**A black background with blue text

Description automatically generated**

**Summer 2023**

**Project Title: Implementation and Performance Analysis of FP-Growth Algorithms.**

**Submitted to**

**Jesan Ahammed Ovi**

**Senior Lecturer**

**Department of Computer Science & Engineering**

**Submitted by**

| Md Shadman Shakib | 2019-2-60-026 |
| --- | --- |
| Md Shorif Hossain. | 2019-2-60-039 |
| S M Arafat Rahman | 2019-2-60-094 |
| JB Sohan | 2019-2-60-0 |
| Md. Redwan Ahmed | 2019-1-60-249 |

**Problem Statement**

Frequent itemset mining is a fundamental task in data mining and has various applications such as market basket analysis, recommendation systems, and more. The FP-Growth algorithm is a popular and efficient approach for discovering frequent itemsets from large datasets. This project aims to explore the FP-Growth algorithm, implement it, and comprehensively analyze its performance in terms of time complexity and memory usage.

**System Requirement**

Apple M2 Chip

8-core CPU with 4 performance cores and 4 efficiency cores

10-core GPU

16-core Neural Engine

100GB/s memory bandwidth

Operating System: MacOS

IDE: Visual Studio Code

Apple M2

**Implementation:**

We have implemented the FP growth algorithm  algorithm. The important functions of the algorithm which are used is discussed below:

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | **class** **Tree**:  **def** \_\_init\_\_(self, transactions):  self.root = self.Node(**None**, **None**)  self.nodes = {}  **for** t **in** transactions:  self.add\_nodes(t)  **class** **Node**:  **def** \_\_init\_\_(self, value, parent): *#I5 = [I5,I5]*  self.freq = 1  self.value = value  self.parent = parent  self.childs = []  **def** \_\_repr\_\_(self):  **return** self.value **if** self.value **else** 'Root'  **def** \_\_eq\_\_(self, other):  **if** isinstance(other, str):  **return** self.value == other  **return** **False**  **def** \_\_contains\_\_(self, item):  **for** child **in** self.childs:  **if** item == child.value:  **return** **True**  **return** **False**  **def** \_\_hash\_\_(self):  **return** hash(self.value + str(self.freq))  **def** add\_child(self, child):  self.childs.append(child)  **def** add\_nodes(self, transaction):  curr = self.root  **for** item **in** transaction: *# ['I2', 'I1', 'I4']*  **if** item **not** **in** curr.childs:  node = self.Node(item, curr)  **if** item **in** self.nodes:  self.nodes[item].append(node)  **else**:  self.nodes[item] = [node]  curr.childs.append(node)  curr = node  **else**:  **for** child **in** curr.childs:  **if** item == child:  curr = child  curr.freq += 1  *# print('in curr', curr.value)*  **break** |
| --- | --- |

The **Tree class** represents the tree structure, and it's initialized with a list of transactions.  
The **Node class** represents a node in the tree, each corresponding to an item in a transaction. The **\_\_init\_\_** method in the **Node class** initializes a node with a value (an item), a parent node, a frequency count (initialized to 1), and an empty list of child nodes. The **add\_child** method in the Node class is used to add a child node to the current node.The **add\_nodes** method in the **Tree class** is used to add transactions to the tree. It iterates through the items in a transaction and adds nodes for each item, incrementing the frequency count if the item already exists as a child of the current node.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | **def** generate\_patterns(self, ordered\_items, min\_sup):  freq\_patterns = [([k], v) **for** k, v **in** ordered\_items.items()]  ordered\_items = list(reversed(ordered\_items.keys()))[:-1]  print(ordered\_items)    **for** pattern **in** ordered\_items:  instances = self.nodes[pattern]  paths = []  **for** instance **in** instances:  path = self.get\_path(instance)[1:]  paths.append((path, instance.freq))  freqs\_obj, freqs\_values = {}, {}  **for** tups **in** paths:  path, freq = tups[0], tups[1]  **for** x **in** path:  freqs\_obj[x] = freqs\_obj.get(x, 0) + freq  freqs\_values[x.value] = freqs\_values.get(x.value,0) + freq  freqs\_values = {k: v **for** k, v **in** freqs\_values.items() **if** v >= min\_sup}  curr\_patterns = []  **for** k, v **in** freqs\_obj.items():  **if** **not** k.value **in** freqs\_values:  **continue**  curr\_patterns.append(([pattern, k], v))  freq\_patterns.append(([pattern, k.value], freqs\_values[k.value]))  **for** conditional **in** curr\_patterns:  self.helper(conditional, freq\_patterns, min\_sup)  print()  freq\_patterns = sorted(freq\_patterns, key=**lambda** x: len(x[0]))  unique\_tuples = []  seen\_tuples = set()  **for** pattern **in** freq\_patterns:  key = (frozenset(pattern[0]), pattern[1])  **if** key **not** **in** seen\_tuples:  unique\_tuples.append(pattern)  seen\_tuples.add(key)  **return** unique\_tuples |
| --- | --- |
|  |  |

The **generate\_patterns** method in your code appears to be used for generating frequent itemset patterns from the transaction data stored in your tree structure. This method takes a list of ordered items (likely items ordered by their frequency) and a minimum support threshold (min\_sup) as input parameters. It then returns a list of frequent itemset patterns that meet the minimum support threshold.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | **def** helper(self, conditional, freq\_patterns, min\_sup):  cond, freq = conditional *#([I4, I3], 1)*  **if** **not** cond[-1].value:  **return**  path = self.get\_path(cond[-1])[1:]  freqs\_obj, freqs\_values = {}, {}  **for** x **in** path:  freqs\_obj[x] = freqs\_obj.get(x, 0) + freq  freqs\_values[x.value] = freqs\_values.get(x.value, 0) + freq  freqs\_values = {k: v **for** k, v **in** freqs\_values.items() **if** v >= min\_sup}  curr\_patterns = []  **for** k, v **in** freqs\_obj.items():  **if** **not** k.value **in** freqs\_values:  **continue**  curr\_patterns.append((cond + [k], v))  freq\_patterns.append((cond + [k.value], freqs\_values[k.value]))  **for** conditional **in** curr\_patterns:  self.helper(conditional, freq\_patterns, min\_sup)  **def** get\_path(self, instance):  path = []  **while** instance.parent:  instance = instance.parent  path.append(instance)  **return** path[::-1] |
| --- | --- |

The **helper method** is a recursive function used to assist in generating frequent itemset patterns based on conditional patterns. The purpose of the helper method is to systematically explore the tree structure to generate frequent itemset patterns by extending conditional patterns. It accumulates frequencies along threshold.

Together, these methods are used in the process of generating frequent itemset patterns from the transaction data stored in the tree structure while considering the minimum support threshold.

**Performance analysis:**

The performance of the FP-Growth and Apriori algorithms can vary depending on several factors, including dataset characteristics, support and confidence thresholds, and implementation details. Here, we'll discuss the key performance considerations for both algorithms:

**Time Complexity:**

**Apriori**: This algorithm has a worst-case time complexity of O(2^N), where N is the number of items in the dataset. This means that as the dataset grows, the runtime of Apriori can become impractical.

**FP**-**Growth**: FP-Growth is generally more efficient than Apriori, especially on large datasets. It has a complexity of O(N^2) for building the FP-Tree and O(N) for mining frequent itemsets, where N is the number of transactions.

**Execution time:** We used mushroom.dat dataset for comparison between FP Growth algorithm and Apriori algorithm.

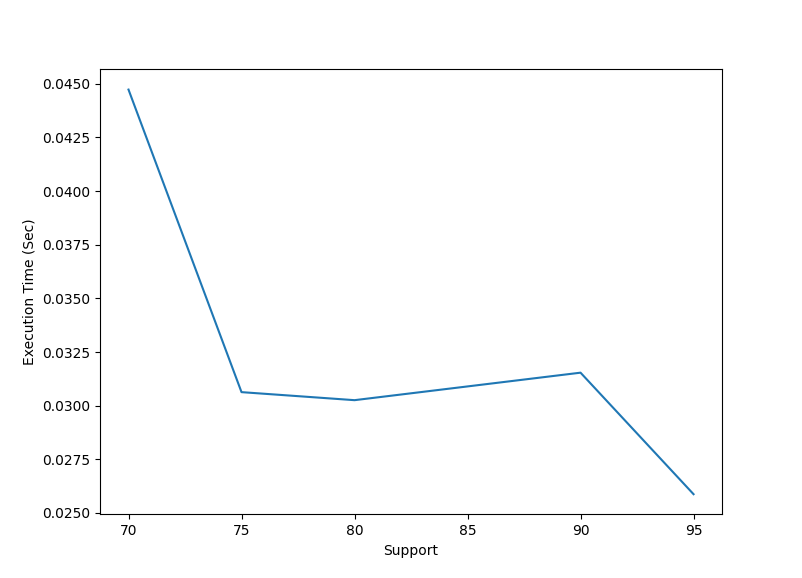
****

Figure1. Execution time of FP Growth Algorithm with different threshold

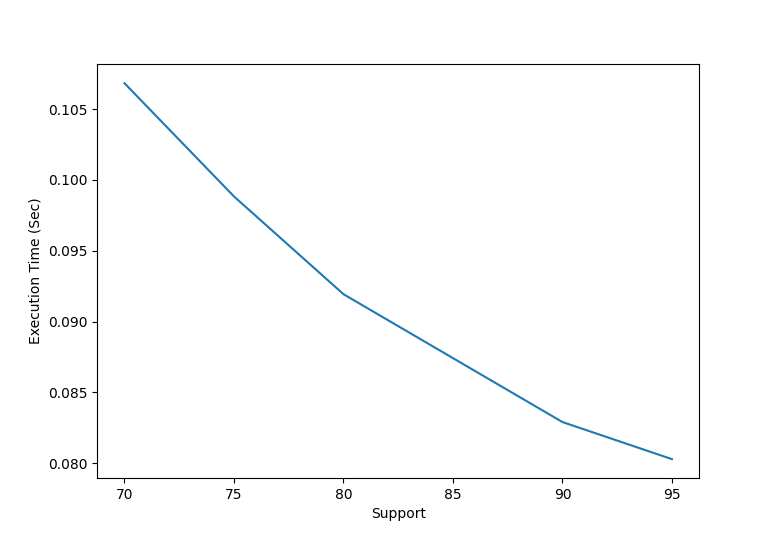
****

Figure2. Execution time of Apriori Algorithm with different threshold

**Memory Usage:**

**Experimental Results:**

We have used two dataset too see the performance of the algorithm . We used mushroom.dat and retail.dat dataset yo see our algorithm is working properly or not .

**Mushroom Dataset:** In mushroom dataset we are getting 1-Itemset which have frequency 5 but in itemset-2,3 we are getting the highest frequency which is 10 **.**

**A graph with blue bars

Description automatically generated with medium confidence**

Figure3. Mushroom dataset output result

**Retail dataset:**In retail dataset we are getting 1-Itemset which have frequency 20 but in itemset-2 we are getting the highest frequency which is 22 and 4-itemset having the lowest frequency .

**A graph with blue squares

Description automatically generated**

Figure4. Retail dataset output result