APRIORI, A Depth First Implementation

Walter A. Kosters
Leiden Institute of Advanced Computer Science
Universiteit Leiden
P.O. Box 9512, 2300 RA Leiden
The Netherlands
kosters@liacs.nl

Wim Pijls
Department of Computer Science
Erasmus University
P.O. Box 1738, 3000 DR Rotterdam
The Netherlands
pijls@few.eur.nl

Abstract

We will discuss $\mathcal{DF}$, the depth first implementation of APRIORI *as devised in 1999 (see [8]). Given a database, this algorithm builds a trie in memory that contains* all *frequent itemsets, i.e., all sets that are contained in at least* minsup *transactions from the original database. Here* minsup *is a threshold value given in advance. In the trie, that is constructed by adding one item at a time, every path corresponds to a unique frequent itemset. We describe the algorithm in detail, derive theoretical formulas, and provide experiments.*

1 Introduction

In this paper we discuss the depth first ($\mathcal{DF}$, see [8]) implementation of APRIORI (see [1]), one of the fastest known data mining algorithms to find *all* frequent itemsets in a large database, i.e., all sets that are contained in at least *minsup* transactions from the original database. Here *minsup* is a threshold value given in advance. There exist many implementations of APRIORI (see, e.g., [6, 11]). We would like to focus on algorithms that assume that the whole database fits in main memory, this often being the state of affairs; among these, $\mathcal{DF}$ and $\mathcal{FP}$ (the FP-growth implementation of APRIORI, see [5]) are the fastest. In most papers so far little attention has been given to theoretical complexity. In [3, 7] a theoretical basis for the analysis of these two algorithms was presented.

The depth first algorithm is a simple algorithm that proceeds as follows. After some preprocessing, which involves reading the database and a sorting of the single items with respect to their support, $\mathcal{DF}$ builds a trie in memory, where every path from the root downwards corresponds to a unique frequent itemset; in consecutive steps items are added to this trie one at a time. Both the database and the trie are kept in main memory, which might cause memory problems: both are usually very large, and in particular the trie gets much larger as the support threshold decreases. Finally the algorithm outputs all paths in the trie, i.e., all frequent itemsets. Note that once completed, the trie allows for fast itemset retrieval in the case of online processing.

We formerly had two implementations of the algorithm, one being time efficient, the other being memory efficient (called `dftime.cc` and `dfmemory.cc`, respectively), where the time efficient version could not handle low support thresholds. The newest version (called `dffast.cc`) combines them into one even faster implementation, and runs for all support thresholds.

In this paper we first describe $\mathcal{DF}$, we then give some formal definitions and theoretical formulas, we discuss the program, provide experimental results, and conclude with some remarks.

2 The Algorithm

An appropriate data structure to store the frequent itemsets of a given database is a *trie*. As a running example in this section we use the dataset of Figure 1. Each line represents a transaction. The trie of frequent patterns is shown in Figure 2. The entries (or cells) in a node of a trie are usually called *buckets*, as is also the case for a hash-tree. Each bucket can be identified with its path to the root and hence with a unique frequent itemset. The example trie has 9 nodes and 18 buckets, representing 18 frequent itemsets. As an example, the frequent itemset $\{A, B, E, F\}$ can be seen as the leftmost path in the trie; and a set as $\{A, B, C\}$ is not present.

One of the oldest algorithms for finding frequent patterns is APRIORI, see [1]. This algorithm successively finds all frequent 1-itemsets, all frequent 2-itemsets, all frequent 3-itemsets, and so on. (A k-itemset has k items.) The frequent k-itemsets are used to generate candidate $(k+1)$-itemsets,

Dataset

transaction number	items
1	B C D
2	A B E F
3	A B E F
4	A B C F
5	A B C E F
6	C D E F

Frequent itemsets when *minsup* $= 3$

support	frequent itemsets
5	B, F
4	A, AB, AF, ABF, BF, C, E, EF
3	AE, ABE, ABEF, AEF, BC, BE, BEF, CF

Figure 1. An example of a dataset along with its frequent itemsets.

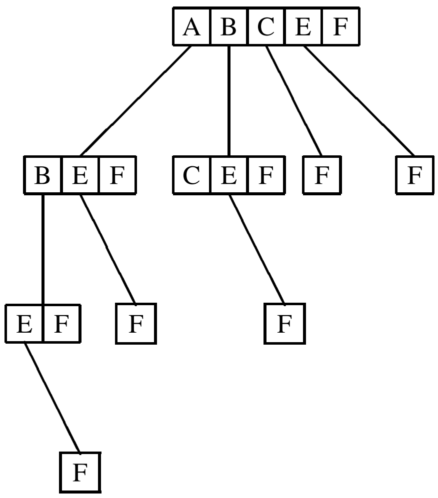

Figure 2. An example of a trie (without support counts).

where the candidates are only known to have two frequent subsets with k elements. After a pruning step, where candidates still having infrequent subsets are discarded, the support of the candidates is determined. The way APRIORI finds the frequent patterns implies that the trie is built layer by layer. First the nodes in the root (depth $= 0$) are constructed, next the trie nodes at depth 1 are constructed, and so on. So, APRIORI can be thought of as an algorithm that builds the pattern trie in a *breadth first* way. We propose an algorithm that builds the trie in a *depth first* way. We will explain the depth first construction of the trie using the dataset of Figure 1. Note that the trie grows from right to left.

The algorithm proceeds as follows. In a preprocessing step, the support of each single item is counted and the infrequent items are eliminated. Let the n frequent items be denoted by $i_1, i_2, \ldots, i_n$. Next, the code from Figure 3 is executed.

(1) $T :=$ *the trie including only bucket* i_n*;*
(2) **for** $m := n - 1$ **downto** *1* **do**
(3) $\quad T' := T$*;*
(4) $\quad T := T'$ *with* i_m *added to the left and*
$\qquad$ *a copy of* T' *appended to* i_m*;*
(5) $\quad S := T \setminus T'$ *(= the subtrie rooted in* i_m*);*
(6) $\quad$ *count*(S, i_m)*;*
(7) $\quad$ *delete the infrequent itemsets from* S*;*

(9) **procedure** *count*(S, i_m) *::*
(10) **for** *every transaction t including item* i_m **do**
(11) $\quad$ **for** *every itemset* I *in* S **do**
(12) $\qquad$ **if** t *supports* I **then** I*.support++;*

Figure 3. The algorithm.

The procedure *count*(S, i_m) determines the support of each itemset (bucket) in the subtrie S. This is achieved by a database pass, in which each transaction including item i_m is considered. Any such transaction is one at a time "pushed" through S, where it only traverses a subtrie if it includes the root of this subtrie, meanwhile updating the support fields in the buckets. In the last paragraph from Section 4 a refinement of this part of the algorithm is presented. On termination of the algorithm, T exactly contains the frequent itemsets.

Figure 4 illustrates the consecutive steps of the algorithm applied to our example. The single items surpassing the minimum support threshold 3 are $i_1 = A, i_2 = B, i_3 = C, i_4 = E$ and $i_5 = F$. In the figure, the shape of T after each iteration of the **for** loop is shown. Also the infrequent itemsets to be deleted at the end of an iteration are mentioned. At the start of the iteration with index m, the root of trie T consists of the 1-itemsets $i_{m+1}, \ldots, i_n$. (We denote a 1-itemset by the name of its only item, omitting curly braces and commas as in Figure 1 and Figure 4.) By the statement in line *(3)* from Figure 3, this trie may also be referred to as T'. A new trie T is composed by adding bucket i_m to the root and by appending a copy of T' (the former value of T) to i_m. The newly added buckets are the new candidates and they make up a subtrie S. In Figure 4, the candidate set S is in the left part of each trie and is drawn in bold. Notice that

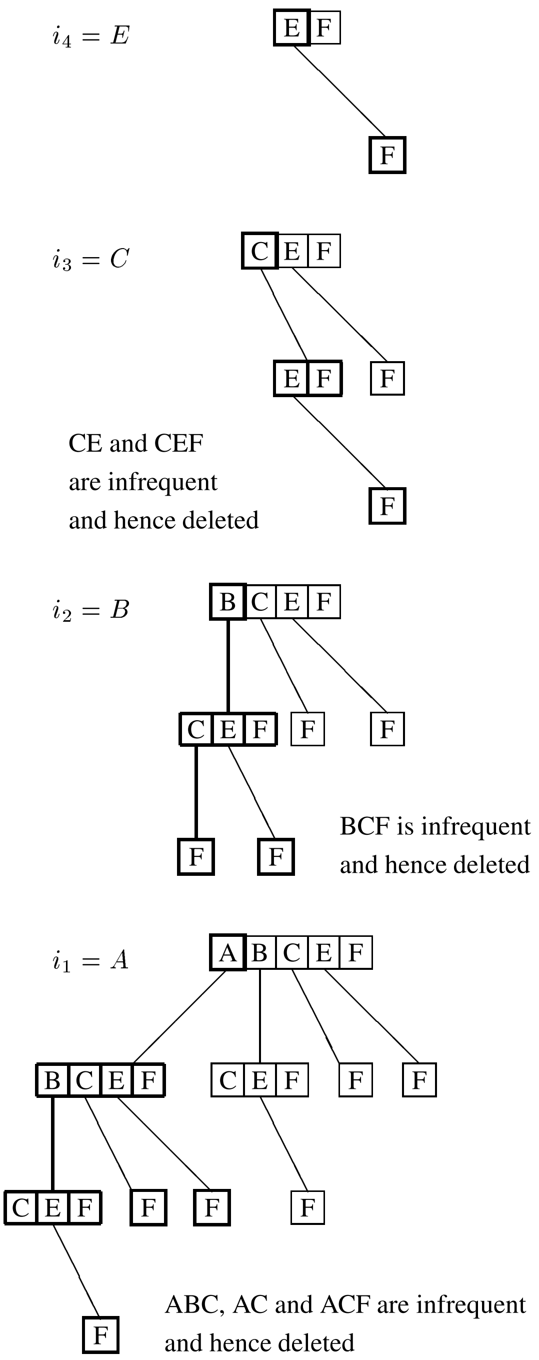

Figure 4. Illustrating the algorithm.

the final trie (after deleting infrequent itemsets) is identical to Figure 2.

The number of iterations in the **for** loop is one less than the number of frequent 1-itemsets. Consequently, the number of database passes is one less than the number of frequent 1-itemsets. This causes the algorithm to be tractable only if the database under consideration is memory-resident. Given the present-day memory sizes, this is not a real constraint any more.

As stated above, our algorithm has a preprocessing step which counts the support for each single item. After this preprocessing step, the items may be re-ordered. The most favorable execution time is achieved if we order the items by increasing frequency (see Section 3 for a more formal motivation). It is better to have low support at the top of the deeper side (to the left bottom) of the trie and hence, high support at the top of the shallow part (to the upper right) of the trie.

We may distinguish between "dense" data sets and "sparse" datasets. A dense dataset has many frequent patterns of large size and high support, as is the case for test sets such as `chess` and `mushroom` (see Section 5). In those datasets, many transactions are similar to each other. Datasets with mainly short patterns are called sparse. Longer patterns may exist, but with relatively small support. Real-world transaction databases of supermarkets mostly belong to this category. Also the synthetic datasets from Section 5 have similar properties: interesting support thresholds are much lower than in the dense case.

Algorithms for finding frequent patterns may be divided into two types: algorithms respectively with and without candidate generation Any APRIORI-like instance belongs to the first type. Eclat (see [9]) may also be considered as an instance of this type. The FP-growth algorithm $\mathcal{FP}$ from [5] is the best-known instance of the second type (though one can also defend the point of view that it does generate candidates). For dense datasets, $\mathcal{FP}$ performs better than candidate generating algorithms. $\mathcal{FP}$ stores the dataset in a way that is very efficient especially when the dataset has many similar transactions. In case of algorithms that do apply candidate generation, dense sets produce a large number of candidates. Since each new candidate has to be related to each transaction, the database passes take a lot of time. However, for sparse datasets, candidate generation is a very suitable method for finding frequent patterns. To our experience, the instances of the APRIORI family are very useful when searching transaction databases. According to the results in [7] the depth first algorithm $\mathcal{DF}$ outperforms FP-growth $\mathcal{FP}$ in the synthetic transaction sets (see Section 5 for a description of these sets).

Finally, note that technically speaking $\mathcal{DF}$ is not a full implementation of APRIORI, since every candidate itemset is known to have only one frequent subset (resulting from the part of the trie which has already been completed) instead of two. Apart from this, its underlying candidate generation mechanism strongly resembles the one from APRIORI.

3 Theoretical Complexity

Let m denote the number of transactions (also called customers), and let n denote the number of products (also called items). Usually m is much larger than n. For a non-empty itemset $A \subseteq \{1, 2, \ldots, n\}$ we define:

- $supp(A)$ is the *support* of A: the number of customers that buy all products from A (and possibly more), or equivalently the number of transactions that contain A;
- $sm(A)$ is the smallest number in A;
- $la(A)$ is the largest number in A.

In line with this we let $supp(\emptyset) = m$. We also put $la(\emptyset) = 0$ and $sm(\emptyset) = n + 1$. A set $A \subseteq \{1, 2, \ldots, n\}$ is called *frequent* if $supp(A) \geq minsup$, where the so-called *support threshold minsup* is a fixed number given in advance.

We assume every 1-itemset to be frequent; this can be effected by the first step of the algorithms we are looking at, which might be considered as preprocessing.

A "database query" is defined as a question of the form "Does customer C buy product P?" (or "Does transaction T has item I?"), posed to the original database. Note that we have mn database queries in the "preprocessing" phase in which the supports of the 1-itemsets are computed and ordered: every field of the database is inspected once. (By the way, the sorting, in which the items are assigned the numbers $1, 2, \ldots, n$, takes $O(n \log n)$ time.) The number of database queries for $\mathcal{DF}$ equals:

$$m(n-1) + \sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} \sum_{j=1}^{sm(A)-1} supp(\{j\} \cup A \setminus \{la(A)\}) \quad . \quad (1)$$

For a proof, see [3]. It relies on the fact that in order for a node to occur in the trie the path to it (except for the root) should be frequent, and on the observation that this particular node is "questioned" every time a transaction follows this same path. In [3] a simple version of $\mathcal{FP}$ is described in a similar style, leading to

$$\sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} \sum_{\substack{j=la(A)+1 \\ \{j\} \cup A \setminus \{la(A)\} \text{ frequent}}}^{n} supp(A) \qquad (2)$$

database queries in "local databases" (FP-trees), except for the preprocessing phase. Note the extra condition on the inner summation (which is "good" for $\mathcal{FP}$: we have less summands there), while on the other hand the summands are larger (which is "good" for $\mathcal{DF}$: we have a smaller contribution there).

It makes also sense to look at the total number of nodes of the trie during its construction, which is connected to the effort of maintaining and using the datastructures. Counting each trie-node with the number of buckets it contains, the total is computed to be:

$$n + \sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} \sum_{j=1}^{sm(A)-1} 1 = \sum_{A \text{ frequent}} [sm(A) - 1] \quad . \quad (3)$$

When the trie is finally ready the number of remaining buckets equals the number of frequent sets, each item in a node being the end of the path that represents the corresponding itemset.

Notice that the complexity heavily depends on the sorting order of the items at the top level. It turns out that an increasing order of items is beneficial here. This is suggested by the contribution of the 1-itemsets in Equation (1):

$$\sum_{i=1}^{n} (n - i) \; supp(\{i\}) \quad , \qquad (4)$$

which happens to be minimal in that case. This 1-itemset contribution turns out to be the same for both $\mathcal{DF}$ and $\mathcal{FP}$: see [3, 7], where also results for $\mathcal{FP}$ are presented in more detail.

4 Implementation Issues

In this section we discuss some implementation details of our program. As mentioned in Section 2, the database is traversed many times. It is therefore necessary that the database is memory-resident. Fortunately, only the occurrences of frequent items need to be stored. The database is represented by a two-dimensional boolean array. For efficiency reasons, one array entry corresponds to one bit. Since the function *count* in the algorithm considers the database transaction by transaction, a horizontal layout is chosen, cf. [4].

We have four preprocessing steps before the algorithm of Section 2 actually starts.

1 The range of the item values is determined. This is necessary, because some test sets, e.g., the BMS-WebView sets, have only values $> 10,000$.

2 This is an essential initial step. First, for each item the support is counted. Next, the frequent items are selected and sorted by frequency. This process is relevant, since the frequency order also prescribes the order in the root of the trie, as stated before. The sorted frequent items along with their supports are retained in an array.

3 If a transaction has zero or one frequent item, it needs not to be stored into the memory-resident representation of the database. The root of the trie is constructed

according to the information gathered in step 2. For constructing the other buckets, only transactions with at least two frequent items are relevant. In this step, we count the relevant transactions.

4 During this step the databases is stored into a two-dimensional array with horizontal layout. Each item is given a new number, according to its rank in the frequency order. The length of the array equals the result of step 3; the width is determined by the number of frequent items.

After this preparatory work, which in practice usually takes a few seconds, the code as described in Section 2 is executed. The cells of the root are constructed using the result of initial step 2.

In line *(12)* from Figure 3 in Section 2, backtracking is applied to inspect each path P of S. Inspecting a path P is aborted as soon as an item i with i outside the current transaction t is found. Obviously, processing one transaction during the *count* procedure is a relatively expensive task, which is unfortunately inevitable, whichever version of APRIORI is used.

As mentioned in the introduction, we used to have two implementations, one being time efficient, the other being memory efficient. These two have been used in the overall FIMI'03 comparisons. The newest implementation (called `dffast.cc`) combines these versions by using the following refinement. Instead of appending a copy T' of T to i_m (see Figure 3 in Section 2), first the counting is done in auxiliary fields in the original T, after which only the frequent buckets are copied underneath i_m. This makes the deletion of infrequent itemsets (line *(7)* from Figure 3) unnecessary and leads to better memory management. Another improvement might be achieved by using more auxiliary fields while adding two root items simultaneously in each iteration, thereby halving the number of database passes at the cost of more bookkeeping.

5 Experiments

Using the relatively small database `chess` (342 kB, with 3,196 transactions; available from the FIMI'03 website at `http://fimi.cs.helsinki.fi/testdata.html`), the database `mushroom` (570 kB, with 8,124 transactions; also available from the FIMI'03 website) and the well-known IBM-Almaden synthetic databases (see [2]) we shall examine the complexity of the algorithm. These databases have either few, but coherent records (`chess` and `mushroom`), or many records (the synthetic databases). The parameters for generating a synthetic database are the number of transactions D (in thousands), the average transaction size T and the average length I of so-called maximal potentially large itemsets. The number of items was set to $N = 1{,}000$, following the design in [2]. We use `T10I4D100K` (4.0 MB) and `T40I10D100K` (15.5 MB), both also available from the FIMI'03 website mentioned above; they both contain 100,000 transactions.

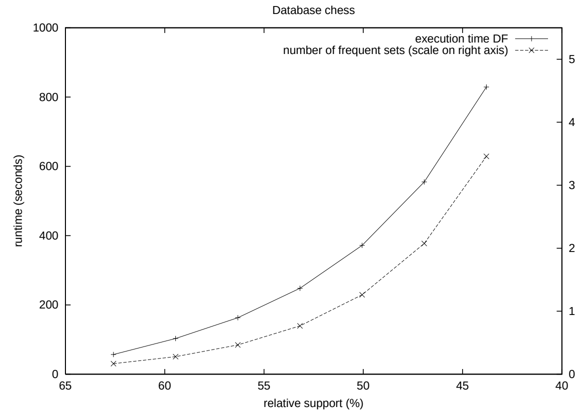

Figure 5. Experimental results for database `chess`.

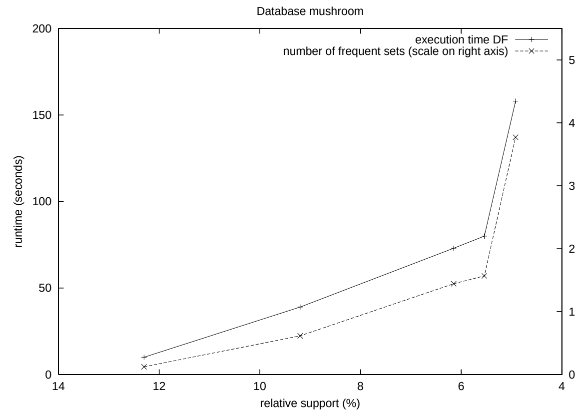

Figure 6. Experimental results for database `mushroom`.

The experiments were conducted at a Pentium-IV machine with 512 MB memory at 2.8 GHz, running Red Hat Linux 7.3. The program was developed under the GNU C++ compiler, version 2.96.

The following statistics are plotted in the graphs: the execution time in seconds of the algorithm (see Section 4), and the total number of frequent itemsets: in all figures the corresponding axis is on the right hand side and scales 0–5,500,000 (0–8,000,000 for `T10I4D100K`). The execution time excludes preprocessing: in this phase the database is read three times in order to detect the frequent items (see

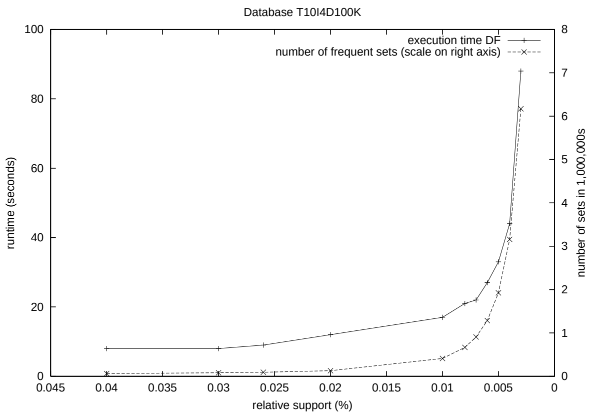

Figure 7. Experimental results for database `T10I4D100K`**.**

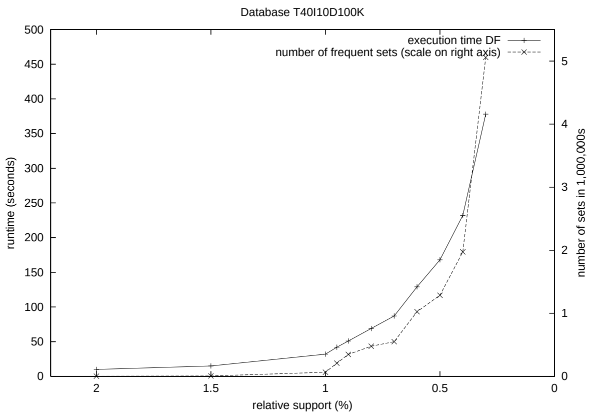

Figure 8. Experimental results for database `T40I10D100K`**.**

before); also excluded is the time needed to print the resulting itemsets. These actions together usually only take a few seconds. The number of frequent 1-itemsets (n from the previous sections, where we assumed all 1-itemsets to be frequent) has range 31–39 for the experiments on the database `chess`, 54–76 for `mushroom`, 844–869 for `T10I4D100K` and 610–862 for `T40I10D100K`. Note the very high support thresholds for `mushroom` (at least 5%) and `chess` (at least 44%); for `T10I4D100K` a support threshold as low as 0.003% was even feasible.

The largest output files produced are of size 110.6 MB (`chess`, *minsup* = 1,400, having 3,771,728 frequent sets with 13 frequent 17-itemsets), 121.5 MB (`mushroom`, *minsup* = 400, having 3,457,747 frequent sets with 24 frequent 17-itemsets), 131.1 MB (`T10I4D100K`, *minsup* = 3, having 6,169,854 frequent sets with 30 frequent 13-itemsets and 1 frequent 14-itemset) and 195.9 MB (`T40I10D100K`, *minsup* = 300, having 5,058,313 frequent sets, with 21 frequent 19-itemsets and 1 frequent 20-itemset). The final trie in the `T40I10D100K` case occupies approximately 65 MB of memory — the output file in this case being 3 times as large.

Note that the 3,457,747 sets for the `chess` database with *minsup* = 1,400 require 829 seconds to find, whereas the 3,771,728 frequent itemsets for `mushroom` with *minsup* = 400 take 158 seconds — differing approximately a factor 5 in time. This difference in runtime is probably caused by the difference in the absolute *minsup* value. Each cell corresponding to a frequent itemset is visited at least 1400 times in the former case against 400 times in the latter case. A similar phenomenon is observed when comparing `T40I10D100K` with absolute *minsup* value 300 and `T10I4D100K` with *minsup* = 3: this takes 378 versus 88 seconds. Although the outputs have the same orders of magnitude, the runtimes differ substantially. We see that, besides the number of frequent itemsets and the sizes of these sets, the absolute *minsup* value is a major factor determining the runtime.

6 Conclusions

In this paper, we addressed $\mathcal{DF}$, a depth first implementation of APRIORI. To our experience, $\mathcal{DF}$ competes with any other well-known algorithm, especially when applied to large databases with transactions.

Storing the database in the primary memory is no longer a problem. On the other hand, storing the candidates causes trouble in situations, where a dense database is considered with a small support threshold. This is the case for any algorithm using candidates. Therefore, it would be desirable to look for a method which stores candidates in secondary memory. This is an obvious topic for future research. To our knowledge, $\mathcal{FP}$ is the only algorithm that can cope with memory limitations. However, for real world retail databases this algorithm is surpassed by $\mathcal{DF}$, as we showed in [7]. Other optimizations might also be possible. Besides improving the C++ code, ideas from, e.g., [10] on diffsets with vertical layouts might be used.

Our conclusion is that $\mathcal{DF}$ is a simple, practical, straightforward and fast algorithm for finding all frequent itemsets.

References

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI/MIT Press, 1996.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J.B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings 20th International Conference on Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 1994.

[3] J.M. de Graaf, W.A. Kosters, W. Pijls, and V. Popova. A theoretical and practical comparison of depth first and FP-growth implementations of Apriori. In H. Blockeel and M. Denecker, editors, *Proceedings of the Fourteenth Belgium-Netherlands Artificial Intelligence Conference (BNAIC 2002)*, pages 115–122, 2002.

[4] B. Goethals. Survey on frequent pattern mining. Helsinki, 2003. `http://www.cs.helsinki.fi/u/-goethals/publications/survey.ps`.

[5] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, pages 1–12, 2000.

[6] J. Hipp, U. Günther, and G. Nakhaeizadeh. Mining association rules: Deriving a superior algorithm by analyzing today's approaches. In D.A. Zighed, J. Komorowski, and J. Żytkov, editors, *Principles of Data Mining and Knowledge Discovery, Proceedings of the 4th European Conference (PKDD 2000)*, Springer Lecture Notes in Computer Science 1910, pages 159–168. Springer Verlag, 2000.

[7] W.A. Kosters, W. Pijls, and V. Popova. Complexity analysis of depth first and FP-growth implementations of Apriori. In P. Perner and A. Rosenfeld, editors, *Machine Learning and Data Mining in Pattern Recognition, Proceedings MLDM 2003*, Springer Lecture Notes in Artificial Intelligence 2734, pages 284–292. Springer Verlag, 2003.

[8] W. Pijls and J.C. Bioch. Mining frequent itemsets in memory-resident databases. In E. Postma and M. Gyssens, editors, *Proceedings of the Eleventh Belgium-Netherlands Conference on Artificial Intelligence (BNAIC1999)*, pages 75–82, 1999.

[9] M.J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12:372–390, 2000.

[10] M.J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proceedings 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2003.

[11] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In F. Provost and R. Srikant, editors, *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2001)*, pages 401–406, 2001.