Efficient Implementations of Apriori and Eclat

Christian Borgelt

Department of Knowledge Processing and Language Engineering
School of Computer Science, Otto-von-Guericke-University of Magdeburg
Universitätsplatz 2, 39106 Magdeburg, Germany
Email: borgelt@iws.cs.uni-magdeburg.de

Abstract

Apriori and Eclat are the best-known basic algorithms for mining frequent item sets in a set of transactions. In this paper I describe implementations of these two algorithms that use several optimizations to achieve maximum performance, w.r.t. both execution time and memory usage. The Apriori implementation is based on a prefix tree representation of the needed counters and uses a doubly recursive scheme to count the transactions. The Eclat implementation uses (sparse) bit matrices to represent transactions lists and to filter closed and maximal item sets.

1. Introduction

Finding frequent item sets in a set of transactions is a popular method for so-called market basket analysis, which aims at finding regularities in the shopping behavior of customers of supermarkets, mail-order companies, on-line shops etc. In particular, it is tried to identify sets of products that are frequently bought together.

The main problem of finding frequent item sets, i.e., item sets that are contained in a user-specified minimum number of transactions, is that there are so many possible sets, which renders naïve approaches infeasible due to their unacceptable execution time. Among the more sophisticated approaches two algorithms known under the names of Apriori [1, 2] and Eclat [8] are most popular. Both rely on a top-down search in the subset lattice of the items. An example of such a subset lattice for five items is shown in figure 1 (empty set omitted). The edges in this diagram indicate subset relations between the different item sets.

To structure the search, both algorithms organize the subset lattice as a prefix tree, which for five items is shown in Figure 2. In this tree those item sets are combined in a node which have the same prefix w.r.t. to some arbitrary, but fixed order of the items (in the five items example, this order is simply a, b, c, d, e). With this structure, the item sets contained in a node of the tree can be constructed easily in the following way: Take all the items with which the edges leading to the node are labeled (this is the common prefix) and add an item that succeeds, in the fixed order of the items, the last edge label on the path. Note that in this way we need only one item to distinguish between the item sets represented in one node, which is relevant for the implementation of both algorithms.

The main differences between Apriori and Eclat are how they traverse this prefix tree and how they determine the *support* of an item set, i.e., the number of transactions the item set is contained in. Apriori traverses the prefix tree in breadth first order, that is, it first checks item sets of size 1, then item sets of size 2 and so on. Apriori determines the support of item sets either by checking for each candidate item set which transactions it is contained in, or by traversing for a transaction all subsets of the currently processed size and incrementing the corresponding item set counters. The latter approach is usually preferable.

Eclat, on the other hand, traverses the prefix tree in depth first order. That is, it extends an item set prefix until it reaches the boundary between frequent and infrequent item sets and then backtracks to work on the next prefix (in lexicographic order w.r.t. the fixed order of the items). Eclat determines the support of an item set by constructing the list of identifiers of transactions that contain the item set. It does so by intersecting two lists of transaction identifiers of two item sets that differ only by one item and together form the item set currently processed.

2. Apriori Implementation

My Apriori implementation uses a data structure that directly represents a prefix tree as it is shown in figure 2. This tree is grown top-down level by level, pruning those branches that cannot contain a frequent item set [4].

2.1. Node Organization

There are different data structures that may be used for the nodes of the prefix tree. In the first place, we may use simple vectors of integer numbers to represent the counters for the item sets. The items (note that we only need one item to distinguish between the counters of a node, see above) are not explicitly stored in this case, but are implicit in the vector index. Alternatively, we may use vectors, each element of which consists of an item identifier (an integer number) and a counter, with the vector elements being sorted by the item identifier.

The first structure has the advantage that we do not need any memory to store the item identifiers and that we can very quickly find the counter for a given item (simply use the item identifier as an index), but it has the disadvantage that we may have to add "unnecessary" counters (i.e., counters for item sets, of which we know from the information gathered in previous steps that they must be infrequent), because the vector may not have "gaps". This problem can only partially be mitigated by enhancing the vector with an offset to the first element and a size, so that unnecessary counters at the margins of the vector can be discarded. The second structure has the advantage that we only have the counters we actually need, but it has the disadvantage that we need extra memory to store the item identifiers and that we have to carry out a binary search in order to find the counter corresponding to a given item.

A third alternative would be to use a hash table per node. However, although this reduces the time needed to access a counter, it increases the amount of memory needed, because for optimal performance a hash table must not be too full. In addition, it does not allow us to exploit easily the order of the items in the counting process (see below). Therefore I do not consider this alternative here.

Obviously, if we want to optimize speed, we should choose simple counter vectors, despite the gap problem. If we want to optimize memory usage, we can decide dynamically, which data structure is more efficient in terms of memory, accepting the higher counter access time due to the binary search if necessary.

It should also be noted that we need a set of child pointers per node, at least for all levels above the currently added one (in order to save memory, one should not create child pointers before one is sure that one needs them). For organizing these pointers there are basically the same options as for organizing the counters. However, if the counters have item identifiers attached, there is an additional possibility: We may draw on the organization of the counters, using the same order of the items and leaving child pointers nil if they are not needed. This can save memory, even though we may have unnecessary nil pointers, because we do not have to store item identifiers a second time.

2.2. Item Coding

It is clear that the way in which the items are coded (i.e., are assigned integer numbers as identifiers) can have a significant impact on the gap problem for pure counter vectors mentioned above. Depending on the coding we may need large vectors with a lot of gaps or we may need only short vectors with few gaps. A good heuristic approach to minimize the number and the size of gaps seems to be this: It is clear that frequent item sets contain items that are frequent individually. Therefore it is plausible that we have only few gaps if we sort the items w.r.t. their frequency, so that the individually frequent items receive similar identifiers if they have similar frequency (and, of course, infrequent items are discarded entirely). In this case it can be hoped that the offset/size representation of a counter vector can eliminate the greater part of the unnecessary counters, because these can be expected to cluster at the vector margins.

Extending this scheme, we may also consider to code the items w.r.t. the number of frequent pairs (or even triples etc.)

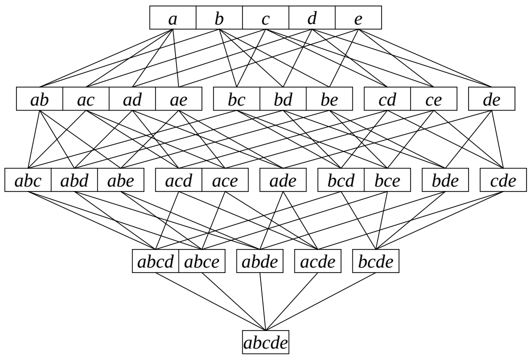

Figure 1. A subset lattice for five items (empty set omitted).

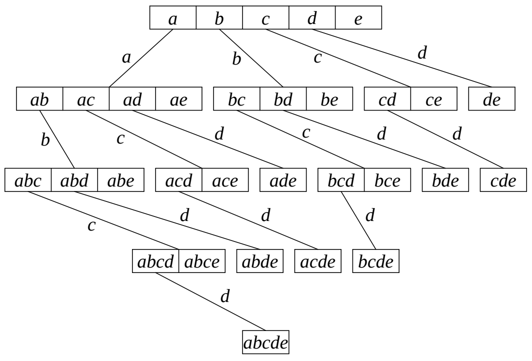

Figure 2. A prefix tree for five items (empty set omitted).

they are part of, thus using additional information from the second (or third etc.) level to improve the coding. This idea can most easily be implemented for item pairs by sorting the items w.r.t. the sum of the sizes of the transactions they are contained in (with infrequent items discarded from the transactions, so that this sum gives a value that is similar to the number of frequent pairs, which, as these are heuristics anyway, is sufficient).

2.3. Recursive Counting

The prefix tree is not only an efficient way to store the counters, it also makes processing the transactions very simple, especially if we sort the items in a transaction ascendingly w.r.t. their identifiers. Then processing a transaction is a simple doubly recursive procedure: To process a transaction for a node of the tree, (1) go to the child corresponding to the first item in the transaction and process the remainder of the transaction recursively for that child and (2) discard the first item of the transaction and process it recursively for the node itself (of course, the second recursion is more easily implemented as a simple loop through the transaction). In a node on the currently added level, however, we increment a counter instead of proceeding to a child node. In this way on the current level all counters for item sets that are part of a transaction are properly incremented.

By sorting the items in a transaction, we can also apply the following optimizations (this is a bit more difficult—or needs additional memory—if hash tables are used to organize the counters and thus explains why I am not considering hash tables): (1) We can directly skip all items before the first item for which there is a counter in the node, and (2) we can abort the recursion if the first item of (the remainder of) a transaction is beyond the last one represented in the node. Since we grow the tree level by level, we can even go a step further: We can terminate the recursion once (the remainder of) a transaction is too short to reach the level currently added to the tree.

2.4. Transaction Representation

The simplest way of processing the transactions is to handle them individually and to apply to each of them the recursive counting procedure described in the preceding section. However, the recursion is a very expensive procedure and therefore it is worthwhile to consider how it can be improved. One approach is based on the fact that often there are several similar transactions, which lead to a similar program flow when they are processed. By organizing the transactions into a prefix tree (an idea that has also been used in [6] in a different approach) transactions with the same prefix can be processed together. In this way the procedure for the prefix is carried out only once and thus considerable performance gains can result. Of course, the gains have to outweigh the additional costs of constructing such a transaction tree to lead to an overall gain.

2.5. Transaction Filtering

It is clear that in order to determine the counter values on the currently added level of the prefix tree, we only need the items that are contained in those item sets that are frequent on the preceding level. That is, to determine the support of item sets of size k, we only need those items that are contained in the frequent item sets of size $k-1$. All other items can be removed from the transactions. This has the advantage that the transactions become smaller and thus can be counted more quickly, because the size of a transaction is a decisive factor for the time needed by the recursive counting scheme described above.

However, this can only be put to work easily if the transactions are processed individually. If they are organized as a prefix tree, a possibly costly reconstruction of the tree is necessary. In this case one has to decide whether to continue with the old tree, accepting the higher counting costs resulting from unnecessary items, or whether rebuilding the tree is preferable, because the costs for the rebuild are outweighed by the savings resulting from the smaller and simpler tree. Good heuristics seem to be to rebuild the tree if

$$\frac{n_{\mathrm{new}}}{n_{\mathrm{curr}}} \frac{t_{\mathrm{tree}}}{t_{\mathrm{count}}} < 0.1,$$

where n_{curr} is the number of items in the current transaction tree, n_{new} is the number of items that will be contained in the new tree, t_{tree} is the time that was needed for building the current tree and t_{count} is the time that was needed for counting the transactions in the preceding step. The constant 0.1 was determined experimentally and on average seems to lead to good results (see also Section 4).

2.6. Filtering Closed and Maximal Item Sets

A frequent item set is called *closed* if there is no superset that has the same support (i.e., is contained in the same number of transactions). Closed item sets capture all information about the frequent item sets, because from them the support of any frequent item set can be determined.

A frequent item set is called *maximal* if there is no superset that is frequent. Maximal item sets define the boundary between frequent and infrequent sets in the subset lattice.

Any frequent item set is often also called a *free* item set to distinguish it from closed and maximal ones.

In order to find closed and maximal item sets with Apriori one may use a simple filtering approach on the prefix tree: The final tree is traversed top-down level by level (breadth first order). For each frequent item set all subsets with one item less are traversed and marked as not to be reported if they have the same support (closed item sets) or unconditionally (maximal item sets).

3. Eclat Implementation

My Eclat implementation represents the set of transactions as a (sparse) bit matrix and intersects rows to determine the support of item sets. The search follows a depth first traversal of a prefix tree as it is shown in Figure 2.

3.1. Bit Matrices

A convenient way to represent the transactions for the Eclat algorithm is a bit matrix, in which each row corresponds to an item, each column to a transaction (or the other way round). A bit is set in this matrix if the item corresponding to the row is contained in the transaction corresponding to the column, otherwise it is cleared.

There are basically two ways in which such a bit matrix can be represented: Either as a true bit matrix, with one memory bit for each item and transaction, or using for each row a list of those columns in which bits are set. (Obviously the latter representation is equivalent to using a list of transaction identifiers for each item.) Which representation is preferable depends on the density of the dataset. On 32 bit machines the true bit matrix representation is more memory efficient if the ratio of set bits to cleared bits is greater than 1:31. However, it is not advisable to rely on this ratio in order to decide between a true and a sparse bit matrix representation, because in the search process, due to the intersections carried out, the number of set bits will decrease. Therefore a sparse representation should be used even if the ratio of set bits to cleared bits is greater than 1:31. In my current implementation a sparse representation is preferred if the ratio is greater than 1:7, but this behavior can be changed by a user.

A more sophisticated option would be to switch to the sparse representation of a bit matrix during the search once the ratio of set bits to cleared bits exceeds 1:31. However, such an automatic switch, which involves a rebuild of the bit matrix, is not implemented in the current version.

3.2. Search Tree Traversal

As already mentioned, Eclat searches a prefix tree like the one shown in Figure 2 in depth first order. The transition of a node to its first child consists in constructing a new bit matrix by intersecting the first row with all following rows. For the second child the second row is intersected with all following rows and so on. The item corresponding to the row that is intersected with the following rows thus is added to form the common prefix of the item sets processed in the corresponding child node. Of course, rows corresponding to infrequent item sets should be discarded from the constructed matrix, which can be done most conveniently if we store with each row the corresponding item identifier rather than relying on an implicit coding of this item identifier in the row index.

Intersecting two rows can be done by a simple logical *and* on a fixed length integer vector if we work with a true bit matrix. During this intersection the number of set bits in the intersection is determined by looking up the number of set bits for given word values (i.e., 2 bytes, 16 bits) in a precomputed table. For a sparse representation the column indices for the set bits should be sorted ascendingly for efficient processing. Then the intersection procedure is similar to the merge step of merge sort. In this case counting the set bits is straightforward.

3.3. Item Coding

As for Apriori the way in which items are coded has an impact on the execution time of the Eclat algorithm. The reason is that the item coding not only affects the number and the size of gaps in the counter vectors for Apriori, but also the structure of the *pruned* prefix tree and thus the structure of Eclat's search tree. Sorting the items usually leads to a better structure. For the sorting there are basically the same options as for Apriori (see Section 2.2).

3.4. Filtering Closed and Maximal Item Sets

Determining closed and maximal item sets with Eclat is slightly more difficult than with Apriori, because due to the backtrack Eclat "forgets" everything about a frequent item set once it is reported. In order to filter for closed and maximal item sets, one needs a structure that records these sets, and which allows to determine quickly whether in this structure there is an item set that is a superset of a newly found set (and whether this item set has the same support if closed item sets are to be found).

In my implementation I use the following approach to solve this problem: Frequent item sets are reported in a node of the search tree *after* all of its child nodes have been processed. In this way it is guaranteed that all possible supersets of an item set that is about to be reported have already been processed. Consequently, we can maintain a repository of already found (closed or maximal) item sets and only have to search this repository for a superset of the item set in question. The repository can only grow (we never have to remove an item set from it), because due to the report order a newly found item set cannot be a superset of an item set in the repository.

For the repository one may use a bit matrix in the same way as it is used to represent the transactions: Each row corresponds to an item, each column to a found (closed or maximal) frequent item set. The superset test consists in intersecting those rows of this matrix that correspond to the items in the frequent item set in question. If the result is empty, there is no superset in the repository, otherwise there is (at least) one. (Of course, the intersection loop is terminated as soon as an intersection gets empty.)

To include the information about the support for closed item sets, an additional row of the matrix is constructed, which contains set bits in those columns that correspond to item sets having the same support as the one in question. With this additional row the intersection process is started.

It should be noted that the superset test can be avoided if any direct descendant (intersection product) of an item set has the same support (closed item sets) or is frequent (maximal item set).

In my implementation the repository bit matrix uses the same representation as the matrix that represents the transactions. That is, either both are true bit matrices or both are sparse bit matrices.

4. Experimental Results

I ran experiments with both programs on five data sets, which exhibit different characteristics, so that the advantages and disadvantages of the two approaches and the different optimizations can be observed. The data sets I used are: BMS-Webview-1 (a web click stream from a leg-care company that no longer exists, which has been used in the KDD cup 2000 [7, 9]), T10I4D100K (an artificial data set generated with IBM's data generator [10]), census (a data set derived from an extract of the US census bureau data of 1994, which was preprocessed by discretizing numeric attributes), chess (a data set listing chess end game positions for king vs. king and rook), and mushroom (a data set describing poisonous and edible mushrooms by different attributes). The last three data sets are available from the UCI machine learning repository [3]. The discretization of the numeric attributes in the census data set was done with a shell/gawk script that can be found on the WWW page mentioned below. For the experiments I used an AMD Athlon XP 2000+ machine with 756 MB main memory running S.u.S.E. Linux 8.2 and gcc version 3.3.

The results for these data sets are shown in Figures 3 to 7. Each figure consists of five diagrams, a to e, which are organized in the same way in each figure. Diagram a shows the decimal logarithm of the number of free (solid), closed (short dashes), and maximal item sets (long dashes) for different support values. From these diagrams it can already be seen that the data sets have clearly different characteristics. Only census and chess appear to be similar.

Diagrams b and c show the decimal logarithm of the execution time in seconds for different parameterizations of Apriori (diagram b) and Eclat (diagram c). To ease the comparison of the two diagrams, the default parameter curve for the other algorithm (the solid curve in its own diagram) is shown in grey in the background.

The curves in diagram b represent the following settings:

solid: Items sorted ascendingly w.r.t. the sum of the sizes of the transactions they are contained in; prefix tree to represent the transactions, which is rebuild every time the heuristic criterion described in section 2.5 is fulfilled.

short dashes: Like solid curve, prefix tree used to represent the transactions, but never rebuild.

long dashes: Like solid curve, but transactions are *not* organized as a prefix tree; items that are no longer needed are *not* removed from the transactions.

dense dots: Like long dash curve, but items sorted ascendingly w.r.t. their frequency in the transactions.

In diagram b it is not distinguished whether free, closed, or maximal item sets are to be found, because the time for filtering the item sets is negligible compared to the time needed for counting the transactions (only a small difference would be visible in the diagrams, which derives mainly from the fact that less time is needed to write the smaller number of closed or maximal item sets).

In diagram c the solid, short, and long dashes curve show the results for free, closed, and maximal item sets, respectively, with one representation of the bit matrix, the dense dots curve the results for free item sets for the other representation (cf. section 3.1). Whether the solid, short, and long dashes curve refer to a true bit matrix and the dense dots curve to a sparse one or the other way round depends on the data set and is indicated in the corresponding section below.

Diagrams d and e show the decimal logarithm of the memory in bytes used for different parameterizations of Apriori (diagram d) and Eclat (diagram e). Again the grey curve refers to the default parameter setting of the other algorithm (the solid curve in its own diagram).

The curves in diagram d represent the following settings:

solid: Items sorted ascendingly w.r.t. the sum of the sizes of the transaction they are contained in; transactions organized as a prefix tree; memory saving organization of the prefix tree nodes as described in section 2.1.

short dashes: Like solid, but *no* memory saving organization of the prefix tree nodes (always pure vectors).

long dashes: Like short dashes, but items sorted *descendingly* w.r.t. the sum of the sizes of the transaction they are contained in.

dense dots: Like long dashes, but items *not* sorted.

Again it is not distinguished whether free, closed, or maximal item sets are to be found, because this has no influence on the memory usage. The meaning of the line styles in diagram e is the same as in diagram c (see above).

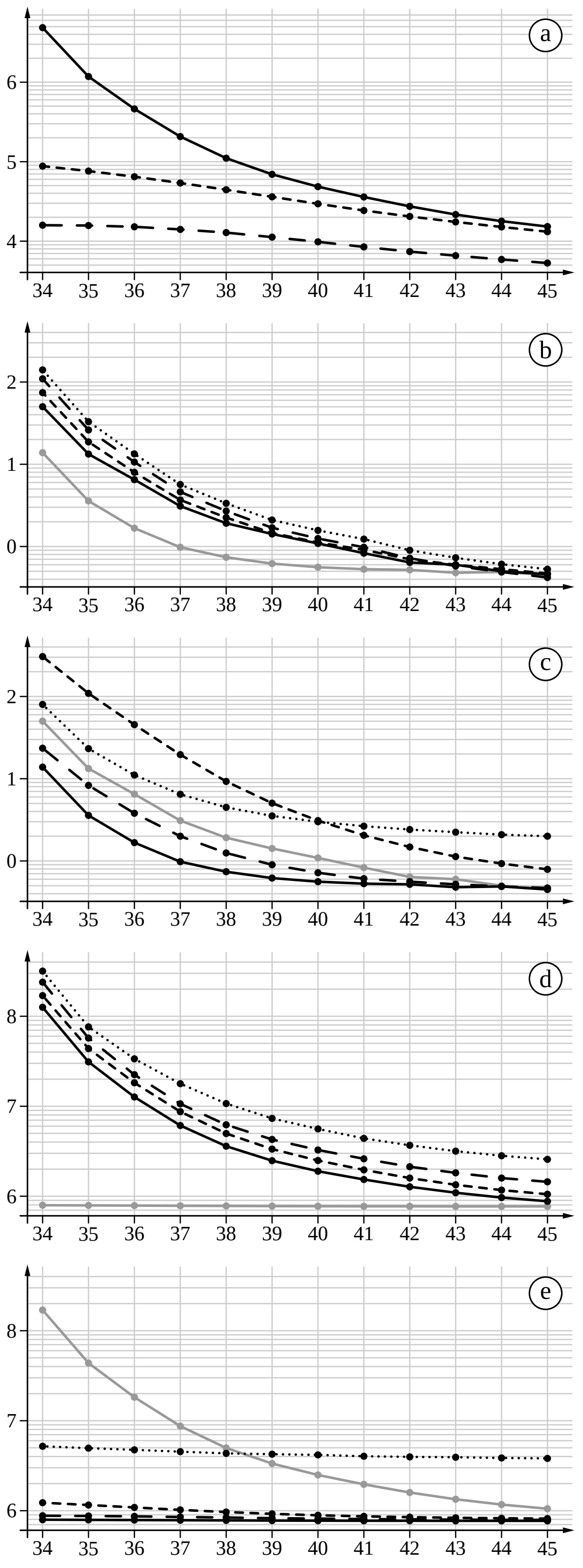

Figure 3. Results on BMS-Webview-1

BMS-Webview-1: Characteristic for this data set is the divergence of the number of free, closed, and maximal item sets for lower support values. W.r.t. the execution time of Apriori this data set shows perfectly the gains that can result from the different optimizations. Sorting the items w.r.t. the sum of transactions sizes (long dashes in diagram b) improves over sorting w.r.t. simple frequency (dense dots), organizing the transactions as a prefix tree (short dashes) improves further, removing no longer needed items yields another considerable speed-up (solid curve). However, for free and maximal item sets and a support less than 44 transactions Eclat with a sparse bit matrix representation (long dashes and solid curve in diagram c) is clearly better than Apriori, which also needs a lot more memory. Only for closed item sets Apriori is the method of choice (Eclat: short dashes in diagram c), which is due to the more expensive filtering with Eclat. Using a true bit matrix with Eclat is clearly not advisable as it performs worse than Apriori and down to a support of 39 transactions even needs more memory (dense dots in diagrams c and e).

T10I4D100K: The numbers of all three types of item sets sharply increase for lower support values; there is no divergence as for BMS-Webview-1. For this data set Apriori outperforms Eclat, although for a support of 5 transactions Eclat takes the lead for free item sets. For closed and maximal item sets Eclat cannot challenge Apriori. It is remarkable that for this data set rebuilding the prefix tree for the transactions in Apriori slightly degrades performance (solid vs. short dashes in diagram b, with the dashed curve almost covered by the solid one). For Eclat a sparse bit matrix representation (solid, short, and long dashes curve in diagrams c and e) is preferable to a true bit matrix (dense dots). (Remark: In diagram b the dense dots curve is almost identical to the long dashes curve and thus is covered.)

Census: This data set is characterized by an almost constant ratio of the numbers of free, closed, and maximal item sets, which increase not as sharply as for T10I4D100K. For free item sets Eclat with a sparse bit matrix representation (solid curve in diagram c) always outperforms Apriori, while it clearly loses against Apriori for closed and maximal item sets (long and short dashes curves in diagrams c and e, the latter of which is not visible, because it lies outside the diagram — the execution time is too large due to the high number of closed item sets). For higher support values, however, using a true bit matrix representation with Eclat to find maximal item sets (sparse dots curves in diagrams c and e) comes close to being competitive with Apriori. Again it is remarkable that rebuilding the prefix tree of transactions in Apriori slightly degrades performance.

Chess: W.r.t. the behavior of the number of free, closed, and maximal item sets this dataset is similar to census, although the curves are bend the other way. The main difference to the results for census are that for this data set a true bit ma-

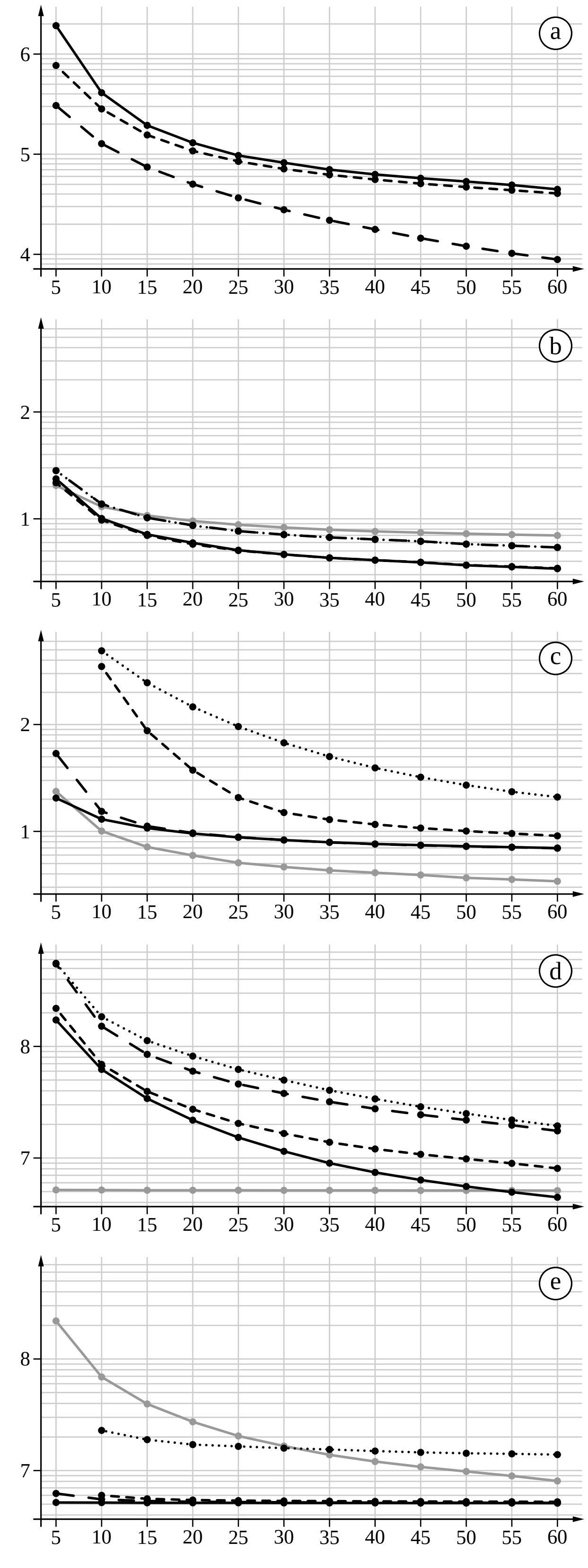

Figure 4. Results on T10I4D100K

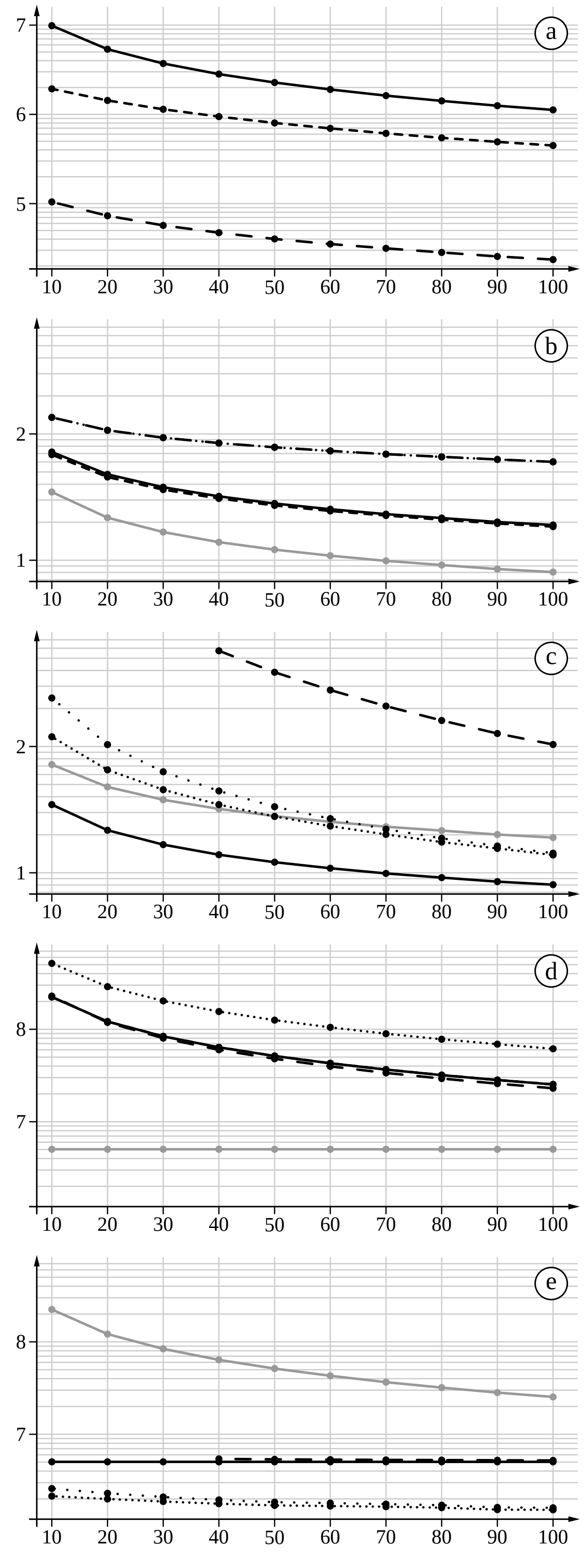

Figure 5. Results on census

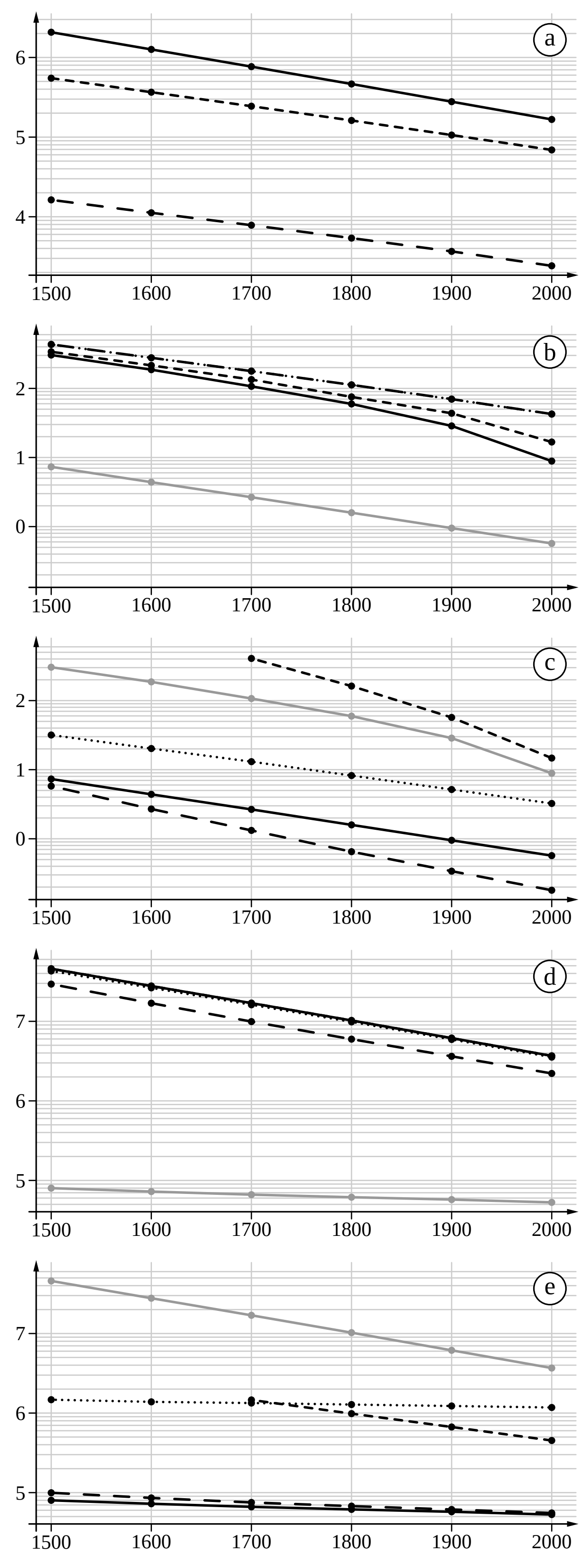

Figure 6. Results on chess

trix representation for Eclat (solid, short, and long dashes curves in diagrams c and e) is preferable to a sparse one (dense dots), while for census it is the other way round. The true bit matrix representation also needs less memory, indicating a very dense data set. Apriori can compete with Eclat only when it comes to closed item sets, where it performs better due to its more efficient filtering of the fairly high number of closed item sets.

Mushroom: This data set differs from the other four in the position of the number of closed data sets between the number of free and maximal item sets. Eclat with a true bit matrix representation (solid, short, and long dashes curves in diagrams c and e) outperforms Eclat with a sparse bit matrix representation (dense dots), which in turn outperforms Apriori. However, the sparse bit matrix (dense dots in diagram c) gains ground towards lower support values, making it likely to take the lead for a minimum support of 100 transactions. Even for closed and maximal item sets Eclat is clearly superior to Apriori, which is due to the small number of closed and maximal item sets, so that the filtering is not a costly factor. (Remark: In diagram b the dense dots curve is almost identical to the long dashes curve and thus is covered. In diagram d the short dashes curve, which lies over the dense dots curve, is covered the solid one.)

5. Conclusions

For free item sets Eclat wins the competition w.r.t. execution time on four of the five data sets and it always wins w.r.t. memory usage. On the only data set on which it loses the competition (T10I4D100K), it takes the lead for the lowest minimum support value tested, indicating that for lower minimum support values it is the method of choice, while for higher minimum support values its disadvantage is almost negligible (note that for this data set all execution times are less than 30s).

For closed item sets the more efficient filtering gives Apriori a clear edge w.r.t. execution time, making it win on all five data sets. For maximal item sets the picture is less clear. If the number of maximal item sets is high, Apriori wins due to its more efficient filtering, while Eclat wins for a lower number of maximal item sets due to its more efficient search.

6. Programs

The implementations of Apriori and Eclat described in this paper (Windows™ and Linux™ executables as well as the source code) can be downloaded free of charge at

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

The special program versions submitted to this workshop rely on the default parameter settings of these programs (solid curves in the diagrams b to e of Section 4).

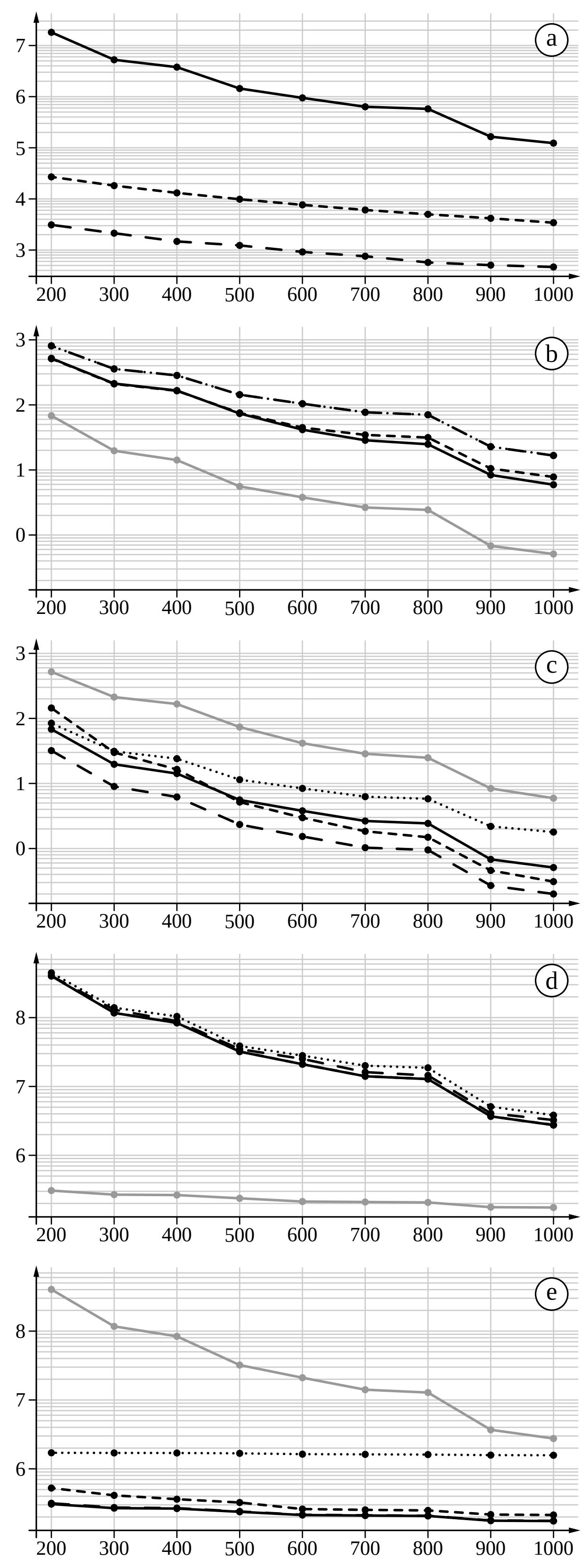

Figure 7. Results on mushroom

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: [5], 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 and R. Kruse. Induction of Association Rules: Apriori Implementation. *Proc. 14th Conf. on Computational Statistics (COMPSTAT)*. Berlin, Germany 2002

[5] 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

[6] 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

[7] 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.

[8] 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

[9] Z. Zheng, R. Kohavi, and L. Mason. Real World Performance of Association Rule Algorithms. In: *Proc. 7th Int. Conf. on Knowledge Discovery and Data Mining (SIGKDD'01)*. ACM Press, New York, NY, USA 2001

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