LINFO2364 - Project 1 - Group 48
Algorithms for frequent itemsets mining

Bourez Nicolas
UCLouvain
Ecole polytechnique de Louvain-la-Neuve
Louvain-la-Neuve, Belgium
28462000

Demaret Dimitri
UCLouvain
Ecole polytechnique de Louvain-la-Neuve
Louvain-la-Neuve, Belgium
38502000

***Abstract*—This report aims to compare two different algorithms for which the goal is to find the most frequent itemsets in a given dataset. The first algorithm is the apriori algorithm and has been implemented in three different ways. The second one is the ECLAT algorithm. Along this report, we will give a short description of the optimizations we implemented, we will justify our implementations, and finally compare these algorithms to see how they behave with different sizes of datasets.**

I. Introduction

Frequent itemset mining is a critical task in data mining with various applications. It involves identifying sets of items that occur together frequently in a database. Among the numerous algorithms devised to tackle this problem, the Apriori and ECLAT algorithms stand out due to their foundational approaches and widespread usage.

This report delves into a comparative analysis of these algorithms, specifically focusing on three distinct implementations of the Apriori algorithm—namely, the Dummy Apriori, the Smart Apriori, and the Apriori with trie data structure—and the ECLAT algorithm.

Through rigorous analysis and experimentation on multiple datasets, this report aims to provide a comprehensive comparison in order to understand the conditions under which each algorithm excels.

II. Description of the algorithms

A. Dummy Apriori

First, we started by implementing a dummy version of the Apriori algorithm. We used the naïve way of generating k-itemsets candidates by trying all possibilities of a combination of items of size k.

To count the support of each candidate, we pass through all the transactions, and for each transaction, we search if the set "candidate" is a subset of the transaction set. To do this, we use `frozenset` from Python and its defined function `.issubset`.

The `frozenset` in Python is an immutable variant of the standard set. For some operations, `frozenset` can offer better performance than a mutable set. For example, checking if an element is in a set (in or not in) can be done faster with a `frozenset` than with a set, because the internal structure of a `frozenset` can be optimized knowing the set will not change. Also, they are usable as dictionary keys, making them convenient to use to store frequent itemsets.

B. Smart Apriori

In this version, we changed the way of generating k-itemsets candidates. Starting with frequent itemsets of size $k-1$, we check if they have the same prefix (all the items except the last one) and if it is true, we merge them together. For example, the frequent itemsets $\{1,3,5\}$, $\{1,2,6\}$, $\{1,3,4\}$ will create only one candidate $\{1,3,4,5\}$ allowing us to eliminate combinations where we know "simpler" subset of it was already infrequent, implying the candidate set to be infrequent too.

C. Apriori with trie

Our trie implementation is designed to compactly store itemsets as paths from a root to a node, where each node represents an item from the itemset. This structure is particularly advantageous for the Apriori algorithm for several reasons:

- **Compact Storage:** Our trie structure allows for the sharing of common prefixes among itemsets, reducing the memory footprint.
- **Efficient Candidate Generation and Support counting:** It facilitates the generation of candidate itemsets and the counting process taking advantage of the tree format of a trie structure.

Implementation Steps with Our Trie: The integration of our trie in the Apriori algorithm unfolds as follows:

1) **Initialization:** Create the root of the trie. Initially, the trie is empty.
2) **Populating the Trie:** In the first pass over the dataset, insert each item of the transactions that is frequent alone into the trie as individual paths. This step uses the same process as before to find frequent 1-itemset.
3) **Generating k-Itemset Candidates:** To generate candidate itemsets of size k, we traverse the trie to find all leaves of depth $k-1$ with the function `get_nodes`

and we create new nodes of depth k with two `while` in the same way as on Figure 1.

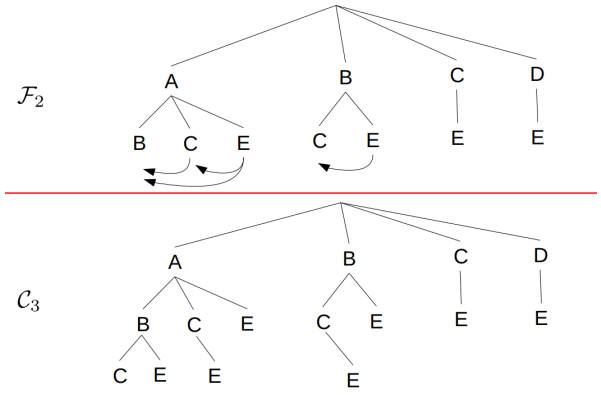

Fig. 1. smart k-itemsets generation using trie structure

4) **Counting Support:** For counting the support of each candidate, we leverage our `count_support` function. As we process each transaction, we traverse through the trie following the items present in the transaction. If a complete path representing a candidate itemset is successfully traversed, its support count is incremented.
5) **Pruning:** After support counts are determined, we prune infrequent itemsets using our `delete_nodes` method. This involves removing nodes that do not meet the minimum support threshold, thereby also eliminating all their descendant paths, which are not itemsets of size k anymore.
6) **Iterating:** Finally we increment k and go bak to step 3 while the `frequent_itemets` (of size k) is not empty.

D. ECLAT algorithm

ECLAT is an algorithm that performs a depth-first search through the space of possible itemsets to find the ones whose frequency is above the given threshold.

The algorithm first counts the occurrences of each item in the dataset and associates them with the corresponding transaction IDs. This operation is performed in $\mathcal{O}(n)$ with n the size of the dataset. This implementation choice is justified by the fact that to find the transactions of the itemsets at level l, we will have to take the intersection of the transactions of the $l-1$ level itemsets.

Starting from the frequent single items found in the first step, we can perform a depth-first search. It iteratively combines itemsets of increasing size and checks their frequency against the minimum threshold. To not search several times in the same part of the tree, we use a double `for(i in range) for j in range (i+1,...)` loop.

III. Comparison of the performances

A. Datasets analysis

First, we will present the 3 datasets for which we analyzed our algorithm. It will be useful to fully understand which upgrades work better over which kind of dataset.

Indeed, we observe in Figure 2 and Figure 3 that the datasets `retail.dat`, `accidents.dat` and `chess.dat` are very different. The "retail" dataset has a ratio of the number of different items very high in comparison to the others. On the other hand, we have the "accidents" dataset which has a high number of transactions but a ratio of items/transactions very low.

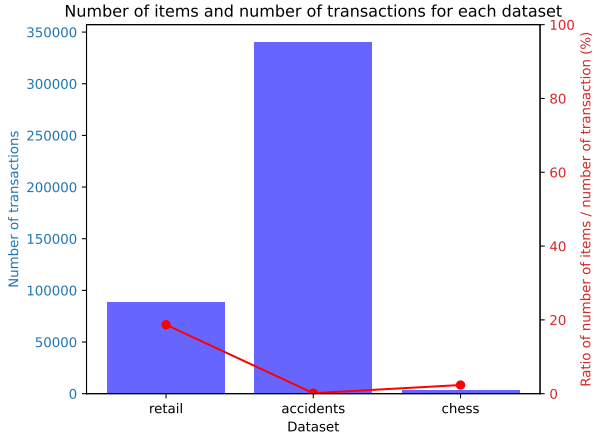

Fig. 2. number of items and transactions with ratio

The "retail" dataset has a high variation in the number of items by transactions, we observe that the maximum is far from the average. The "chess" dataset has exactly 37 items in each transaction, a very uniform repartition in opposition to `retail.dat`.

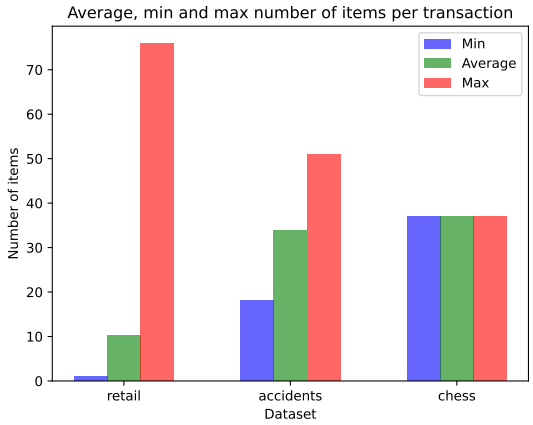

Fig. 3. average/min/max number of items per transaction

B. Execution time analysis

We will be comparing the execution time of our algorithms over 3 datasets with varying properties. We chose only the 3 last ones because the dummy version takes much more time and is thus not very relevant, we prefer having the 3 others in cleaner plots.

In the first two analyzed datasets, "accidents.dat" and "chess.dat", our `alternative_miner` algorithm outperforms the two different `apriori` algorithms (see Figure 4 and 5). This is due to the recursive deep-first search in the alternative miner which is well-optimized for datasets with a consequent number of large frequent itemsets.

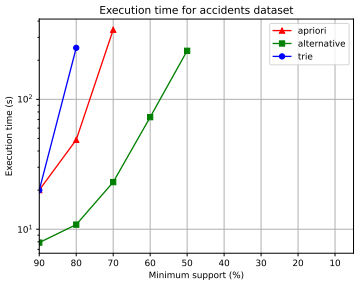

Fig. 4. Accidents dataset time analysis

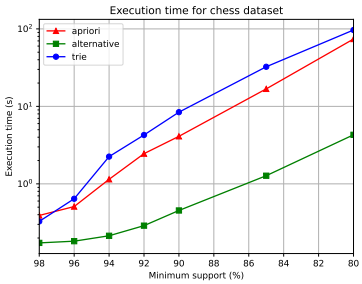

Fig. 5. Chess dataset time analysis

As we can see in Figure 8, the "retail.dat" dataset provides different results due to the format of the dataset. Indeed, there is a smaller number of frequent itemsets than in the other datasets when the frequency is above 0.1 as there are a lot of different items (see the ratio in Figure 2). The trie structure used in the `apriori_trie` algorithm allows us to take advantage of this small amount of frequent dataset, performing even better than the `alternative_miner` for values of `minFrequency` from $[0.2, 0.9]$.

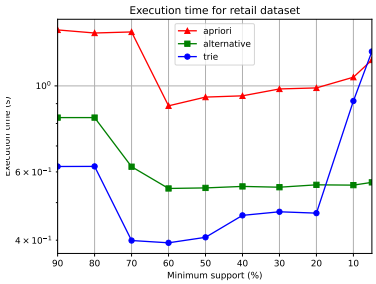

Fig. 6. Retail dataset time analysis

C. Use of memory space analysis

In this section, we will compare the space consumption of our tree algorithms on the chess dataset with a minimum frequency of 0.8.

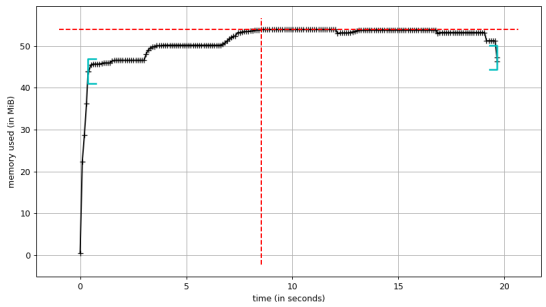

Fig. 7. Memory consumption of the Smart apriori algorithm

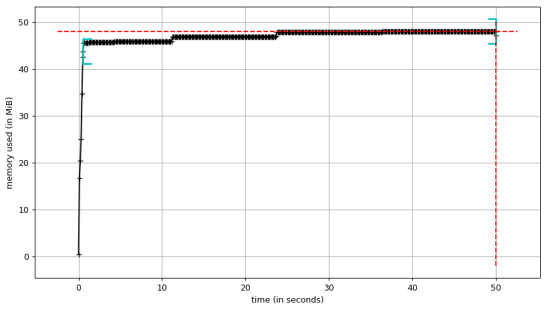

Fig. 8. Memory consumption of apriori algorithm with trie data structure

We can clearly spot differences between the three algorithms. At first, we can see that the ECLAT algorithm uses a lot of memory space in comparison to the two others. Its huge advantage in time is thus compensated for a higher space complexity. Now if we compare the two apriori algorithms, we can clearly see that, during the execution, the trie structure uses less space than the classical apriori algorithm. This is because we iteratively update the trie-structure and that we delete the unfrequent itemsets. This leads to less memory consumption and this can be useful in some specific cases.

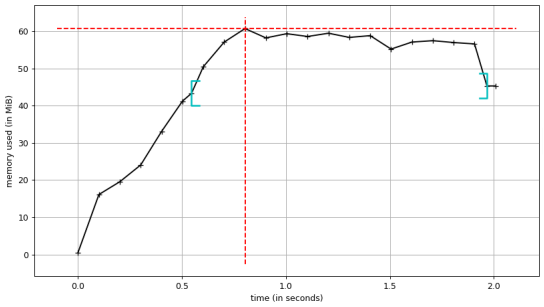

Fig. 9. Memory consumption of the ECLAT algorithm

IV. Difficulties encountered during the project

The main difficulty we encountered was probably the implementation of the apriori_trie algorithm, which was very challenging. We had to understand the [2] paper as much as possible and turn the content of the paper into code. We know that our algorithm is still not perfect but we can already see some advantages of using that kind of structure, especially for memory usage.

V. Conclusion

After implementing different frequent item-sets mining algorithms and analyzing their performance on several datasets, we can conclude that the ECLAT is the best one most of the time, even if the trie apriori can lead to better results in some particular situations. Indeed, ECLAT is able to get results for some low frequent thresholds, where the apriori algorithm would take an exponential amount of time. Another thing we can conclude is that it is important to check the structure of the dataset before applying a frequent itemset mining algorithm. Indeed for some datasets and some thresholds, the ECLAT algorithm is not always the best. Finally, using well-adapted data structures like tries is able to gain significant space in memory. It can be useful if we are limited in space or if a dataset is too big.

References

[1] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] F. Bodon, ”A fast APRIORI implementation,” Informatics Laboratory, Computer and Automation Research Institute, Hungarian Academy of Sciences, H-1111 Budapest, Lágymányosi u. 11, Hungary.

[3] GitHub - WissenY/FIM: Apriori, Eclat, FP-Growth algorithms with some dataset — github.com. https://github.com/WissenY/FIM, [Accessed 08-03-2024]