An Implementation of the FP-growth Algorithm

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

The FP-growth algorithm is currently one of the fastest approaches to frequent item set mining. In this paper I describe a C implementation of this algorithm, which contains two variants of the core operation of computing a projection of an FP-tree (the fundamental data structure of the FP-growth algorithm). In addition, projected FP-trees are (optionally) pruned by removing items that have become infrequent due to the projection (an approach that has been called FP-Bonsai). I report experimental results comparing this implementation of the FP-growth algorithm with three other frequent item set mining algorithms I implemented (Apriori, Eclat, and Relim).

1. INTRODUCTION

One of the currently fastest and most popular algorithms for frequent item set mining is the FP-growth algorithm [8]. 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. The basic idea of the FP-growth algorithm 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) and delete this item from them. Recurse to process the obtained reduced (also known as *projected*) database, remembering that the item sets found in the recursion share the deleted 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, 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 describe an efficient C implementation of the FP-growth algorithm. In Section 2 I briefly review how the transaction database is preprocessed in a way that is common to basically all frequent item set mining algorithms. Section 3 explains how the initial FP-tree is built from the (preprocessed) transaction database, yielding the starting point of the algorithm. The main step is described in Section 4, namely how an FP-tree is projected in order to obtain an FP-tree of the (sub-)database containing the transactions with a specific item (though with this item removed). The projection step is the most costly in the algorithm and thus it is important to find an efficient way of executing it. Section 5 considers how a projected FP-tree may be further pruned using a technique that has been called FP-Bonsai [4]. Such pruning can sometimes shrink the FP-tree considerably and thus lead to much faster projections. Finally, in Section 6 I report experiments with my implementation, comparing it with my implementations [5, 6] of the Apriori [1, 2] and Eclat [10] algorithms.

2. PREPROCESSING

Similar to several other algorithms for frequent item set mining, like, for example, Apriori or Eclat, FP-growth 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 *descending* 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. An *ascending* order leads to a particularly slow operation in my experiments, performing even worse than a random order. (In this respect FP-growth behaves in exactly the opposite way as Apriori, which in my implementation usually runs fastest if items are sorted ascendingly, but in the same way as Eclat, which also profits from items being sorted descendingly.)

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 descendingly, are shown in the middle of this table. 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 each transaction descendingly w.r.t. their frequencies we obtain the reduced database shown in Table 1 on the right.

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

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

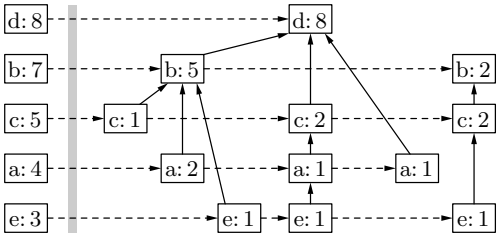

Figure 1: FP-tree for the (reduced) transaction database shown in Table 1.

3. BUILDING THE INITIAL FP-TREE

After all individually infrequent items have been deleted from the transaction database, it is turned into an FP-tree. An FP-tree is basically a prefix tree for the transactions. That is, each path represents a set of transactions that share the same prefix, each node corresponds to one item. In addition, all nodes referring to the same item are linked together in a list, so that all transactions containing a specific item can easily be found and counted by traversing this list. The list can be accessed through a head element, which also states the total number of occurrences of the item in the database. As an example, Figure 1 shows the FP-tree for the (reduced) database shown in Table 1 on the right. The head elements of the item lists are shown to the left of the vertical grey bar, the prefix tree to the right of it.

In my implementation the initial FP-tree is built from a main memory representation of the (preprocessed) transaction database as a simple list of integer arrays. This list is sorted lexicographically (thus respecting the order of the items in the transactions, which reflects their frequency). The sorted list can easily be turned into an FP-tree with a straightforward recursive procedure: at recursion depth k, the k-th item in each transaction is used to split the database into sections, one for each item. For each section a node of the FP-tree is created and labeled with the item corresponding to the section. Each section is then processed recursively, split into subsections, a new layer of nodes (one per subsection) is created etc. Note that in doing so one has to take care that transactions that are only as long as the current recursion depth are handled appropriately, that is, are removed from the section before going into recursion.

Of course, this is not the only way in which the initial FP-tree can be built. At first sight it may seem to be more natural to build it by inserting transaction after transaction into an initially empty FP-tree, creating the necessary nodes for each new transaction. Indeed, such an approach even has the advantage that the transaction database need not be loaded in a simple form (for instance, as a list of integer arrays) into main memory. Since only one transaction is processed at a time, only the FP-tree representation and one new transaction is in main memory. This usually saves space, because an FP-tree is often a much more compact representation of a transaction database.

Nevertheless I decided against such a representation for the following reasons: in order to build a prefix tree by sequentially adding transactions, one needs pointers from parent nodes to child nodes, so that one can descend in the tree according to the items present in the transaction. However, this is highly disadvantageous. As we will see later on, the further processing of an FP-tree, especially the main operation of projecting it, does not need such parent-to-child pointers in my implementation, but rather child-to-parent pointers. Since each node in an FP-tree (with the exception of the roots) has exactly one parent, this, in principle, makes it possible to work with nodes of constant size. If, however, we have to accomodate an array of child pointers per node, the nodes either have variable size or are unnecessarily large (because we have pointers that are not needed), rendering the memory management much less efficient.

It has to be conceded, though, that instead of using an array of child pointers, one may also link all children into a list. This, however, has the severe disadvantage that when inserting transactions into the FP-tree, one such list has to be searched (linearly!) for each item of the transaction in order to find the child to go to—a possibly fairly costly operation.

In contrast to this, first loading the transaction database as a simple list of integer arrays, sorting it, and building the FP-tree with a recursive function (as outlined above), makes it possible to do without parent-to-child pointers entirely. Since the FP-tree is built top down, the parent is already known when the children are created. Thus it can be passed down in the recursion, where the parent pointers of the children are set directly. As a consequence, the nodes of the FP-tree can be kept very small. In my implementation, an FP-tree node contains only fields for (1) an item identifier, (2) a counter, (3) a pointer to the parent node, (4) a pointer to the successor node (referring to the same item) and (5) an auxiliary pointer that is used when projecting the FP-tree (see below). That is, an FP-tree node needs only 20 bytes (on a 32 bit machine).

However, if we used the standard memory management, allocating a block of memory for each node, there would be an additional overhead of 4 to 12 bytes (depending on the memory system implementation) for each node for bookkeeping purposes (for instance, for storing the size of the memory block). In addition, allocating and deallocating a large number of such small memory blocks is usually not very efficient. Therefore I use a specialized memory management in my implementation, which makes it possible to efficiently handle large numbers of equally sized small mem-

ory objects. The idea is to allocate larger arrays (with several thousand elements) of these objects and to organize the elements into a "free" list (i.e., a list of available memory blocks of equal size). With such a system allocating and deallocating FP-tree nodes gets very efficient: the former retrieves (and removes) the first element of the free list, the latter adds the node to deallocate at the beginning of the free list. As experiments showed, introducing this specialized memory management led to a considerable speed-up.

4. PROJECTING AN FP-TREE

The core operation of the FP-growth algorithm is to compute an FP-tree of a projected database, that is, a database of the transactions containing a specific item, with this item removed. This projected database is processed recursively, remembering that the frequent item sets found in the recursion share the removed item as a prefix.

My implementation of the FP-growth algorithm contains two different projection methods, both of which proceed by copying certain nodes of the FP-tree that are identified by the deepest level of the FP-tree, thus producing a kind of "shadow" of it. The copied nodes are then linked and detached from the original FP-tree, yielding an FP-tree of the projected database. Afterwards the deepest level of the original FP-tree, which corresponds to the item on which the projection was based, is removed, and the next higher level is processed in the same way. The two projections methods differ mainly in the order in which they traverse and copy the nodes of the FP-tree (branchwise vs. levelwise).

The first method is illustrated in Figure 2 for the example FP-tree shown in Figure 1. The red arrows show the flow of the processing and the blue "shadow" FP-tree is the created projection. In an outer loop, the lowest level of the FP-tree, that is, the list of nodes corresponding to the projection item, is traversed. For each node of this list, the parent pointers are followed to traverse all ancestors up to the root. Each encountered ancestor is copied and linked from its original (this is what the auxiliary pointer in each node, which was mentioned above, is needed for). During the copying, the parent pointers of the copies are set, the copies are also organized into level lists, and a sum of the counter values in each node is computed in head elements for these lists (these head elements are omitted in Figure 2).

Note that the counters in the copied nodes are determined only from the counters in the nodes on the deepest level, which are propagated upwards, so that each node receives the sum of its children. Note also that due to this we cannot stop following the chain of ancestors at a node that has already been copied, even though it is clear that in this case all ancestors higher up in the FP-tree must already have been copied. The reason is that one has to update the number of transactions in the copies, adding the counter value from the current branch to all copies of the ancestors on the path to the root. This is what the second projection method tries to improve upon.

In a second traversal of the same branches, carried out in exactly the same manner, the copies are detached from their originals (the auxiliary pointers are set to null), which yields the independent projected FP-tree shown in Figure 3. This

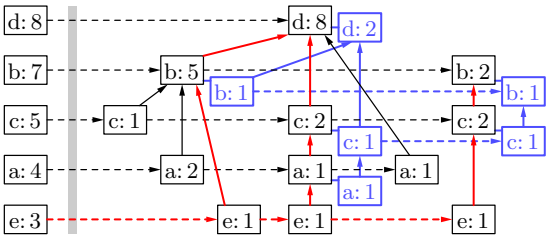

Figure 2: Computing a projection of the database w.r.t. the item e by traversing the lowest level and following all paths to the root.

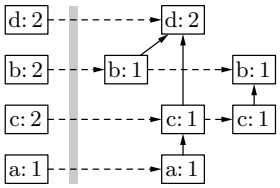

Figure 3: Resulting projected FP-tree after it has been detached from the original FP-tree.

FP-tree is then processed recursively with the prefix e. Note, however, that in this FP-tree all items are infrequent (and thus all item sets containing item e and one other item are infrequent). Hence in this example, no recursive processing would take place. This is, of course, due to the chosen example database and the support threshold.

The second projection method also traverses, in an outer loop, the deepest level of the FP-tree. However, it does not follow the chain of parent pointers up the root, but only copies the parent of each node, not its higher ancestors. In doing so, it also copies the parent pointers of the original FP-nodes, thus making it possible to find the ancestors in later steps. These later steps consist in traversing the levels of the (partially constructed) "shadow" FP-tree (not the levels of the original one!) from bottom to top. On each level the parents of the copied nodes (which are nodes in the original tree) are determined and copied, and the parent pointers of the copies are set. That is, instead of branch by branch, the FP-tree is rather constructed level by level (even though in each step nodes on several levels may be created). The advantage of this method over the one described above is that for branches that share a path close to the root, this common path has to be traversed only once with this method (as the counters for all branches are summed before they are passed to the next higher level). However, the experiments reported below show that the first method is superior in practice. As it seems, the additional effort needed for temporarily setting another parent etc. more than outweighs the advantage of the better combination of the counter values.

5. PRUNING A PROJECTED FP-TREE

After we obtained an FP-tree of a projected database, we may carry out an additional pruning step in order to simplify the tree, thus speeding up projections. I got this idea from [4], which introduces pruning techniques in a slightly different context than pure frequent item set mining (suffice it to say that there are additional constraints). One of these techniques, however, can nevertheless be used here,

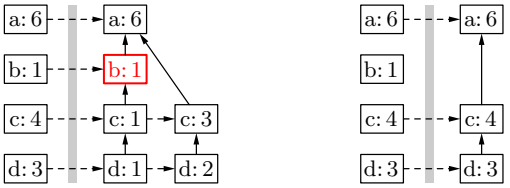

Figure 4: α-pruning of a (projected) FP-tree.

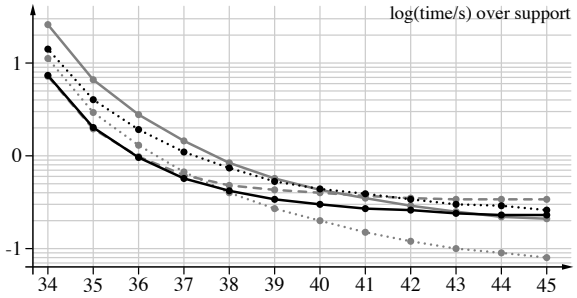

Figure 5: Results on BMS-Webview-1

namely the so-called α-pruning. The idea of this pruning is illustrated with a very simple example in Figure 4. Suppose that the FP-tree shown on the left resulted from a projection and that the minimum support is either 2 or 3. Then item b is infrequent and is not needed in projections. However, it gives rise to a branching in the tree. Hence, by removing it, the tree can be simplified and actually turned into a simple list, as it is shown on the right in Figure 4.

This pruning is achieved by traversing the levels of the FP-tree from top to bottom. The processing starts at the level following the first level that has a non-vanishing support less than the minimum support. (Items having vanishing support can be ignored, because they have no nodes in the FP-tree.) This level and the following ones are traversed and for each node the first ancestor with an item having sufficient support is determined. The parent pointer is then updated to this ancestor, bypassing the nodes corresponding to infrequent items. If by such an operation neighboring nodes receive the same parent, they are merged. They are also merged, if their parents were different originally, but have been merged in a preceding step. As an illustration consider the example Figure 4: after item b is removed, the two nodes for item c can be merged. This has to be recognized in order to merge the two nodes for item d also.

6. EXPERIMENTAL RESULTS

I ran experiments on the same five data sets that I already used in [5, 6], namely BMS-Webview-1 [9], 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 Relim. All experiments were rerun to ensure that the results are comparable.

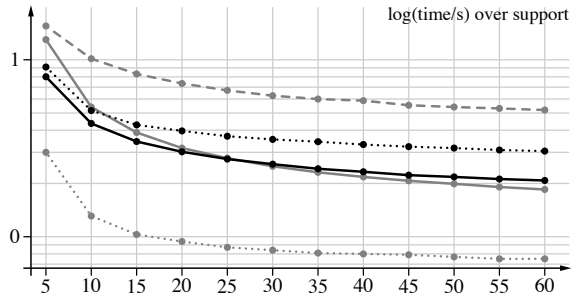

Figure 6: Results on T10I4D100K

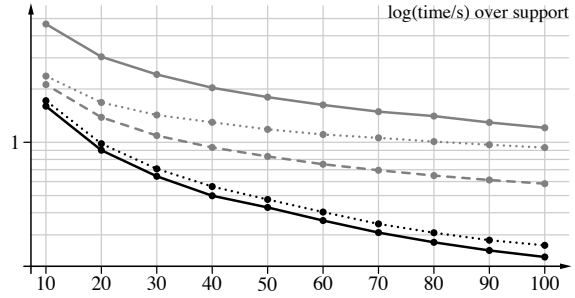

Figure 7: Results on census

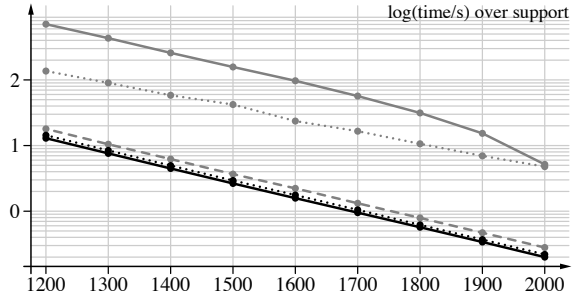

Figure 8: Results on chess

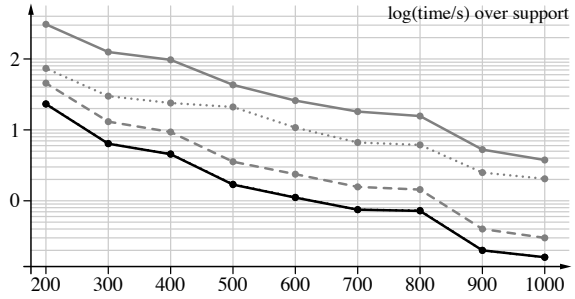

Figure 9: Results on mushroom

Figures 5 to 9 show, each for one of the five data sets, the decimal logarithm of the execution time over different (absolute) minimum support values. The solid black line refers to the implementation of the FP-growth algorithm described here, the dotted black line to the version that uses the alternative projection method. The grey lines represent the corresponding results for Apriori (solid line), Eclat (dashed line), and Relim (dotted line).[1]

Among these implementations, all of which are highly optimized, FP-growth clearly performs best. With the exception of the artificial dataset T10I4D100K, on which it is bet by a considerable margin by Relim, and for higher support values on BMS-Webview-1, where Relim also performs slightly better (presumably, because it does not need to construct a prefix tree), FP-growth is the clear winner. Only on chess, Eclat can come sufficiently close to be called competitive.

The second projection methods for FP-trees (dotted black line) generally fares worse, although there is not much difference between the two methods on chess and mushroom. This is a somewhat surprising result, because there are good reasons to believe that the second projection method may be able to yield better results than the first. I plan to examine this issue in more detail in the future.

7. CONCLUSIONS

In this paper I described an implementation of the FP-growth algorithm, which contains two methods for efficiently projecting an FP-tree—the core operation of the FP-growth algorithm. As the experimental results show, this implementation clearly outperforms Apriori and Eclat, even in highly optimized versions. However, the performance of the two projection methods, especially, why the second is sometimes much slower than the first, needs further investigation.

8. PROGRAM

The implementation of the FP-growth 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 Relim are also available as well as a graphical user interface (written in Java) for finding association rules with Apriori.

9. 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: [7], 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] F. Bonchi and B. Goethals. FP-Bonsai: the Art of Growing and Pruning Small FP-trees. *Proc. 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04, Sydney, Australia)*, 155–160. Springer-Verlag, Heidelberg, Germany 2004

[5] 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/

[6] 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/

[7] 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

[8] 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

[9] 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.

[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

[1] Relim is described in a sibling paper that has also been submitted to this workshop.