A Project Report

on

s

FP Growth Algorithm

by

Harsh Shah

21BCP359

Under the Guidance of

Dr. Archana Nigam

Assistant Professor

For the Course

Mahine Learning Lab

20CP401P

Submitted to

**Department of Computer Science & Engineering,
School of Technology, Pandit Deendayal Energy University,
Gandhinagar 382 426**

November 2024

INDEX

Sr. No.	Title	Page
1	Problem Statement	3
2	Introduction	4
3	Dataset Description	5
4	Methodology	6
5	Results and Insights	7
6	Challenges Faced	8
7	Conclusion and Future Work	9
8	Code and Output	10

PROBLEM STATEMENT

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 project 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.

DATASET DESCRIPTION

The dataset utilized for the project consists of a series of transactions, each recording the items purchased by customers. The data includes transactions, with various products represented by unique letters. This type of dataset, often referred to as "market basket data", captures customer purchasing behaviour and serves as the basis for discovering frequent item-sets and association rules.

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

Below is a brief description and analysis of the dataset:

- The dataset shown in *Table 1* contains two main columns: **TID (Transaction ID)** and **Items Bought**.
- Each entry under "Items Bought" lists products purchased in a transaction, represented by uppercase letters, separated by hyphens (e.g., *"F-A-C-D-G-I-M-P"*).
- The dataset features a mix of commonly occurring items such as *A*, *B*, *C*, *F*, and *M*, as well as less frequent items such as *D*, *G*, and *S*.
- Certain products, such as *A*, *C*, and *F*, appear in multiple transactions, suggesting their popularity and potential frequent itemset status.

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.

The FP-Tree structure in *Figure 1* allows for the compression of large datasets into a compact representation that still retains the necessary information for mining.

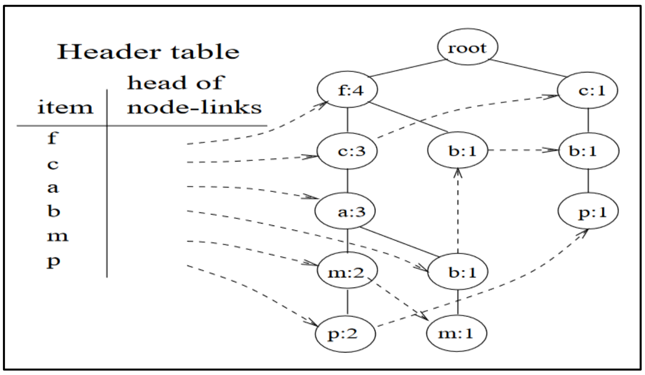


Figure 1: The FP Tree

RESULTS AND INSIGHTS

The dataset consisting of transactions with various products was analysed using the FP-Growth algorithm to mine frequent patterns. Below, we break down the results and provide an analysis of the key findings:

Frequency Map:

The first step involved scanning the dataset to calculate the frequency of each item. The frequency map generated from this scan was: {P: 3, A: 3, B: 3, C: 4, M: 3, F: 4}

This map indicates how many times each item appeared across all transactions. The minimum support threshold for pattern mining was set to 3 such that only items meeting or exceeding this threshold were retained.

Ordered Item Sets:

After filtering and sorting the items by their frequency, transactions were restructured into ordered item sets: [[P, A, C, F, M], [A, B, C, F, M], [B, F], [B, P, C], [P, A, C, F, M]]

Each transaction was sorted in descending order of item frequency, ensuring consistency and facilitating the construction of the FP-Tree. Items not meeting the minimum support threshold were omitted, which streamlined the dataset for more efficient mining.

Frequent Patterns:

The frequent patterns discovered, along with their respective support counts are shown in *Table 2*.

Table 2: Frequent Patterns Observed

{C, P}: 3	{M, F}: 3
{F, A}: 3	{F, C}: 4
{C, A}: 3	{C, A, M}: 3
{F, C, A}: 4	{F, A, M}: 3
{M, A}: 3	{F, C, M}: 3
{M, C}: 3	{F, C, A, M}: 3

The high frequency of item-sets like {F, C} and {A, F, C} highlights product combinations that are often purchased in tandem, suggesting potential product bundling opportunities for businesses. Additionally, the presence of item {M} in several frequent item-sets indicates it plays a significant role in purchasing patterns within the dataset.

CHALLENGES FACED

Implementing the FP-Growth algorithm presented several challenges, including data pre-processing and memory management. Ensuring consistent data formats, sorting items by frequency, and managing large data structures proved difficult. Handling memory efficiently during the tree construction and traversal phases was essential, especially when scaling to larger datasets, as it significantly impacted the system's performance.

Additionally, selecting an optimal minimum support threshold was challenging. A low threshold generated excessive patterns, making analysis complex, while a higher one risked omitting valuable insights. Interpreting the results to extract meaningful patterns required careful consideration and clear visualization strategies to present actionable findings, underscoring the balance needed between computational efficiency and interpretability.

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.

CODE

The program is divided into 4 modules (Java files): ***Main, FPGrowth, FPComponents, FileOps***.

Main.java - *Contains the Main Class which performs the algorithm step-by-step*

```
public class Main {
   public static void main(String[] args) {
      String fileName = "test";
      String inputFile = "data/" + fileName + ".csv";
      String outPutFile = "results/" + fileName + ".txt";
      int minSupport = 3;
      FPGrowth fpGrowth = new FPGrowth(minSupport);

      // Read transactions from CSV
      List<List<String>> transactions = FileOps.readCSV(inputFile);

      // 1. Create Frequency Map
      Map<String, Integer> sortedMap = new
HashMap<>(fpGrowth.createFrequencyMap(transactions));

      // 2. Create Ordered-Item Set
      List<List<String>> filteredTransactions = new
ArrayList<>(fpGrowth.createOrderedItemSet(sortedMap, transactions));

      // 3. Create Frequent Pattern Tree
      Map<String, Integer> frequentPatterns = fpGrowth.findFrequentPatterns(filteredTransactions);

      // Write results to file
      String output = "Transactions: \n" + transactions + "\n" +
            "\nFrequency Map: \n" + sortedMap + "\n" +
            "\nOrdered-Item Set: \n" + filteredTransactions + "\n" +
            "\nFrequent Patterns:\n" + fpGrowth.formatOutput(frequentPatterns);
      FileOps.writeToFile(output, outPutFile);
   }
}
```

FPGrowth.java - *Contains the logic for FPGrowth algorithm*

```
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) {
            // Store pattern with its support count
            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;
}
```

FPComponents.java - _Contains **FPNode**, **FPTree** classes which are data structures for the FPTree_

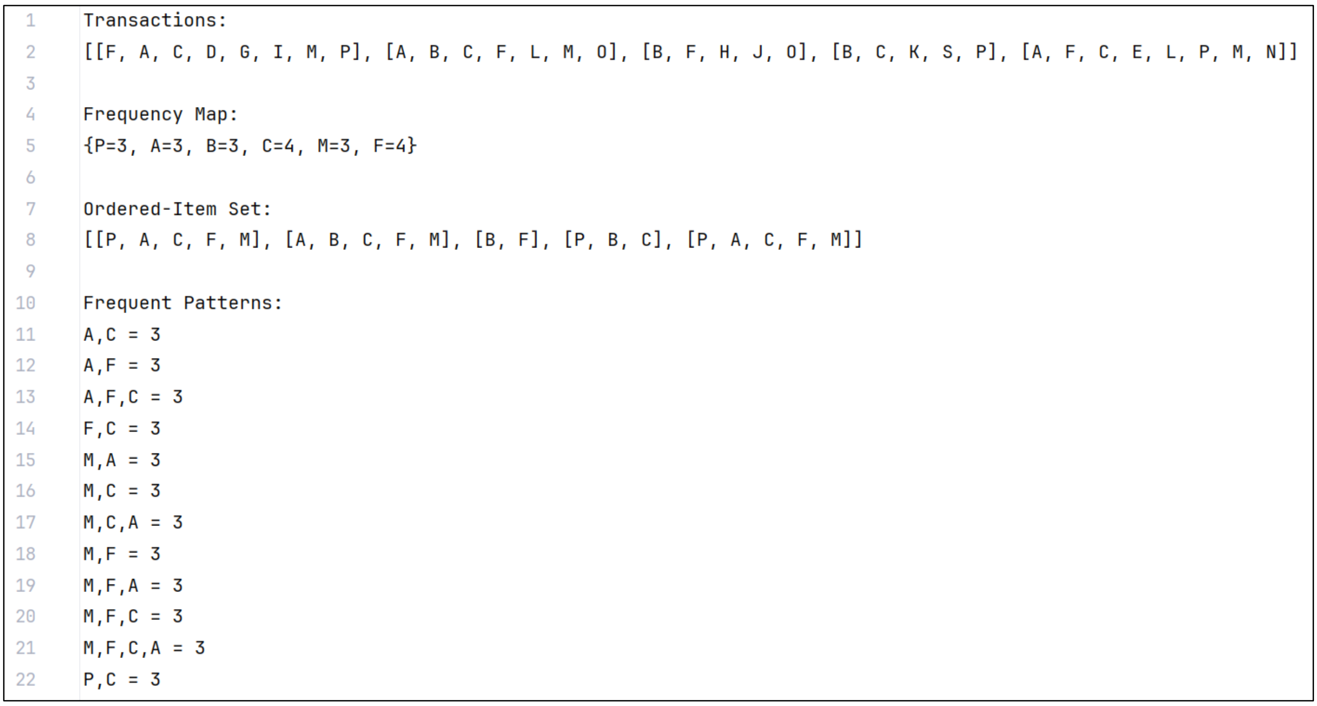
```
class FPNode {
    String item;
    int count;
    FPNode parent;
    Map<String, FPNode> children;
    FPNode link;

    public FPNode(String item, int count) {
        this.item = item;
        this.count = count;
        this.children = new HashMap<>();
        this.link = null;
        this.parent = null;
    }

    public void increment(int count) {
        this.count += count;
    }
}
class FPTree {
    FPNode root;
    Map<String, FPNode> headerTable;
    int minSupport;

    public FPTree(int minSupport) {
        this.root = new FPNode(null, 0);
        this.headerTable = new HashMap<>();
        this.minSupport = minSupport;
    }

    public void buildFPTree(List<List<String>> transactions) {
        Map<String, Integer> itemFrequency = new HashMap<>();

        for (List<String> transaction : transactions) {
            for (String item : transaction) {
                itemFrequency.put(item, itemFrequency.getOrDefault(item, 0) + 1);
            }
        }

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

        for (List<String> transaction : transactions) {
            List<String> modifiableTransaction = new ArrayList<>(transaction);

            modifiableTransaction.sort((a, b) -> {
                int freqCompare = itemFrequency.getOrDefault(b,
0).compareTo(itemFrequency.getOrDefault(a, 0));
                return freqCompare != 0 ? freqCompare : a.compareTo(b);
            });

            modifiableTransaction.removeIf(item -> !itemFrequency.containsKey(item));

            insertTransaction(modifiableTransaction);
        }
    }

    private void insertTransaction(List<String> transaction) {
        FPNode current = root;
        for (String item : transaction) {
            current = insertNode(current, item);
        }
    }

    private FPNode insertNode(FPNode parent, String item) {
        FPNode node = parent.children.get(item);
        if (node == null) {
            node = new FPNode(item, 1);
            node.parent = parent;
            parent.children.put(item, node);

            if (!headerTable.containsKey(item)) {
                headerTable.put(item, node);
            } else {
                FPNode link = headerTable.get(item);
                while (link.link != null) {
                    link = link.link;
                }
                link.link = node;
            }
        } else {
            node.increment(1);
        }
        return node;
    }
}
```

FileOps.java - *Contains methods for reading CSV file and writing results into file.*

```
public class FileOps {
    public static List<List<String>> readCSV(String filePath) {
        List<List<String>> transactions = new ArrayList<>();
        String line;

        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            br.readLine(); // Skip header row
            while ((line = br.readLine()) != null) {
                String[] parts = line.split(",");
                if (parts.length == 2) {
                    List<String> productList = Arrays.asList(parts[1].split("-"));
                    transactions.add(productList);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        return transactions;
    }

    public static void writeToFile(String frequentPatterns, String filePath) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath))) {
            writer.write(frequentPatterns);

            System.out.println("Output written to " + filePath);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

OUTPUT

Figure 2 shows the output of the program. The output is stored in a text file for future use.



```
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