Name:	Pranita Kumbhar
Roll No:	70
Class/Sem:	TE/V
Experiment No.:	9
Title:	Implementation of association mining algorithms like FP Growth using languages like JAVA/ python.
Date of Performance:	18/09/25
Date of Submission:	25/09/25
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:

```
#!/usr/bin/env python3
"""
fp_growth.py
Simple FP-Growth implementation in pure Python (no external libs).
Reads transactions from 'transactions.csv' (one transaction per line, items comma-separated)
Produces frequent itemsets with their support counts.

Usage:
    python fp_growth.py
Change MIN_SUPPORT to an integer (absolute) or float 0<val<=1 (fraction of transactions).
"""

import csv
import math
from collections import defaultdict

# -------------------------
# Configuration
# -------------------------
INPUT_CSV = "C:/Users/Pranita Kumbhar/Downloads/transactions.csv"
# MIN_SUPPORT can be:
# - integer >= 1  -> interpreted as absolute count
# - float between 0 and 1 -> interpreted as fraction of number of transactions (e.g. 0.3 -> 30%)
MIN_SUPPORT = 2  # change to e.g. 0.3 for 30% of transactions

# -------------------------
# FP-Tree node class
# -------------------------
class FPNode:
    def __init__(self, item_name, count, parent):
        self.item_name = item_name
        self.count = count
        self.parent = parent
        self.children = {}      # item_name -> FPNode
        self.node_link = None   # link to next node with same item_name

    def increment(self, n=1):
        self.count += n
```

```
# -------------------------
# Utility: load transactions
# -------------------------
def load_transactions_from_csv(path):
    """
    Accepts either:
     - each line: item1,item2,item3
     - or lines with an id and items: id,item1,item2,...
    Returns a list of transactions: [['bread','milk'], ...]
    """
    transactions = []
    with open(path, newline='') as csvfile:
        reader = csv.reader(csvfile)
        for row in reader:
            if not row:
                continue
            # If single cell, assume it's a comma-separated list of items
            if len(row) == 1:
                items = [i.strip() for i in row[0].split(',') if i.strip()]
            else:
                # If first column looks like an ID, join the rest and split
                rest = ','.join(row[1:]).strip()
                if rest == "":
                    # fallback: treat all columns as items (no ID)
                    items = [i.strip() for i in row if i.strip()]
                else:
                    if ',' in rest:
                        items = [i.strip() for i in rest.split(',') if i.strip()]
                    else:
                        items = [i.strip() for i in rest.split() if i.strip()]
            if items:
                transactions.append(items)
    return transactions

# -------------------------
# Convert MIN_SUPPORT to absolute count
# -------------------------
def min_support_count(min_support, num_transactions):
    if isinstance(min_support, float) and 0 < min_support <= 1:
        return math.ceil(min_support * num_transactions)
    if isinstance(min_support, int) and min_support >= 1:
        return min_support
    if isinstance(min_support, str) and min_support.endswith('%'):
        p = float(min_support.strip('%')) / 100.0
        return math.ceil(p * num_transactions)
    raise ValueError("MIN_SUPPORT must be int>=1 or float (0..1) or percent string like
'40%'")
```

```
# -------------------------
# Build FP-tree
# -------------------------
def build_fp_tree(transactions, min_support):
    # 1. First pass: count item frequencies
    item_counts = defaultdict(int)
    for t in transactions:
        for item in t:
            item_counts[item] += 1

    # 2. Remove items below min_support
    freq_items = {it: cnt for it, cnt in item_counts.items() if cnt >= min_support}
    if not freq_items:
        return None, None

    # 3. Header table: item -> [support_count, head_of_node_link]
    header_table = {it: [cnt, None] for it, cnt in freq_items.items()}

    # 4. Create root of FP-tree
    root = FPNode(None, 1, None)

    # 5. Insert transactions (filter & sort by descending frequency)
    for t in transactions:
        # keep only frequent items in transaction and sort by freq desc, break ties by item
name
        ordered_items = [i for i in sorted(t, key=lambda x: (-freq_items.get(x, 0), x)) if i in
freq_items]
        current_node = root
        for item in ordered_items:
            # if child exists, increment; else create
            if item in current_node.children:
                current_node.children[item].increment(1)
                current_node = current_node.children[item]
            else:
                new_node = FPNode(item, 1, current_node)
                current_node.children[item] = new_node
                # update header table (node link)
                head = header_table[item][1]
                if head is None:
                    header_table[item][1] = new_node
                else:
                    # follow node_link to append
                    while head.node_link is not None:
                        head = head.node_link
                    head.node_link = new_node
                current_node = new_node

    return root, header_table
```

```
# ------------------------
# Helpers for mining
# ------------------------
def ascend_tree(node):
    """
    Ascend from node up to root, returning prefix path (excluding the node itself).
    Returned list is in order: closest-parent, ..., farthest
    """
    path = []
    while node is not None and node.parent is not None and node.parent.item_name is not
None:
        node = node.parent
        path.append(node.item_name)
    return path

def find_prefix_paths(base_item, header_table):
    """
    For a given base_item, follow its node links and collect prefix paths with counts.
    Returns dict: {tuple(prefix_path): count}
    """
    paths = {}
    node = header_table[base_item][1]
    while node is not None:
        prefix = ascend_tree(node)
        if prefix:
            paths[tuple(prefix)] = paths.get(tuple(prefix), 0) + node.count
        node = node.node_link
    return paths

# ------------------------
# Mining the FP-tree (recursive)
# ------------------------
def mine_tree(header_table, min_support, prefix, freq_itemsets):
    """
    header_table: current header table (item -> [support, node])
    prefix: set of items already in the base pattern
    freq_itemsets: dict to accumulate results {frozenset(itemset): support}
    """
    # Process items in header_table in order of increasing support (as in original algorithm)
    sorted_items = sorted(header_table.items(), key=lambda x: x[1][0])
    for item, (support, _) in sorted_items:
        new_pattern = prefix.copy()
        new_pattern.add(item)
        freq_itemsets[frozenset(new_pattern)] = support

        # Build conditional pattern base (prefix paths leading to item)
        conditional_patterns = find_prefix_paths(item, header_table)
        conditional_transactions = []
```

```
    for path, count in conditional_patterns.items():
      # Expand by count (each path appears 'count' times)
      for _ in range(count):
        conditional_transactions.append(list(path))

    # Build conditional FP-tree
    cond_root, cond_header = build_fp_tree(conditional_transactions, min_support)
    if cond_header is not None:
      # recursively mine conditional FP-tree
      mine_tree(cond_header, min_support, new_pattern, freq_itemsets)

# -------------------------
# Main
# -------------------------
def main():
  transactions = load_transactions_from_csv(INPUT_CSV)
  if not transactions:
    print("No transactions found in", INPUT_CSV)
    return
  N = len(transactions)
  ms_count = min_support_count(MIN_SUPPORT, N)
  print(f"Loaded {N} transactions. Using minimum support = {ms_count} (MIN_SUPPORT={MIN_SUPPORT})\n")

  root, header = build_fp_tree(transactions, ms_count)
  if header is None:
    print("No frequent items found for the given minimum support.")
    return

  freq_itemsets = {}
  mine_tree(header, ms_count, set(), freq_itemsets)

  # Pretty print: sort by support desc, then by itemset size desc
  results = sorted(freq_itemsets.items(), key=lambda x: (-x[1], -len(x[0]), sorted(list(x[0]))))
  print("Frequent itemsets (itemset : support):\n")
  for itemset, support in results:
    items = sorted(list(itemset))
    print(f"{items} : {support}")

if __name__ == "__main__":
  main()
```

Output:

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

```
(base) C:\Users\Pranita Kumbhar>python decision_tree_experiment.py
Loaded 10 transactions. Using minimum support = 2 (MIN_SUPPORT=2)

Frequent itemsets (itemset : support):

['diapers'] : 5
['eggs'] : 4
['milk'] : 4
['beer', 'diapers'] : 3
['beer'] : 3
['cola'] : 3
['cola', 'diapers'] : 2
['diapers', 'milk'] : 2

(base) C:\Users\Pranita Kumbhar>
```

Conclusion

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

➢

How FP-Growth Manages and Mines Itemsets of Varying Lengths

1. Problem background

In transactional databases (like market-basket data), itemsets can be of:

- Length 1 (single items like milk)
- Length 2 (pairs like {bread, milk})
- Length 3 or more (combinations like {bread, milk, diapers})

Mining all these efficiently is difficult because the number of possible combinations grows exponentially with the number of items.

2. FP-Growth approach

FP-Growth (Frequent Pattern Growth) avoids generating and testing all candidate itemsets (like Apriori does). Instead, it uses a divide-and-conquer strategy with a compressed FP-Tree structure.

3. Step-by-step explanation

A. Building the FP-Tree

- First, FP-Growth scans the database to count item frequencies.
- Items below minimum support are discarded.
- Transactions are then re-ordered by descending frequency and inserted into a tree (FP-Tree).
- Shared prefixes between transactions are merged, so the tree is compact.

◇ Example:
[bread, milk, diapers] and [bread, milk, cola] will share the prefix [bread, milk] in the FP-tree.

B. Managing itemsets of different lengths

- Each node in the FP-tree represents an item and its frequency.
- The header table links all nodes of the same item for quick access.
- By following these links, FP-Growth can find:
  - Length-1 itemsets directly from item counts.
  - Length-2, 3, … itemsets by combining items along paths.

So instead of explicitly generating "all subsets," FP-Growth implicitly explores longer itemsets using the compressed paths.

C. Mining with Conditional FP-Trees

- For each item, FP-Growth constructs a conditional pattern base (the set of prefix paths ending with that item).
- From this, it builds a conditional FP-tree, which represents transactions containing that item.
- This process recursively discovers:
  - All frequent pairs (2-itemsets)
  - Triplets (3-itemsets)
  - And so on, up to the maximum length present in the data.

◇ Example:
If milk occurs with bread frequently, FP-Growth builds a conditional FP-tree for milk and finds {bread, milk}. If diapers also appear in that conditional tree, it finds {bread, milk, diapers}.

4. Key point

FP-Growth manages varying itemset lengths naturally by:

- Using the FP-tree to compress transactions.
- Traversing conditional trees recursively.
- Expanding frequent patterns step by step without generating unnecessary candidates.

This allows FP-Growth to mine short, medium, and long frequent itemsets efficiently, even in very large datasets.

5. One-line conclusion for your experiment

FP-Growth handles itemsets of varying lengths by recursively building conditional FP-trees from compressed transaction data, allowing efficient discovery of frequent patterns without generating redundant candidates.