Performance FP-growth Algorithms for Chess and Mushroom Datasets

Mubassira Khan
Student ID: 2020-1-60-054
Department of CSE
East West University
Dhaka, Bangladesh

Iffat Tasnim
Student ID: 2020-2-60-004
Department of CSE
East West University
Dhaka, Bangladesh

Mushfiqul Islam
Student ID: 2020-1-60-261
Department of CSE
East West University
Dhaka, Bangladesh

Submitted To

Amit Mandal

Department of Computer Science & Engineering

East West University

Dhaka, Bangladesh

Introduction

Recently, the amount of data in databases has increased quickly, necessitating the employment of sophisticated approaches to effectively extract meaningful information. Large transactional datasets, found in major companies, pose challenges for both humans and machines when trying to analyze them.

Data mining is a valuable technique for extracting meaningful insights from massive databases, particularly in fields like marketing and medicine. Frequent itemset mining, which identifies recurring patterns in data, is a crucial aspect of data mining. However, traditional algorithms for this task often struggle to determine an appropriate minimum support threshold, which is crucial for pattern discovery.

Among the various algorithms, FP-Growth is a commonly used method for mining frequent patterns. FP-Growth is more efficient and avoids candidate generation through a "divide and conquer" strategy.

In this project, we compare used FP-Growth, on two different datasets (Chess and Mushroom) to determine which algorithm performs better under various conditions.

Data Preprocessing:

Here, I have utilized the ‘mushroom.dat’ and ‘chess.dat’ datasets to evaluate the performance of the Fp growth algorithm. The data were cleaned already. So, we did not need any preprocessing technique for cleaning the dataset.

Implementation of FP-Growth Algorithm:

Here, for implementing the FP-Growth Algorithm from sketch, we used Python language.
For IDE we used Google Colabrotory. First, we mounted our drive and then did the rest of the code.

Here is the explanation of the classes and functions that we used in the code:

TreeNode:

- This is a class representing a node in the FP-Tree. It contains attributes like name (item name), count (frequency count of the itemset), nodeLink (link to the next node with the same name), parent (parent node), and children (child nodes).

create_FPTree:

- This function creates the FP-Tree from the dataset.
- It first creates a header table (HeaderTable) to keep track of the frequency of each item.

- It then scans the dataset and filters out items that do not meet the minimum support threshold.
- It builds the FP-Tree using the filtered items.

Node Class:

Represents a node in the FP-growth tree.
Each node has a pattern, frequency, parent, and a list of children.

updateTree:

- This function is used to update the FP-Tree with a new transaction.
- It recursively traverses the tree and increments the counters of existing nodes or creates new nodes if necessary.

update_NodeLink:

- This function updates the link of a node in the FP-Tree.

FPTree_uptransveral:

- This is a recursive function used in conditional pattern base creation. It traverses the tree upwards from a leaf node.

find_prefix_path:

- This function finds the conditional pattern base for a given item.

Mine_Tree:

- This is the main function that mines frequent itemsets from the FP-Tree.
- It generates frequent itemsets recursively by adding one item at a time.
- It also calls find_prefix_path to get conditional pattern bases.

print_tree2 and write_tree_to_file:

- These functions are used to print or write the FP-Tree structure to a file.

write_tree:

- This function is used to specify the filename for writing the FP-Tree structure.

print_support_counts:

- This function prints the support count for each frequent itemset.

Loading Data and Running FP-Growth:

- The code includes loading data from a file (assuming it's a dataset in a specific format).
- It then sets the minimum support threshold (min_Support) based on a specified percentage of the dataset length.
- It runs the FP-Growth algorithm on the data and measures execution time.

Usage:

- An instance of the FPgrowth class is created with a path to the dataset.
- The mine method is called to mine frequent patterns based on a specified threshold.

- The results can be displayed using show_frequent_patterns and count_frequent_pattern methods.

Why choosing these two datasets?

The choice of the Mushroom and Chess datasets for comparison in this project is based on their diversity in data types (categorical and numerical), varying dataset sizes, and potential real-world applications. The Mushroom dataset is known for its simplicity and relevance to mycology, making it suitable for benchmarking and insights. Meanwhile, the Chess dataset represents a different domain (chess analysis and AI), providing a comprehensive evaluation of the algorithms across different data characteristics and sizes. In Fp-growth it scans the database only twice so takes a short time to be executed. It does not generate candidates of false patterns so FP-growth requires less memory. For this reason, it reduces search costs.

Memory Usage Analysis:

As we see before, the total space complexity of the code is approximately *O(n * m)*, dominated by the space used for the FP Tree and HeaderTable, where *'n'* is the number of transactions, and *'m'* is the average number of items per transaction.

We can say that the FP-Growth algorithm is generally more efficient .In Apriori, candidate itemsets can consume substantial memory, particularly for large datasets. FP-Growth, employs a compact FP Tree data structure, significantly reducing the need for candidate itemset storage. This makes FP-Growth a more memory-efficient choice, particularly in scenarios with large datasets or high itemset cardinality, as it minimizes memory consumption while still efficiently mining frequent itemsets.

Performance Optimization:

1. Frequent Itemset Generation:

Bottleneck: Generating frequent itemsets can be slow, especially for large datasets.

Optimization: Implement pruning techniques like the Apriori property to reduce the number of candidates generated.

2. Tree Construction:

Bottleneck: Building the FP Tree can be memory-intensive, especially for large datasets.

Optimization: Use memory-efficient data structures to represent the tree nodes and minimize memory overhead.

3. Conditional FP Tree Construction:

Bottleneck: Creating conditional FP Trees during mining can be time-consuming.
Optimization: Implement path compression techniques to reduce tree size and enhance mining speed.

4. Memory Usage:
Bottleneck: Memory consumption can be high due to storing transactions and conditional pattern bases.
Optimization: Use compact data structures for storing transactions and conditional pattern bases to reduce memory usage.

5. Sorting
Bottleneck: Sorting items within transactions can be time-consuming.
Optimization: Implement efficient sorting algorithms or use data structures that minimize the need for sorting.

6. Parallel Processing:
Bottleneck: FP-Growth is inherently single-threaded.
Optimization: Consider parallelizing certain parts of the algorithm, such as frequent itemset generation or tree construction, to leverage multi-core processors for improved speed.

7. Memory Management:
Bottleneck: Inefficient memory management can lead to memory leaks or excessive memory fragmentation.
Optimization: Implement effective memory management practices, like releasing unused memory promptly and optimizing data structure sizes.

8. Data Pruning:
Bottleneck: Analyzing infrequent data may lead to inefficiencies.
Optimization: Prune or ignore infrequent data early in the process to reduce unnecessary computation.

Conclusion:

For frequent pattern mining in this project, we assessed the FP-Growth Algorithm performance. Our main goal was to compare the amount of time needed to produce the same collection of frequent patterns on the same dataset using threshold.