Comparing the Frequent Pattern Growth Algorithm with the Apriori Algorithm: A Comparative Analysis.

This technical report is a comparative analysis between two algorithms: the Frequent Pattern Growth Algorithm and the Apriori Algorithm. It examines their general approaches, computational efficiency, complexity, limitations, and practical applications.

I certify that all material in this analysis which is not my own work has been identified.

Christopher Man

December 2023

Contents

1 Introduction **2**

2 Main Principles **2**
2.1 General Approaches 2
2.2 Pseudo Code 3

3 Memory and Computational Efficiency **4**

4 Algorithm Complexity **4**

5 Limitations of FP-Growth **5**

6 Practical Applications **5**
6.1 Modern Library Speed Comparison 6

7 Conclusion **6**

1 Introduction

Algorithms form the core of computing, empowering computers to execute tasks with efficiency, precision, and uniformity across diverse domains, thereby holding a pivotal position in our contemporary technological world. [1] This technical report delves into frequent itemset mining, exploring the Frequent Pattern Growth *(FPG)* algorithm and comparing it to the Apriori Algorithm. It discusses the general methodologies of each algorithm, their data traversal, algorithmic intricacies, and considerations regarding memory usage and computational efficacy. Furthermore, it provides an analysis of practical outcomes derived from both algorithms and their respective real-world applications.

2 Main Principles

2.1 General Approaches

Before the adoption of FPG, most of the data science literature relied on the Apriori-like approach. This method involved candidate generation and subsequent testing. However, subsequent research shed light on significant expenses associated with this algorithm, especially when dealing with numerous and lengthy patterns. [2]

Considering the evident fundamental constraints of the Apriori algorithm, a newly proposed method called *"frequent pattern growth (FPG)"* was developed. This innovative approach emphasises mining frequent patterns without dependence on candidate generation. Instead, it adopts a divide-and-conquer strategy, projecting and partitioning databases based on presently discovered frequent patterns and extending these patterns to longer ones within the projected database. [3]

The Apriori algorithm generally uses the "join" and "prune" methods. The join method scans through the dataset presented to identify common items within the dataset (items of length 1). Then it creates candidate itemsets of length (k + 1) by joining pairs of frequent items of length k. The Prune method checks the frequency of occurrence of the candidate itemsets against a minimum threshold. Any candidate itemsets that fall below this threshold are removed ("pruned") from further consideration. The pruning stage assists in reducing the search space and focuses on itemsets likely to be the most frequent. Multiple iterations of the "join" and "prune" methods allow the algorithm to gradually discover frequent pairs of data. [4]

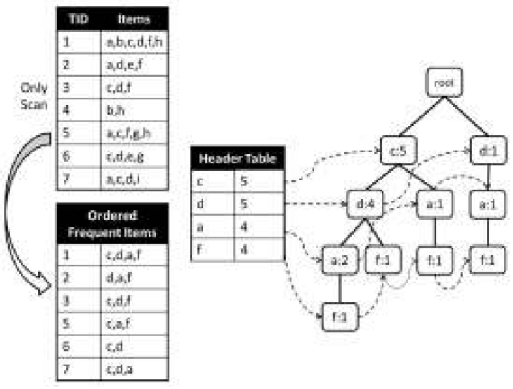


Figure 1: FPG Example Diagram [5]

The FPG algorithm's first step consists of constructing a frequent pattern tree (FP-tree). Scanning the dataset through a couple of iterations, generating a frequency table to identify the support of each item in the dataset. A FP-tree is built upon the frequency table, linking each item depending on their occurrences *(see figure 1)*. In the second step the FP-tree is recursively mined to extract frequent itemsets by the "pattern fragment growth" process. This process efficiently generates frequent item sets by recursively combining frequent patterns found in the FP-tree. [6]

2.2 Pseudo Code

Below is a representation of how the code could be manually implemented using pseudo code. *(see figure 2)*

```
#Pseudo code implementation:

class Node:
    init (self, item, freq, parent):
        initialize all variables to itself
        self.item = item
        ...
        ...
        ...

def create_header_table(dataset, minimum_support):
    header_table = {}
    for transaction in dataset
        for item in transaction:
                header_table [item] = header table get (item, 0) + 1
    header_table = {k: v for k, v in header_table.items() if v >= min_support}

def update tree (item, node, header_table):
    if item in node.child:
        node.child[item].frequency +=1
    else
        new node = node(item, 1, node)
        node.child[item] = new node

        if header_table[item][1] is none then
            header_table[item][1] is new node
        else:
            update_list(header_table[item][1], new node)

    return node.children[item]

def update_list(head, node):
    while head next is not NULL:
        head is head.next
    head.next is node

def fp_growth(dataset, min_support):
    header table is create_header_table (dataset, min_support)
    if length of header_table is 0:
        return nothing

    for item in header_table
    header_table at item is [header_tablep[item], None]

    root = node('Null', 1, None)
    for transcation in the dataset
        sorted_items = sorted (transactions, key: header_table.get(k, 0), reverseTrue)
        current node is root node
        for item in sorted items:
            current node is update tree (item, current node, header_table)

    frequent patterns array = []
    for item in header_table:
        base_pattern is empty array
        conditionalJ_tree is empty array
        node = header_table[item][1]
            while node is not empty:
                conditional_tree.append(node)
                node is node.parent
            for ct_node in the conditional tree:
                base pattern.append (ct_node.item)

            if length of base pattern > 1:
                append to freq patterns array
    return freq patterns
```

Figure 2: Pseudo Code

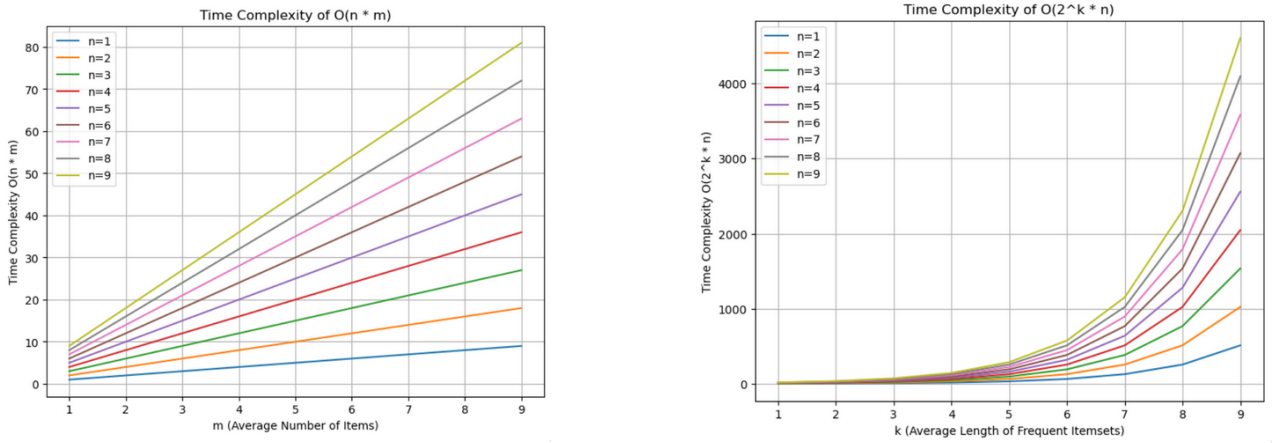
3 Memory and Computational Efficiency

Comparing the Apriori and FPG algorithms we can see some notable differences between the two. In terms of Memory and computational efficiency, the Apriori algorithm in many cases becomes very memory intensive and yields high computational resource cost. The reason as to why is due to Apriori having to repeatedly scan the dataset to generate candidate itemsets during each iteration. This is eloquently illustrated by Rathee et al "The most computationally expensive task is the generation of candidate sets having all possible pairs for singleton frequent items and comparing each pair with every transaction record." [7].

In contrast to Apriori, the FP growth algorithm is significantly memory and computationally efficient. The main contributing factor for this is due to the algorithm creating a frequent pattern tree (FP-tree). This FP-tree compresses the dataset as a whole because of the frequency table, meaning there is a reduction in overhead computational cost. Applying this to a larger scale, as the dataset increases in size the FPG algorithm will see an exponential decrease in computation time in comparison to Apriori due to the FP-tree's compression. [8]

4 Algorithm Complexity

In any algorithm it is important to analyse the number of passes over the data that the algorithm skims through. We can identify an algorithm's complexity by measuring factors that affect its recursive ability on the data. Due to the logic of Apriori the algorithm in many cases has to iterate over the dataset multiple times to find relevant frequent itemsets. Because of this the amount of passes over data will increase as the dataset increases. Illustrating that as the dataset grows so will the computational time and complexity. However, within the FP-growth algorithm there is only a required 2 passes over the dataset. The initial pass for the construction of the FP-tree and subsequent frequent table, and the second for the mining of frequent itemsets. What this means is that as the dataset grows the algorithm's complexity stays the same. The complexity of Apriori depends on the number of itemsets and their dataset size, the FP-growth algorithm has a better worst-case time complexity compared to Apriori in all scenarios. [9]



(a) FPG Time Complexity

(b) Apriori Time Complexity

Figure 3: Graphed Time Complexity

The Apriori algorithm generally has a time complexity of O($2\hat{k}$ * n), where 'k' is the average length of frequent itemsets and 'n' is the number of transactions. Where the FP-growth algorithm's worst-case time complexity is O(n * m), where 'n' is the number of transactions and 'm' is the average number of items in a. In practical scenarios, it can be even better than O(n * m). After graphing the time complexity *(see figure 3)* the Apriori algorithm may be more performant if the dataset is smaller, however once the length of the dataset increases there is an exponential increase in time complexity. The FP-growth algorithm due to its design with the FP-tree has a more linear complexity graph,

showing a dominant performance increase in comparison to Apriori as the dataset increases in total size.

5 Limitations of FP-Growth

Uniquely there are situations where the Apriori algorithm may be a better suit of application in comparison to FPG algorithm. If the data is sparse, Apriori would perform better as the itemsets that are infrequent would be more efficiently identified due to its reiterative nature. Again this is also mimic'd if the dataset is very small, its ability to "prune" the search space removes the itemsets early on in iteration based upon the support threshold. Meaning in comparison to FP-growth utilising the feature tree may not have an advantage over small datasets. [6]

Another niche area in which Apriori might be of better application can be within a machine with limited memory. Due to the FPG algorithm having to create a FP-tree this will have to allocate a certain amount of memory on construction. However with the Apriori algorithm only the candidate itemset will be updated on each iteration. Hence, for machines or environments where memory is very limiting and sparse apriori in this unique space may be the only option. [10]

Another environment where the Apriori algorithm may have potential benefits is within its ease of implementation and easy interpretation. Due to Apriori having a very simple iterative nature, it can be very easy to follow what steps are taken in order to conduct association mining. This implies that it's also straightforward to comprehend and interpret within code. Consequently, if there are specific parameters that require adjustments or modifications, it will be significantly easier to manage compared to the FP-growth algorithm. The FP-growth algorithm acts as a black box with larger datasets due to its very efficient time complexity on very large datasets.

6 Practical Applications

It is conclusive that the FP-growth algorithm is dominant in time complexity and computational efficiency. In a practical application using Python *(see figure 4)* this algorithm can be used in many real life scenarios. The first real life application of this algorithm can be seen in the retail industry regarding market and customer analysis. Utility in helping identify the purchasing trends of customers walking into a store and or could help with inventory management etc. Another explanation could pertain to the medical and healthcare sector, such as for patient health trends and or symptoms linked with medications. Regardless of whatever the situation is, it is apparent that the FP-Growth Algorithm has a wide range of applications within any dataset that has any type of data link whatsoever. An extremely powerful algorithm that could be used to efficiently present links between frequent data.

```
[23]: rules = association_rules(frequent_itemsets, metric = "confidence", min_threshold = 0.8)
      rules
```

[23]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zhangs_metric
0	(Potatoes)	(Bread)	1.000000	0.833333	0.833333	0.833333	1.0	0.000000	1.0	0.0
1	(Bread)	(Potatoes)	0.833333	1.000000	0.833333	1.000000	1.0	0.000000	inf	0.0
2	(Milk)	(Cheese)	0.666667	0.833333	0.666667	1.000000	1.2	0.111111	inf	0.5
3	(Potatoes)	(Cheese)	1.000000	0.833333	0.833333	0.833333	1.0	0.000000	1.0	0.0
4	(Cheese)	(Potatoes)	0.833333	1.000000	0.833333	1.000000	1.0	0.000000	inf	0.0
5	(Milk)	(Potatoes)	0.666667	1.000000	0.666667	1.000000	1.0	0.000000	inf	0.0
6	(Onions)	(Potatoes)	0.666667	1.000000	0.666667	1.000000	1.0	0.000000	inf	0.0
7	(Bread, Cheese)	(Potatoes)	0.666667	1.000000	0.666667	1.000000	1.0	0.000000	inf	0.0
8	(Potatoes, Milk)	(Cheese)	0.666667	0.833333	0.666667	1.000000	1.2	0.111111	inf	0.5
9	(Cheese, Milk)	(Potatoes)	0.666667	1.000000	0.666667	1.000000	1.0	0.000000	inf	0.0
10	(Milk)	(Potatoes, Cheese)	0.666667	0.833333	0.666667	1.000000	1.2	0.111111	inf	0.5

```
[25]: from mlxtend.frequent_patterns import apriori
      %timeit apriori(df, min_support = 0.6)
```

```
1.56 ms ± 84.6 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
[26]: %timeit fpgrowth(df, min_support = 0.6)
```

```
534 µs ± 12.6 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Figure 4: Python Speed Tests

6.1 Modern Library Speed Comparison

Using the latest libraries available such as "mlxtend" *(see figure 4)* [11], a performance comparison test was conducted between the FP-Growth algorithm and the Apriori Algorithm. A randomly generated dataset was utilised for this purpose. In the figure you can clearly see a massive difference of computational speed: the percentage difference from 1.56 milliseconds to 54 microseconds is approximately 96.538%. So in a practical test the FP-growth algorithm is dominant in terms of speed and efficiency.

7 Conclusion

In summary, algorithms like FP-Growth significantly influence the analysis of data in our world, facilitating the identification of frequent patterns within extensive datasets. Derived and enhanced from the Apriori algorithm, FP-Growth showcases a robust engineering approach by evolving from existing algorithms to propose a more efficient method of computation.

References

[1] D. E. Knuth, "Algorithms in modern mathematics and computer science," *Algorithms in Modern Mathematics and Computer Science*, pp. 82–99, 1981.

[2] J. Han and J. Pei, "Mining frequent patterns by pattern-growth: methodology and implications," *ACM SIGKDD explorations newsletter*, vol. 2, no. 2, p. 14–20, 2000.

[3] K.-C. Lin, I.-E. Liao, and Z.-S. Chen, "An improved frequent pattern growth method for mining association rules," *Expert Systems with Applications*, vol. 38, pp. 5154–5161, 05 2011.

[4] D. Magdalene, Angeline, "Association rule generation for student performance analysis using apriori algorithm," *The SIJ Transactions on Computer Science Engineering its Applications (CSEA)*, vol. 1, no. 1, p. 12–16, 2013.

[5] C. Leung, "Fp-tree used in the fp-growth algorithm," 09 2011.

[6] S. A. Ahmed and B. Nath, "Identification of adverse disease agents and risk analysis using frequent pattern mining," *Information Sciences*, vol. 576, pp. 609–641, 10 2021.

[7] *R-apriori: An efficient apriori based algorithm on spark*, Association for Computing Machinery, 2015.

[8] *The benefits of using prefix tree data structure in multi-level frequent pattern mining*, 2007.

[9] *An empirical analysis and comparison of apriori and FP- growth algorithm for frequent pattern mining*, 2014.

[10] S. Nasreen, M. A. Azam, K. Shehzad, U. Naeem, and M. A. Ghazanfar, "Frequent pattern mining algorithms for finding associated frequent patterns for data streams: A survey," *Procedia Computer Science*, vol. 37, pp. 109–116, 2014.

[11] S. Raschka, "mlxtend," 2014.