Keeping Things Simple: Finding Frequent Item Sets by Recursive Elimination

Christian Borgelt

Department of Knowledge Processing and Language Engineering
School of Computer Science, Otto-von-Guericke-University of Magdeburg
Universitätsplatz 2, 39106 Magdeburg, Germany

borgelt@iws.cs.uni-magdeburg.de

ABSTRACT

Recursive elimination is an algorithm for finding frequent item sets, which is strongly inspired by the FP-growth algorithm and very similar to the H-mine algorithm. It does its work without prefix trees or any other complicated data structures, processing the transactions directly. Its main strength is not its speed (although it is not slow, even outperforms Apriori and Eclat on some data sets), but the simplicity of its structure. Basically all the work is done in one simple recursive function, which can be written with relatively few lines of code.

1. INTRODUCTION

One of the currently fastest and most popular algorithms for frequent item set mining is the FP-growth algorithm [7]. It is based on a prefix tree representation of the given database of transactions (called an FP-tree), which can save considerable amounts of memory for storing the transactions. A close relative of this approach is the H-mine algorithm [9], which uses a somewhat simpler data structure called an H-struct, which is faster to build than an FP-tree.

The basic idea of both algorithms can be described as a *recursive elimination* scheme: in a preprocessing step delete all items from the transactions that are not frequent individually, i.e., do not appear in a user-specified minimum number of transactions. Then select all transactions that contain the least frequent item (least frequent among those that are frequent), delete this item from them, and recurse to process the obtained reduced database, remembering that the item sets found in the recursion share the item as a prefix. On return, remove the processed item also from the database of all transactions and start over, i.e., process the second frequent item etc. In these processing steps the prefix tree (or the H-struct), which is enhanced by links between the branches, is exploited to quickly find the transactions containing a given item and also to remove this item from the transactions after it has been processed.

In this paper I study an algorithm that is based on a very similar scheme, but does its work without a prefix tree or an H-struct representation. It rather processes the transactions directly, organizing them merely into singly linked lists. The main advantage of such an approach is that the needed data structures are very simple and that no re-representation of the transactions is necessary, which saves memory in the recursion. In addition, processing the transactions is almost trivial and can be coded in a single recursive function with relatively few lines of code. Surprisingly enough, the price one has to pay for this simplicity is relatively small: my implementation of this recursive elimination scheme yields competitive execution times compared to my implementations [4, 5] of the Apriori [1, 2] and Eclat [10] algorithms.

2. RECURSIVE ELIMINATION

As already indicated in the introduction, recursive elimination is based on a step by step elimination of items from the transaction database together with a recursive processing of transaction subsets. This section describes the details of this scheme as well as some implementation issues.

2.1 Preprocessing

Similar to several other algorithms for frequent item set mining, like, for example, Apriori or FP-growth, recursive elimination preprocesses the transaction database as follows: in an initial scan the frequencies of the items (support of single element item sets) are determined. All infrequent items—that is, all items that appear in fewer transactions than a user-specified minimum number—are discarded from the transactions, since, obviously, they can never be part of a frequent item set.

In addition, the items in each transaction are sorted, so that they are in *ascending* order w.r.t. their frequency in the database. Although the algorithm does not depend on this specific order, experiments showed that it leads to much shorter execution times than a random order. A *descending* order led to a particularly slow operation in my experiments, performing even worse than a random order.

This preprocessing is demonstrated in Table 1, which shows an example transaction database on the left. The frequencies of the items in this database, sorted ascendingly, are shown in the table in the middle. If we are given a user specified minimal support of 3 transactions, items f and g can be discarded. After doing so and sorting the items in

a d f		a d
c d e		e c d
b d		b d
a b c d		a c b d
b c		c b
a b d		a b d
b d e		e b d
b c e g		e c b
c d f		c d
a b d		a b d

g	1
f	2
e	3
a	4
c	5
b	7
d	8

Table 1: Transaction database (left), item frequencies (middle), and reduced transaction database with items in transactions sorted ascendingly w.r.t. their frequency (right).

each transaction ascendingly w.r.t. their frequencies we obtain the reduced database shown in Table 1 on the right.

2.2 Transaction Representation

Each transaction is represented as a simple arrays of item identifiers (which are integer numbers). The initial transaction database is turned into a set of transaction lists, with one list for each item. These lists are stored in a simple array, each element of which contains a support counter and a pointer to the head of the list. The list elements themselves consist only of a successor pointer and a pointer to (or rather into, see below) the transaction. The transactions are inserted one by one into this structure by simply using their leading item as an index. However, the leading item is removed from the transaction, that is, the pointer in the transaction list element points to the second item. Note that this does not lose any information as the first item is implicitely represented by the list the transaction is in.

To illustrate this, Figure 1 shows, at the very top, the representation of the reduced database shown in Table 1 on the right. The first list, corresponding to the item e, contains the second, seventh and eight transaction, with the item e removed. The counter in the array element states the number of transactions containing the corresponding item. It should be noted, as will become clear later, that this counter is not always equal to the length of the associated list, although this is the case for this initial representation of the database. Differences result from (shrunk) transactions that contain no other items and are thus not represented in the list.

For implementations it is important to note that the described scheme, with a pointer into the transaction so that the leading item is skipped, can only be applied in languages that allow for pointer arithmetic. In languages in which this is impossible (like, for instance, Java) the items in the transactions may be sorted the other way round and an element counter, stored in the list elements, may used to specify the subset of the items that is to be considered.

2.3 Recursive Processing

Recursive elimination works as follows: The array of lists that represents a (reduced) transaction database is "disassembled" by traversing it from left to right, processing the transactions in a list in a recursive call to find all frequent

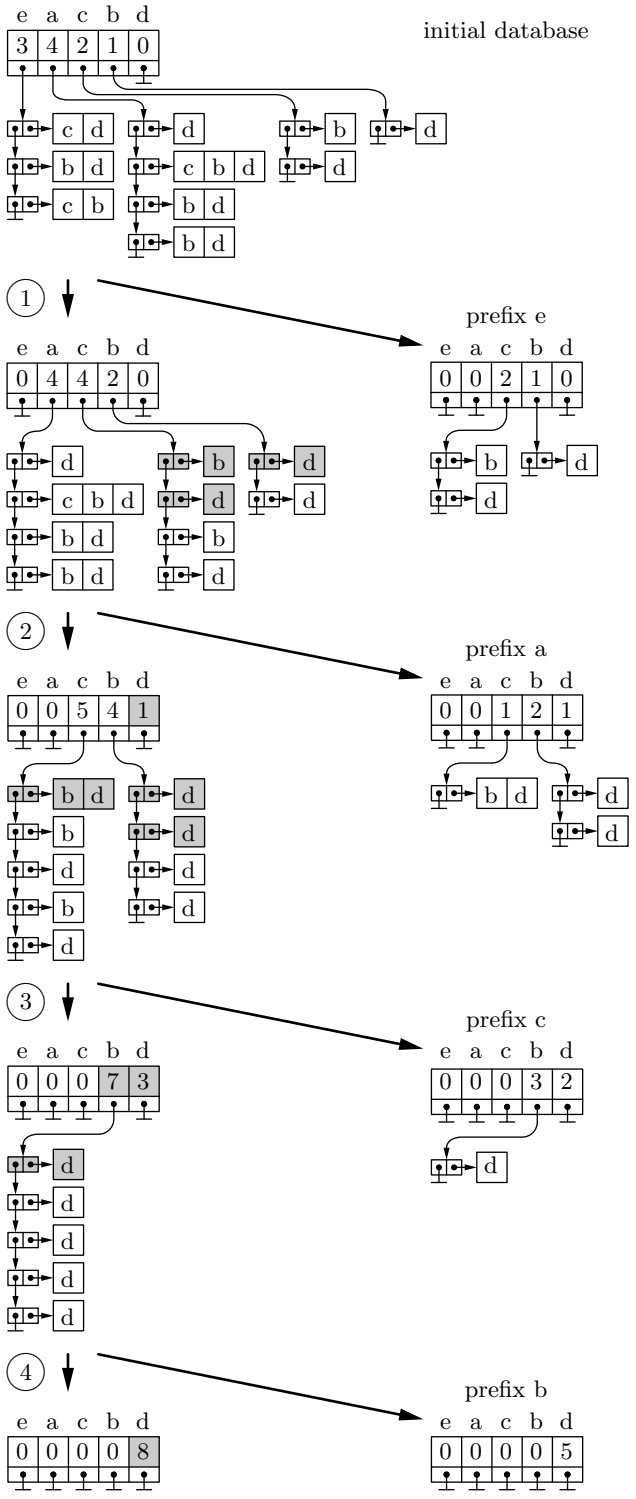

Figure 1: Procedure of the recursive elimination with the modification of the transaction lists (left) as well as the construction of the transaction lists for the recursion (right).

item sets that contain the item the list corresponds to. After a list has been processed recursively, its elements are either reassigned to the remaining lists or discarded (depending on the transactions they represent), and the next list is worked on. Since all reassignments are made to lists that lie to the right of the currently processed one, the list array will finally be empty (will contain only empty lists).

Before a transaction list is processed, however, its support counter is checked, and if it exceeds the user-specified minimum support, a frequent item set is reported, consisting of the item associated with the list and a possible prefix associated with the whole list array (see below).

One transaction list is processed as follows: for each list element the leading item of its (shrunk) transaction is retrieved and used as an index into the list array; then the element is added at the head of the corresponding list. In such a reassignment, the leading item is also removed from the transaction, which can be implemented as a simple pointer increment (or as a counter decrement, see above). In addition, a copy of the list element (with the leading item of the transaction already removed by the pointer increment) is inserted in the same way into an initially empty second array of transaction lists. (Note that only the list element is copied, *not* the transaction. Both list elements, the reassigned one and the copy refer to the same transaction.)

Since the elements of a transaction list all share an item (given by the list index), this second array collects the subset of transactions that contain a specific item and represents them as a set of transaction lists. This set of transaction lists is then processed recursively, noting the item associated with the list it was generated from as a common prefix of all frequent item sets found in the recursion. After the recursion the next transaction list is reassigned, copied, and processed in a recursive call and so on.

The process is illustrated for the root level of the recursion in Figure 1, which shows the transaction list representation of the initial database at the very top. In the first step all item sets containing the item e are found by processing the leftmost list. The elements of this list are reassigned to the lists to the right (grey list elements) and copies are inserted into a second list array (shown on the right). This second list array is then processed recursively, before proceeding to the next list, i.e., the one for item a.

Note that a list element representing a (shrunk) transaction that contains only one item is neither reassigned nor copied, because the transaction would be empty after the removal of the leading item. Instead only the counter in the lists array element is incremented as an indicator of such list elements. Such a situation occurs when the list corresponding to the item a is processed. The first list element refers to a (shrunk) transaction that contains only item d and thus only the counter for item d (grey) is incremented. For the same reason only one of the five elements in the list for item c is reassigned/copied in step 3.

After four steps all transaction lists have been processed and the lists array has become empty. Note that the list for the last element (referring to item d) is always empty, because there are no items left that could be in a transaction and thus all transactions are represented in the counter.

2.4 Optimization Issues

Obviously the allocation of the elements of the transaction lists is a critical issue. This can be done, for example, with a specialized memory management for equally sized small objects, which allocates them in large arrays and then distributes them individually as needed. A related alternative to solve the problem works as follows: in the first place, since by the number of transactions we know the maximum number of list elements we will ever have to create on a recursion level, we can allocate the list elements as an array, in one block of memory. Secondly, we can store the allocated memory blocks in a globally accessible place, so that we only have to allocate them the first time we reach a particular recursion depth and simply reuse them on the second time. This wastes some memory, since one has to allocate on each recursion level as many list elements as there are transactions in the original database in order to be sure that they suffice in all recursion branches. However, since each list element is fairly small (8 bytes on a 32-bit machine) the total amount of needed memory is acceptable. Furthermore, the gains in computation time are substantial and definitely justify this small waste of memory.

Of course, the same scheme of storing allocated blocks of memory in a globally accessible place and resuing them in the recursion is also used for the list arrays. Furthermore, it is clear that copies of the list elements are created only if the support counter corresponding to the currently processed list exceeds the user-specified minimum support, because otherwise no frequent item sets can be found in the recursion and hence building the lists is not necessary.

A final optimization issue concerns the traversal of the lists array. Since in the second lists array only the entries beyond the entry for the item corresponding to the currently processed list can be filled, the traversal can be started at this point in the recursion by passing its index.

2.5 Extensions

An idea that suggests itself when considering how the described recursive elimination scheme may be improved is to use prefix trees instead of simple arrays of item identifiers to represent the transactions. The transaction lists will then be lists of transaction trees and thus can be expected to be shorter. Howver, the disadvantage of such an approach is that processing a list gets much more complex, since reassigning a transaction prefix tree while removing the first item from the represented transactions involves splitting it into its branches, assigning each branch to a different list. In addition, the prefix tree itself is already a much more complicated data structure.

Nevertheless I implemented it, too, in order to check its merits in experiments. Unfortunately the hopes I placed in it were not fulfilled as the experiments reported in the following section show: most of the time using prefix trees actually degrades performance. The cause of this behavior presumably are the more complex operations involved in reassigning prefix trees compared to simple transactions.

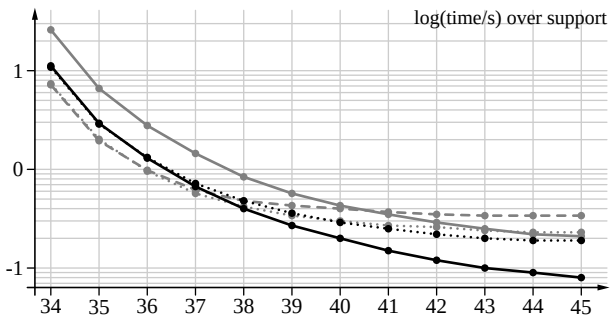

Figure 2: Results on BMS-Webview-1

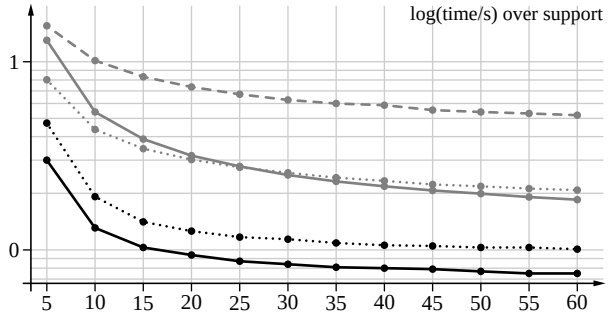

Figure 3: Results on T10I4D100K

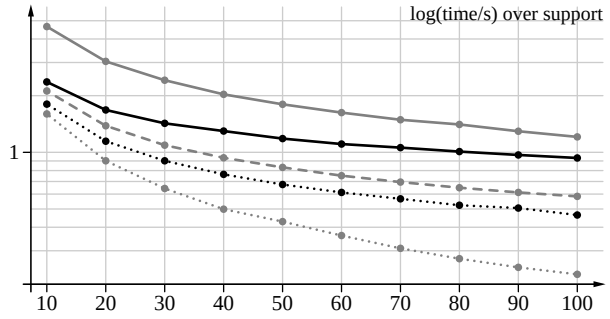

Figure 4: Results on census

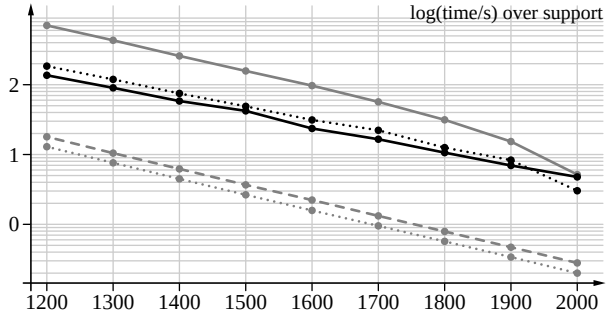

Figure 5: Results on chess

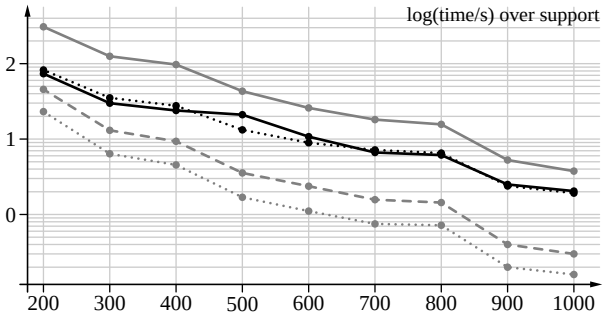

Figure 6: Results on mushroom

3. EXPERIMENTAL RESULTS

I ran experiments on the same five data sets that I already used in [4, 5], namely BMS-Webview-1 [8], T10I4D100K [11], census, chess, and mushroom [3]. However, I used a different machine and an updated operating system, namely a Pentium 4C 2.6GHz system with 1 GB of main memory running S.u.S.E. Linux 9.3 and gcc version 3.3.5). The results were compared to experiments with my implementations of Apriori, Eclat, and FPgrowth. All experiments were rerun to ensure that the results are comparable.

Figures 2 to 6 show, each for one of the five data sets, the decimal logarithm of the execution time over different minimum support values. The solid black line refers to the recursive elimination algorithm studied here, the dotted black line to the version that uses transaction prefix trees. The grey lines represent the corresponding results for Apriori (solid line), Eclat (dashed line), and FPgrowth (dotted line).

In general the recursive elimination algorithm seems to perform in a similar way as Apriori, as can be seen from the fairly similar shapes of the solid black and solid grey lines in all diagrams. However, the recursive elimination algorithm always outperforms Apriori by a considerable margin, on all data sets, particularly pronounced on T10I4D100K. In addition, its superiority usually increases with lower values of the minimum support, an effect that is clearly visible on census, chess, and mushroom.

In a comparison with Eclat recursive elimination also fares pretty well. Although it is bet clearly on chess and, for lower support, on BMS-Webview-1, it almost reaches Eclat's performance on census and mushroom for lower support. FPgrowth, however, fairly clearly outperforms recursive elimination on census, chess, and mushroom.

Although using prefix trees may seem like a good idea at first sight, it almost always degrades performance. The only exception is census, on which prefix trees lead to a substantial gain, even though its relative superiority shrinks with higher support values. Separating the execution times for the tree construction and the recursive elimination shows that it is not the tree construction, but the more complex later processing that is the cause. The gains resulting from the reduced list lengths (which, after some reassignments of a transaction tree, cannot be expected to be so substantial anyway) seem to be too small to outweigh this effect.

4. CONCLUSIONS

Even though its underlying scheme—which is based on deleting items, recursive processing, and reassigning transactions—is very simple and works without complicated data structures, recursive elimination performs surprisingly well, as can be seen from the experiments reported in the preceding section. If a quick and straightforward implementation is desired, it could be the method of choice.

5. PROGRAM

The implementation of the recursive elimination algorithm described in this paper (Windows™ and Linux™ executables as well as the source code, distributed under the LGPL) can be downloaded free of charge at

http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html

At this URL my implementations of Apriori, Eclat, and FP-growth are also available as well as a graphical user interface (written in Java) for finding association rules with Apriori.

6. REFERENCES

[1] R. Agrawal, T. Imielienski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. *Proc. Conf. on Management of Data*, 207–216. ACM Press, New York, NY, USA 1993

[2] A. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. Fast Discovery of Association Rules. In: [6], 307–328

[3] C.L. Blake and C.J. Merz. *UCI Repository of Machine Learning Databases*. Dept. of Information and Computer Science, University of California at Irvine, CA, USA 1998 http://www.ics.uci.edu/~mlearn/MLRepository.html

[4] C. Borgelt. Efficient Implementations of Apriori and Eclat. *Proc. 1st IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL)*. CEUR Workshop Proceedings 90, Aachen, Germany 2003. http://www.ceur-ws.org/Vol-90/

[5] C. Borgelt. Recursion Pruning for the Apriori Algorithm. *Proc. 2nd IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI 2003, Brighton, United Kingdom)*. CEUR Workshop Proceedings 126, Aachen, Germany 2004. http://www.ceur-ws.org/Vol-126/

[6] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, eds. *Advances in Knowledge Discovery and Data Mining*. AAAI Press / MIT Press, Cambridge, CA, USA 1996

[7] J. Han, H. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In: *Proc. Conf. on the Management of Data (SIGMOD'00, Dallas, TX)*. ACM Press, New York, NY, USA 2000

[8] R. Kohavi, C.E. Bradley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 Organizers' Report: Peeling the Onion. SIGKDD Exploration 2(2):86–93. 2000.

[9] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases. *IEEE Conf. on Data Mining (ICDM'01, San Jose, CA)*, 441–448. IEEE Press, Piscataway, NJ, USA 2001

[10] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. *Proc. 3rd Int. Conf. on Knowledge Discovery and Data Mining (KDD'97)*, 283–296. AAAI Press, Menlo Park, CA, USA 1997

[11] Synthetic Data Generation Code for Associations and Sequential Patterns. Intelligent Information Systems, IBM Almaden Research Center http://www.almaden.ibm.com/software/quest/Resources/index.shtml