Mining Patterns in Data — Implementing Apriori

Gilles Peiffer (24321600), Liliya Semerikova (64811600)

***Abstract*—This paper studies the performance of various apriori implementations, as well as some depth-first search algorithms, for the frequent itemset mining problem. Implementations are discussed and compared to experimental results in order to identify bottlenecks, and gain understanding of the inner workings of the various algorithms seen during the lectures.**

Introduction

Apriori is arguably the simplest of the many algorithms developed in order to solve the frequent itemset mining problem, hereinafter referred to as "FIM". Formally, one can define a set of *transaction identifiers* $\mathcal{T}$, and a set of *items* $\mathcal{I}$. A *transactional database* $\mathcal{D}$ can then be represented as a function that returns, for every transaction identifier, a set of items: $\mathcal{D}\colon \mathcal{T} \to 2^{\mathcal{I}}$. An itemset $I \subseteq \mathcal{I}$ is said to *cover* a transaction $T \subseteq \mathcal{T}$ if and only if $I \subseteq T$. Similarly, the *cover* of an itemset $I \subseteq \mathcal{I}$ in a database $\mathcal{D}$ is the set of transactions it covers in the database. Finally, the *support* of an itemset is the cardinality of its cover. This can be extended to define the notion of *frequent*: an itemset is frequent if its normalized support is above a given minimum support threshold, ϑ. FIM is concerned with finding the set of frequent itemsets for a given database: $\{I \subseteq \mathcal{I} : |\{t \in \mathcal{T} : I \subseteq \mathcal{D}(t)\}| > \vartheta\}$.

I. Algorithms and Implementations

Various algorithms exist for FIM, each with its own properties. This section explains the various algorithms which were used for the performance tests of Section II.

A. Apriori

The apriori algorithm is the simplest of the FIM algorithms. Different implementations were used, each of which has its own characteristics. More information about the effect of using various data structures on the algorithm can be found in [1], [2].

1) Naive Apriori (`apriori_naive`)

Naive apriori is the most basic version of the apriori algorithm. It starts by generating all candidate subsets $\mathcal{S} \subseteq \mathcal{I}$ of a given cardinality, then, for each $\mathcal{S}$, it computes all subsets of $\mathcal{S}$ and checks whether these are all frequent. If not, $\mathcal{S}$ is discarded based on the anti-monotonicity property, but if they are, then the support of $\mathcal{S}$ is computed and is checked against the minimum support treshold. `apriori_naive2` does not implement this anti-monotonicity pruning. To compute the support, the itemset is checked against every transaction to determine whether it is a subset. This process is continued while there are frequent itemsets left at the level below.

2) Apriori with Prefix Generation (`apriori_prefix`)

Instead of generating all possible itemsets, those at level $\ell - 1$ can be combined if they share the same prefix, thus only generating candidates which already have at least two frequent subsets at level ℓ. Due to the reduced likelihood of having infrequent candidates, anti-monotonicity pruning is not used. Additionally, one can compute the number of covering transactions for singleton itemsets, and use this to quickly compute their support.

3) Apriori with Prefix Generation and Vertical Database Representation (`apriori_prefix_vdb`)

One can speed up the support computation by reusing the list of supporting transactions for each singleton itemset. To compute the support of a given itemset $I = \{i_1, \ldots, i_n\}$, one can then simply compute $\text{support}(I) = \bigcap_{j=1}^{n} \text{support}(\{i_j\})$, which can be done efficiently using the `set` data structure to store the vertical representations of the database. This algorithm thus uses projected databases, on which DFS algorithms are based.

4) Apriori with Prefix Generation and Vertical Database Representation using Dictionaries (`apriori_dict`)

Using dictionaries, one can easily store the vertical representations for larger itemsets as well, without recomputing intersections too often.

B. Depth-First Search

DFS-based algorithms are slightly more complex than apriori. In order to showcase the performance differences with apriori, this paper also explores some of these algorithms. DFS algorithms use the idea of projected databases to construct new candidate itemsets. Our first implementation, `dfs1`, uses a stack to make this happen in a depth-first manner, and a dictionary to store the vertical representations of the dataset, as explained in Section I-A4. By generating candidates more intelligently, one can reduce the number of generated candidate sets further, as done in the implementation `dfs2`. More information about DFS algorithms can be found in [3].

II. Performance

A. Function calls

Table II-A summarizes, for each algorithm ran on the "chess" dataset with a minimum frequency threshold $\vartheta = 0.98$, the number of candidates it generates; the number of times it computes the support of a candidate; and the number of frequent itemsets it finds. From this table, it is clear that for apriori, using prefixes to generate candidates is incredibly important. Despite all having the same number of functions calls, the "intelligent" apriori algorithms have sizeable differences in performance.

Algorithm	Generated	Support	Frequent
`apriori_naive`	18545215	93	21
`apriori_naive2`	18545215	18545215	21
`apriori_prefix`	94	94	21
`apriori_prefix_vdb`	94	94	21
`apriori_dict`	94	94	21
`dfs1`	509	509	509
`dfs2`	101	101	21

Table I
Number of generated candidates, supports computed and frequent itemsets found by each algorithm.

In the rest of this section, algorithms are analyzed both from an execution time point of view, and from a memory usage point of view, on each of the available datasets. Experiments were run on an Early 2015 MacBook Pro, using Python 3.6.8 and macOS Sierra 10.12.6, using a 2.9 GHz Intel Core i5 processor, with 8 GB of 1867 MHz DDR3 RAM and an Intel Iris Graphics 6100 GPU.

B. Time

All times were measured using the `time.perf_counter()` function, with a timeout of 200 s.

1) Accidents

As shown on Figure 1, the best algorithms on the "accidents" dataset are `dfs2` and `apriori_dict`, with the latter edging out the former for lower frequency thresholds. `dfs1` is by far the worst, which makes sense in light of Table II-A.

2) Chess

Figure 2 gives the execution time comparison for the "chess" dataset. All algoritms perform similarly, except for `apriori_prefix_vdb`, which performs significantly better.

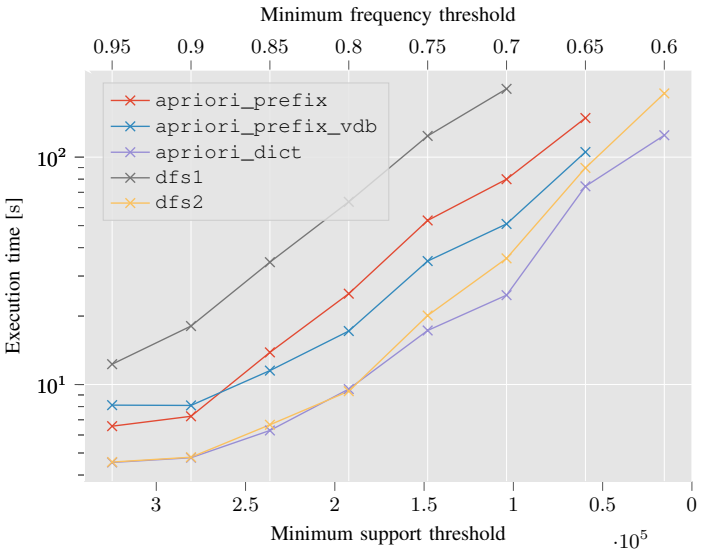

Figure 1. Execution time comparison for the “accidents” dataset.

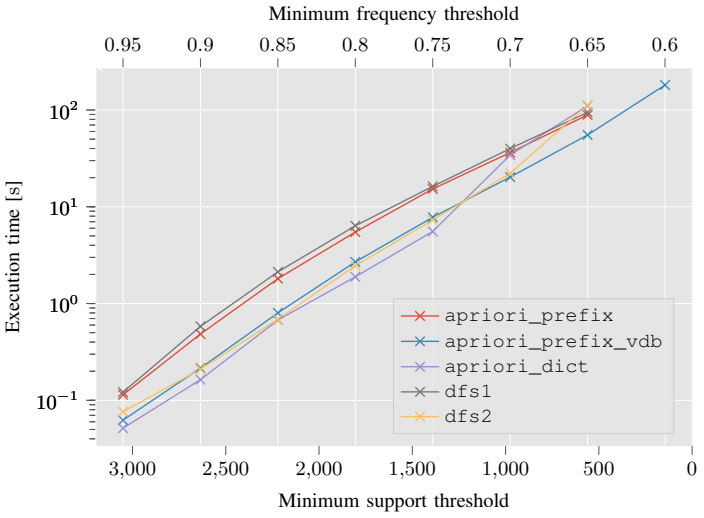

Figure 2. Execution time comparison for the “chess” dataset.

3) Connect
Figure 3, using the “connect” dataset, looks qualitatively very similar to Figure 2. Quantitatively, however, its viable frequency thresholds are much higher.

4) Mushroom
Figure 4 compares the algorithms on the “mushroom” dataset. It shows that `apriori_dict`, `apriori_prefix_vdb` and `dfs2` are the fastest algorithms. The execution times near the beginning are nearly constant and very small; this is probably due to the fact that the dataset initialization is taking up most of the execution time.

5) Pumsb
Figure 5 was created using the “pumsb” dataset. `apriori_dict` and `apriori_prefix_vdb` are clearly the fastest.

6) Pumsb_star
Figure 6 gives the execution time comparison for the “pumsb_star” dataset. `apriori_dict` is clearly the fastest,.

7) Retail
Figure 7 looks at the execution times on the “retail” dataset. `dfs2` and `apriori_dict` are the fastest algorithms, with the latter edging out the former for lower frequencies. Similarly to what is visible in Figure 4, the

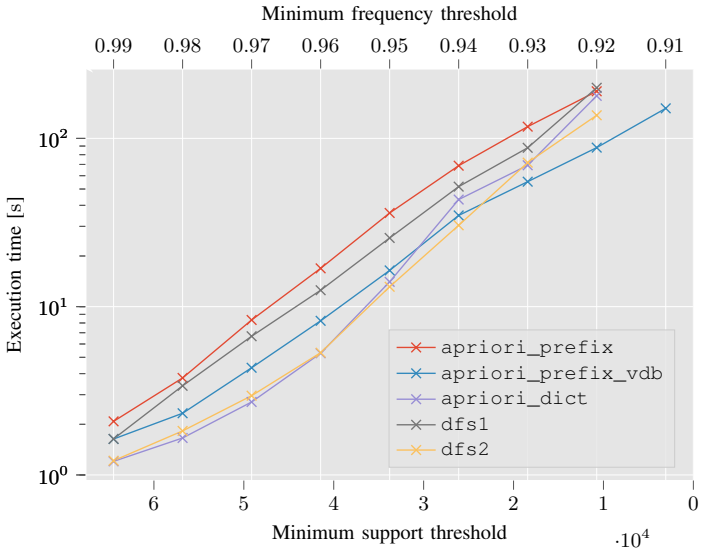

Figure 3. Execution time comparison for the “connect” dataset.

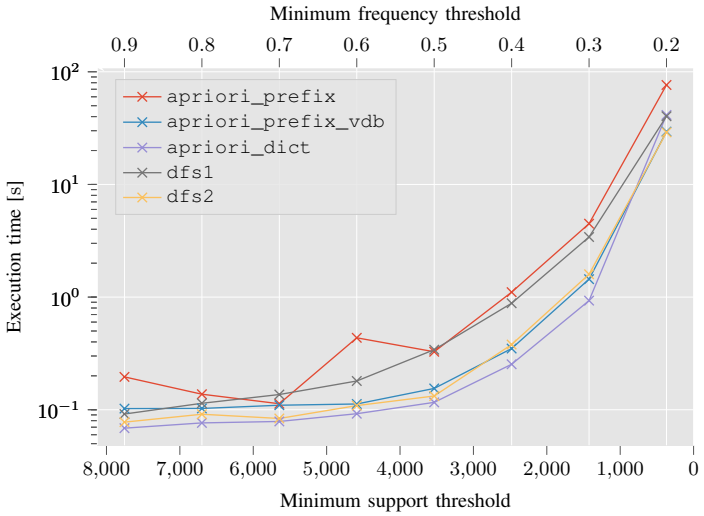

Figure 4. Execution time comparison for the “mushroom” dataset.

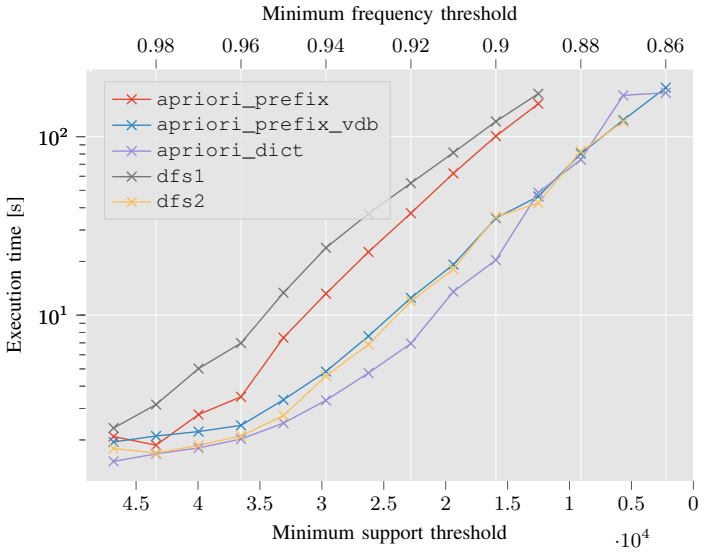

Figure 5. Execution time comparison for the “pumsb” dataset.

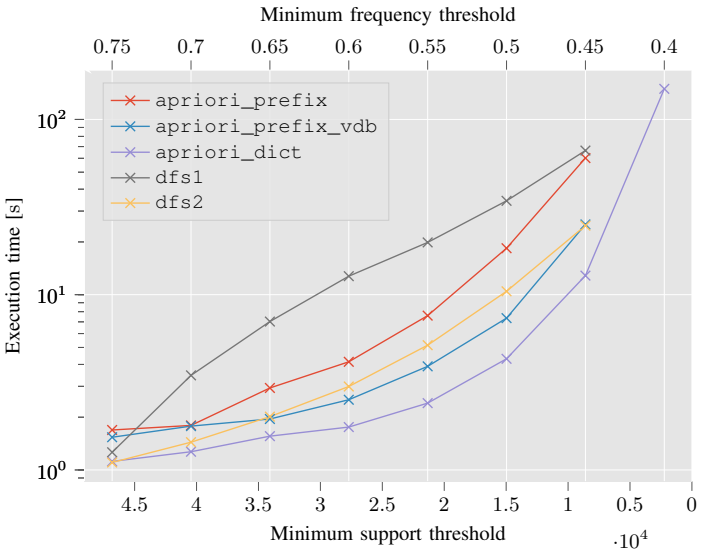

Figure 6. Execution time comparison for the "pumsb_star" dataset.

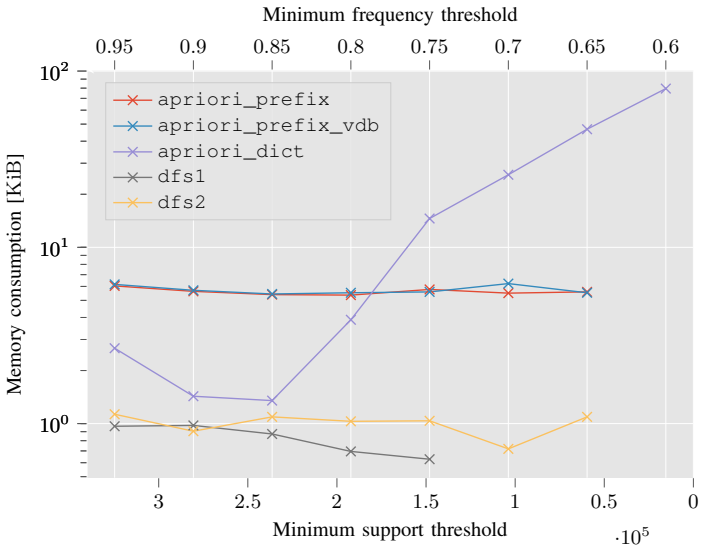

Figure 8. Memory consumption comparison for the "accidents" dataset.

execution times near the beginning are nearly constant and very small; this is probably due to the fact that the dataset initialization is taking up most of the execution time.

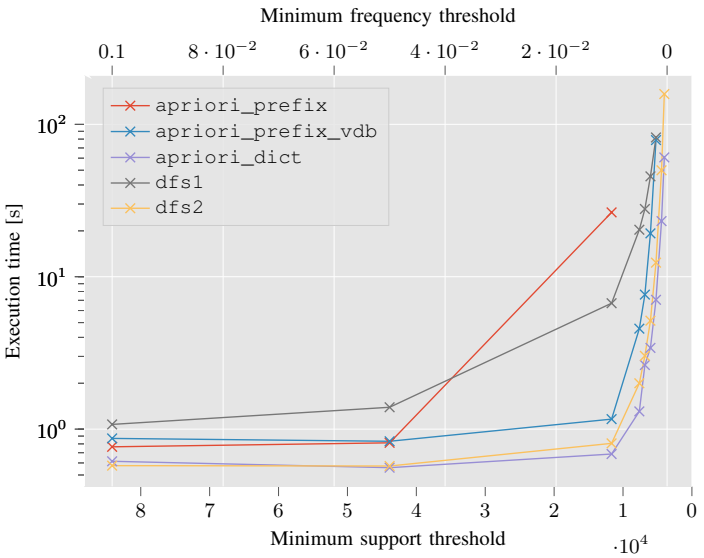

Figure 7. Execution time comparison for the "retail" dataset.

track of extra data structures to speed up computation. The other algorithms are approximately constant in their memory consumption.

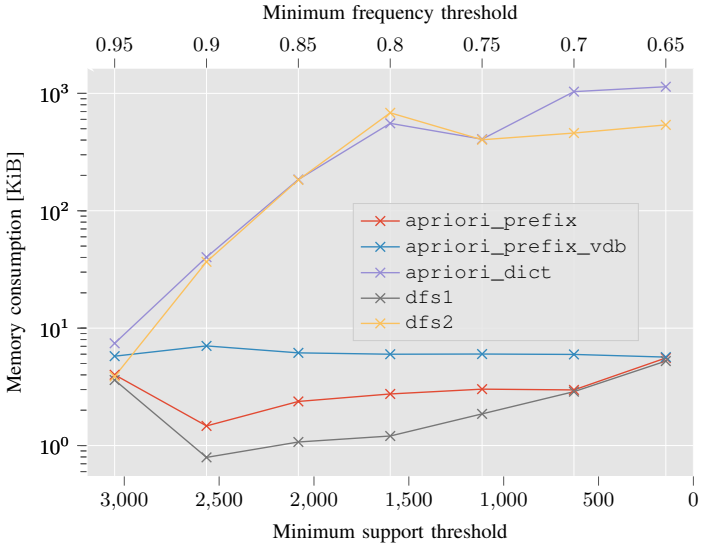

Figure 9. Memory consumption comparison for the "chess" dataset.

C. Memory

In order to measure the memory consumption of the program, we use `tracemalloc`, comparing snapshots before and after the function call. This method is far from perfect, but it is the most reliable choice in the Python standard library. Results therefore ought to be taken with a grain of salt.

1) Accidents

Figure 8 showcases the memory consumption on the "accidents" dataset. `apriori_dict` uses by far the most memory, as it maintains a large dictionary with vertical database representations. The other apriori algorithms are nearly constant, as are the DFS algorithms, though both classes are about an order of magnitude apart.

2) Chess

On Figure 9, which summarizes memory consumption on the "chess" dataset, one can observe again that `apriori_dict` usesthe most memory. `dfs2` is also memory-intensive, due to these algorithms keeping

3) Connect

The memory consumption for the "connect" dataset is shown on Figure 10, and is nearly identical (qualitatively) to the consumption on the "chess" dataset.

4) Mushroom

Figure 11 shows that the "mushroom" dataset follows the same trends as the others.

5) Pumsb

Figure 12 shows that on the "pumsb" dataset, everything is qualitatively similar, but quantitatively, `dfs1` uses about ten times less memory than the next best algorithm, size-wise.

6) Pumsb_star

Results for the "pumsb_star" dataset are nearly identical to the ones for "`pumsb`, as can be seen on Figure 13.

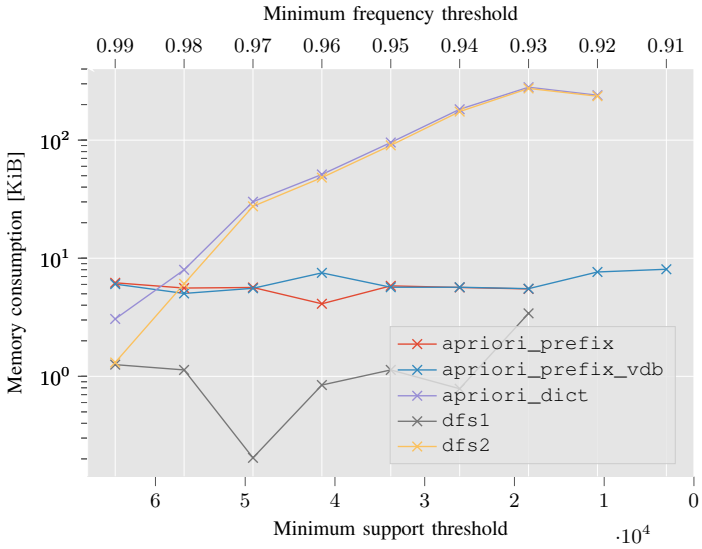

Figure 10. Memory consumption comparison for the “connect” dataset.

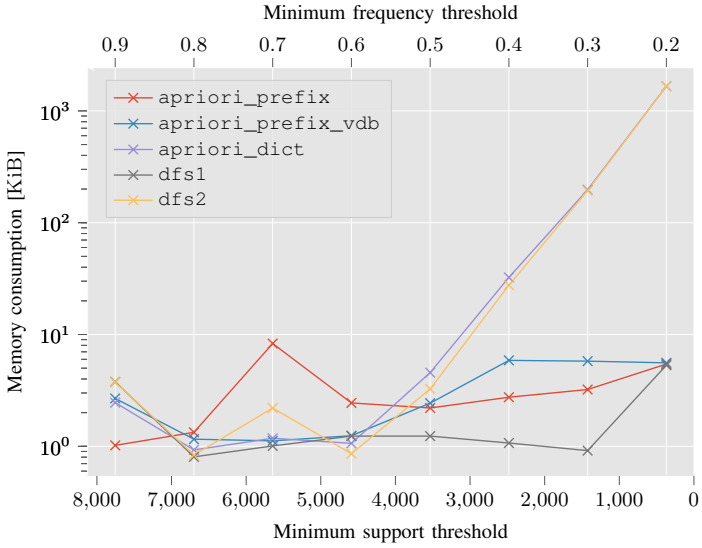

Figure 11. Memory consumption comparison for the “mushroom” dataset.

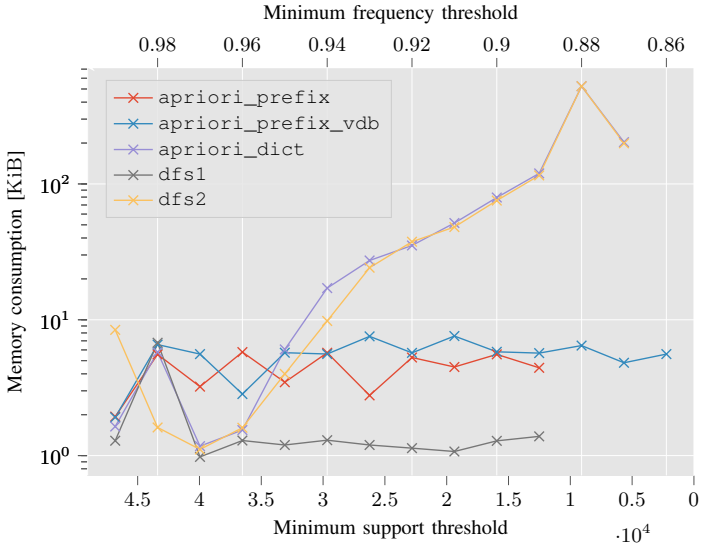

Figure 12. Memory consumption comparison for the “pumsb” dataset.

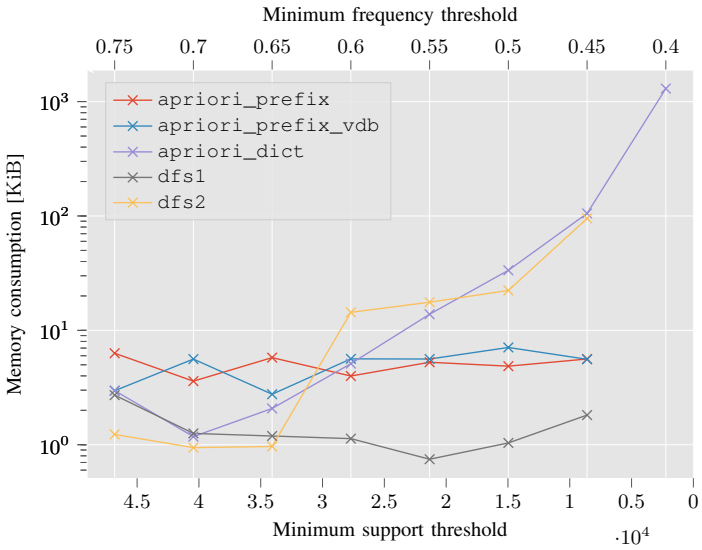

Figure 13. Memory consumption comparison for the “pumsb_star” dataset.

7) Retail

Figure 14 also shows that the simple apriori algorithms are nearly constant in their memory consumption. `apriori_dict` is unsurprisingly the most memory-intensive algorithm, but all algorithms get exponentially more memory-intensive for low frequency thresholds.

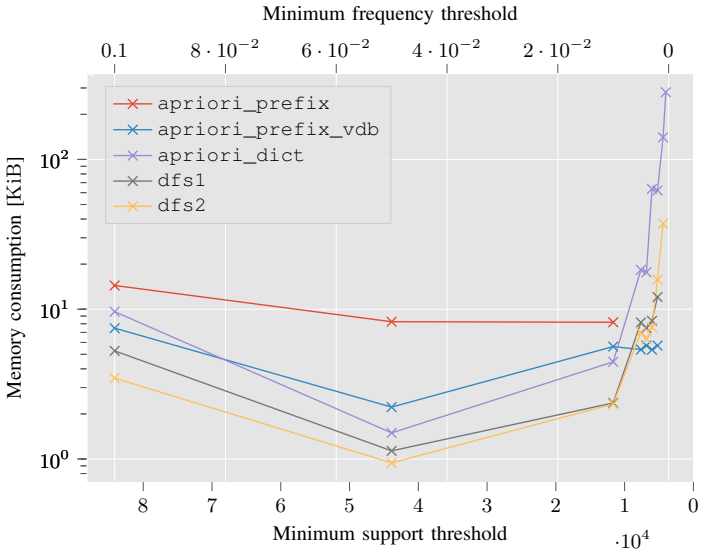

Figure 14. Memory consumption comparison for the “retail” dataset.

Conclusion

There are many things one can conclude from these experimental results. For a start, there is no algorithm which is both optimal from an execution time point of view and from a memory consumption point of view. A second thing to infer is that no algorithm is optimal (for a given characteristic) for every dataset. On the datasets that were tested, `apriori_dict` *usually* runs the fastest (but uses the most memory) while `dfs1` *usually* uses the least memory (but runs the slowest). Algorithm design for FIM is thus dependent on the properties of the dataset, in order to obtain a good trade-off between speed and size.

References

[1] Ferenc Bodon, *Surprising Results of Trie-based FIM Algorithms*, link, 2004.

[2] Gösta Grahne and Jianfei Zhu, *Efficiently Using Prefix-trees in Mining Frequent Itemsets*, link, 2003.

[3] Lars Schmidt-Thieme, *Algorithmic Features of Eclat*, link, 2004.