| **Name:** | **James Lewis** |
| --- | --- |
| **Roll No:** | **32** |
| **Class/Sem:** | TE/V |
| **Experiment No.:** | 9 |
| **Title:** | Implementation of association mining algorithms like FP Growth using languages like JAVA/ python. |
| **Date of Performance:** |  |
| **Date of Submission:** |  |
| **Marks:** |  |
| **Sign of Faculty:** |  |

#### 

#### Aim :-To implement the FP-Growth algorithm using Python.

#### Objective: Understand the working principles of the FP-Growth algorithm and implement it in Python.

#### Theory

FP-Growth (Frequent Pattern Growth) is an algorithm for frequent item set mining and association rule learning over transactional databases. It efficiently discovers frequent patterns by constructing a compact data structure called the FP-Tree and mining it to extract frequent item sets.

Key Concepts:

1. FP-Tree: A data structure that represents the transaction database compressed by linking frequent items in a tree structure, along with their support counts.
2. Header Table: A compact structure that stores pointers to the first occurrences of items in the FP-Tree and their support counts.
3. Frequent Item Set Mining:
   * Conditional Pattern Base: For each frequent item, construct a conditional pattern base consisting of the prefix paths in the FP-Tree.
   * Conditional FP-Tree: Construct a conditional FP-Tree from the conditional pattern base and recursively mine frequent item sets.

Steps in FP-Growth Algorithm:

1. Build FP-Tree: Construct the FP-Tree by inserting transactions and counting support for each item.
2. Create Header Table: Build a header table with links to the first occurrences of items in the FP-Tree.
3. Mine FP-Tree:
   * Identify frequent single items by their support.
   * Construct conditional pattern bases and conditional FP-Trees recursively.
   * Combine frequent item sets from conditional FP-Trees to find all frequent item sets.

#### Example

Given a transactional database:

* Implement the FP-Growth algorithm to find all frequent itemsets with a specified minimum support threshold.

**Code:  
  
from collections import defaultdict, namedtuple**

**class FPNode:**

**def \_\_init\_\_(self, item, count, parent):**

**self.item = item**

**self.count = count**

**self.parent = parent**

**self.children = {}**

**self.link = None**

**def increment(self, count):**

**self.count += count**

**def build\_fp\_tree(transactions, min\_support):**

**item\_counts = defaultdict(int)**

**for transaction in transactions:**

**for item in transaction:**

**item\_counts[item] += 1**

**item\_counts = {item: count for item, count in item\_counts.items() if count >= min\_support}**

**if not item\_counts:**

**return None, None**

**header\_table = {item: [count, None] for item, count in item\_counts.items()}**

**root = FPNode(None, 1, None)**

**for transaction in transactions:**

**sorted\_items = [item for item in transaction if item in item\_counts]**

**sorted\_items.sort(key=lambda x: item\_counts[x], reverse=True)**

**current\_node = root**

**for item in sorted\_items:**

**if item in current\_node.children:**

**current\_node.children[item].increment(1)**

**else:**

**new\_node = FPNode(item, 1, current\_node)**

**current\_node.children[item] = new\_node**

**if header\_table[item][1] is None:**

**header\_table[item][1] = new\_node**

**else:**

**link\_node = header\_table[item][1]**

**while link\_node.link is not None:**

**link\_node = link\_node.link**

**link\_node.link = new\_node**

**current\_node = current\_node.children[item]**

**return root, header\_table**

**def find\_prefix\_paths(base\_item, node):**

**cond\_pats = []**

**while node is not None:**

**prefix\_path = []**

**parent = node.parent**

**while parent is not None and parent.item is not None:**

**prefix\_path.append(parent.item)**

**parent = parent.parent**

**prefix\_path.reverse()**

**if prefix\_path:**

**for \_ in range(node.count):**

**cond\_pats.append(prefix\_path)**

**node = node.link**

**return cond\_pats**

**def mine\_fp\_tree(header\_table, min\_support, prefix, frequent\_itemsets):**

**sorted\_items = sorted(header\_table.items(), key=lambda x: x[1][0])**

**for item, (support, node) in sorted\_items:**

**new\_freq\_set = prefix.copy()**

**new\_freq\_set.add(item)**

**frequent\_itemsets.append((new\_freq\_set, support))**

**cond\_pats = find\_prefix\_paths(item, node)**

**cond\_tree, cond\_header = build\_fp\_tree(cond\_pats, min\_support)**

**if cond\_header is not None:**

**mine\_fp\_tree(cond\_header, min\_support, new\_freq\_set, frequent\_itemsets)**

**def fp\_growth(transactions, min\_support):**

**root, header\_table = build\_fp\_tree(transactions, min\_support)**

**frequent\_itemsets = []**

**if header\_table is not None:**

**mine\_fp\_tree(header\_table, min\_support, set(), frequent\_itemsets)**

**return frequent\_itemsets**

**transactions = [**

**["milk", "bread", "nuts", "apple"],**

**["milk", "bread", "nuts"],**

**["milk", "bread"],**

**["milk", "bread", "apple"],**

**["milk", "apple"],**

**["bread", "nuts"],**

**["bread", "apple"]**

**]  
min\_support = 2**

**frequent\_itemsets = fp\_growth(transactions, min\_support)  
print("Frequent Itemsets with Support Counts:")**

**for itemset, support in frequent\_itemsets:**

**print(f"{itemset} : {support}")**

**Output:**

* List of all frequent itemsets along with their support counts.

Frequent Itemsets with Support Counts:

{'nuts'} : 3

{'milk', 'nuts'} : 2

{'bread', 'milk', 'nuts'} : 2

{'bread', 'nuts'} : 3

{'apple'} : 4

{'bread', 'apple'} : 3

{'milk', 'apple'} : 3

{'milk', 'bread', 'apple'} : 2

{'milk'} : 5

{'bread', 'milk'} : 4

{'bread'} : 6

#### Conclusion

**Explain how FP-Growth manages and mines item sets of varying lengths in transactional databases.**

1. Compact Representation with FP-Tree

* Instead of scanning the database repeatedly (like Apriori), FP-Growth compresses the data into a Frequent Pattern Tree (FP-Tree).
* This tree structure groups common prefixes of transactions, allowing efficient storage of transactions with different itemset lengths.

1. Recursive Mining Using Conditional FP-Trees

* FP-Growth does not generate candidate itemsets explicitly.
* For each frequent item, it builds a Conditional Pattern Base (sub-transactions leading to that item).
* From this, it constructs a Conditional FP-Tree and recursively mines frequent itemsets of increasing length.

1. Handling Itemsets of Varying Lengths

* The algorithm starts with single items (length-1 itemsets).
* It then recursively extends them into larger itemsets (length-2, length-3, etc.) by combining items from conditional trees.
* This process continues until no more frequent patterns can be formed, thus automatically handling itemsets of all lengths.

1. Efficiency in Mining

* Since the FP-Tree shares overlapping parts of transactions, longer itemsets are mined without scanning the whole database again.
* This makes FP-Growth highly efficient for large transactional datasets compared to Apriori.