database of paths).
    - **Recursively mine** each conditional FP-tree to find frequent item-sets, adding them to the list of results.

From the FP-tree construction process, we can see that one needs exactly two scans of the transaction database, the first collects the set of frequent items, and the second constructs the FP-tree. The cost of inserting a transaction T into the FP-tree is $O(|T|)$, where $|T|$ is the number of frequent items in Transactions. We will show that the FP-tree contains the complete information for frequent pattern mining.

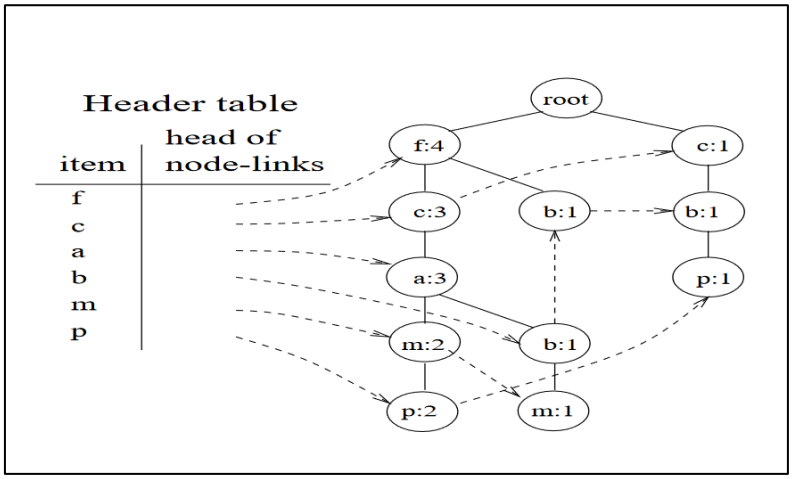

Figure 1: The FP Tree

Implementation

```
public class FPGrowth {
    private final int minSupport;
    private final Map<String, Integer> patternSupport;

    public FPGrowth(int minSupport) {
        this.minSupport = minSupport;
        this.patternSupport = new HashMap<>();
    }

    public Map<String, Integer> createFrequencyMap(List<List<String>> transactions) {
        Map<String, Integer> itemFrequencies = new HashMap<>();
        for (List<String> transaction : transactions) {
            for (String item : transaction) {
                itemFrequencies.merge(item, 1, Integer::sum);
            }
        }

        itemFrequencies.entrySet().removeIf(entry -> entry.getValue() < minSupport);

        return itemFrequencies.entrySet().stream().sorted((entry1, entry2) ->
entry2.getValue().compareTo(entry1.getValue())).collect(Collectors.toMap(Map.Entry::getKey,
Map.Entry::getValue, (e1, e2) -> e1, LinkedHashMap::new));
    }

    public List<List<String>> createOrderedItemSet(Map<String, Integer> sortedMap,
List<List<String>> transactions) {
        Set<String> validItems = sortedMap.keySet();
        List<List<String>> filteredTransactions = new ArrayList<>();
        for (List<String> transaction : transactions) {
            Set<String> filteredTransactionSet = new HashSet<>();
```

```
            for (String item : transaction) {
                if (validItems.contains(item)) {
                    filteredTransactionSet.add(item);
                }
            }
            filteredTransactions.add(new ArrayList<>(filteredTransactionSet));
        }
        return filteredTransactions;
    }

    public Map<String, Integer> findFrequentPatterns(List<List<String>> transactions) {
        FPTree tree = new FPTree(minSupport);
        tree.buildFPTree(transactions);

        minePatterns(tree, new ArrayList<>());
        return patternSupport;
    }

    public void minePatterns(FPTree tree, List<String> prefix) {
        if (tree.root.children.isEmpty()) {
            return;
        }
        for (Map.Entry<String, FPNode> entry : tree.headerTable.entrySet()) {
            String item = entry.getKey();
            List<String> newPattern = new ArrayList<>(prefix);
            newPattern.add(item);
            int support = 0;
            FPNode node = entry.getValue();
            while (node != null) {
                support += node.count;
                node = node.link;
            }

            if (support >= minSupport) {
                if (newPattern.size() > 1) { // Only store patterns with 2 or more items
                    List<String> sortedPattern = new ArrayList<>(newPattern);
                    String patternKey = String.join(",", sortedPattern);
                    patternSupport.put(patternKey, support);
                }

                List<List<String>> conditionalPatternBase = new ArrayList<>();
                node = entry.getValue();

                while (node != null) {
                    List<String> path = new ArrayList<>();
                    FPNode current = node.parent;


                    while (current.item != null) {
                        path.addFirst(current.item);
                        current = current.parent;
                    }
```

```
                if (!path.isEmpty()) {
                    for (int i = 0; i < node.count; i++) {
                        conditionalPatternBase.add(new ArrayList<>(path));
                    }
                }
                node = node.link;
            }
            if (!conditionalPatternBase.isEmpty()) {
                FPTree conditionalTree = new FPTree(minSupport);
                conditionalTree.buildFPTree(conditionalPatternBase);
                minePatterns(conditionalTree, newPattern);
            }
        }
    }
}

public StringBuilder formatOutput(Map<String, Integer> frequentPatterns) {
    StringBuilder output = new StringBuilder();

    List<Map.Entry<String, Integer>> sortedPatterns = new ArrayList<>(frequentPatterns.entrySet());
    sortedPatterns.sort((e1, e2) -> {
        int supportCompare = e2.getValue().compareTo(e1.getValue());
        if (supportCompare != 0) return supportCompare;
        return e1.getKey().compareTo(e2.getKey());
    });
    for (Map.Entry<String, Integer> entry : sortedPatterns) {
        output.append(entry.getKey()).append(" = ").append(entry.getValue());
        output.append("\n");
    }
    return output;
}
}
```

Results

The FP-Growth algorithm's efficiency in handling the dataset confirmed its advantages over traditional algorithms. The construction of the FP-tree and subsequent mining operations were performed without generating unnecessary candidates, thus optimizing processing time and memory use. Larger item-sets consisting of six or more products, demonstrated the algorithm’s capability to uncover complex relationships that might not be immediately obvious through basic analysis.

Table 1: Input Data

TID	Items Bought
100	F-A-C-D-G-I-M-P
200	A-B-C-F-L-M-O
300	B-F-H-J-O
400	B-C-K-S-P
500	A-F-C-E-L-P-M-N

```
Transactions:
[[F, A, C, D, G, I, M, P], [A, B, C, F, L, M, O], [B, F, H, J, O], [B, C, K, S, P], [A, F, C, E, L, P, M, N]]

Frequency Map:
{P=3, A=3, B=3, C=4, M=3, F=4}

Ordered-Item Set:
[[P, A, C, F, M], [A, B, C, F, M], [B, F], [P, B, C], [P, A, C, F, M]]

Frequent Patterns:
A,C = 3
A,F = 3
A,F,C = 3
F,C = 3
M,A = 3
M,C = 3
M,C,A = 3
M,F = 3
M,F,A = 3
M,F,C = 3
M,F,C,A = 3
P,C = 3
```

Figure 2: Output of Algorithm

Conclusion

The **FP-Growth algorithm** stands as a powerful and efficient solution for mining frequent item-sets from large datasets. By eliminating the need for candidate generation and employing a compact data structure in the form of an FP-tree, it significantly reduces the computational overhead compared to traditional methods like the Apriori algorithm. This makes FP-Growth particularly suitable for real-world applications where datasets are vast and require scalable solutions.

The algorithm's use in various domains—ranging from market basket analysis to web usage mining and bioinformatics—demonstrates its versatility and impact in uncovering hidden patterns that can inform decision-making and strategic initiatives. Despite its efficiency, careful consideration should be given to memory management when working with extensive or highly dense datasets, as the construction of the FP-tree may demand significant resources.