Mining Fuzzy Frequent Item Sets

Xiaomeng Wang, Christian Borgelt, and Rudolf Kruse

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

Abstract: Due to various reasons transaction data often lack information about some items. This leads to the problem that some potentially interesting frequent item sets cannot be discovered, since by exact matching the number of supporting transactions may be smaller than the user-specified minimum. In this study we try to find such frequent item sets nevertheless by inserting missing items into transactions during the mining process in order to allow approximate matching. We present a recursive elimination algorithm, based on a step by step elimination of items from the transaction database together with a recursive processing of transaction subsets. This algorithm is very simple, works without complicated data structures, and allows us to find fuzzy frequent item sets easily.

Keywords: Frequent Item Set Mining, Fuzzy Frequent Item Sets, Alarm Sequence Analysis

1 Introduction

Methods for mining frequent item sets have been studied extensively. Among the best-known algorithms are Apriori [1, 2], Eclat [11, 4], and FP-growth [7]. With these methods it was investigated how to find frequent patterns efficiently and how to mine data with different characteristics such as dense vs. sparse, long vs. short patterns, memory-based vs. disk-based etc. Algorithms for mining frequent item sets also form the basis of manifold other tasks, for example, mining basket data, mining molecular fragments, analysis of event sequences or web accesses etc. Different applications represent different frameworks and thus focus on different problems.

The framework of our study is alarm sequence analysis in telecommunication networks. A telecommunication network consists of a huge number of interconnected components: switches, exchanges, transmission equipment etc. Each component and software module can produce alarms, which are messages describing some kind of abnormal situation.

A core task of analyzing alarm sequences is to find collections of alarms occurring frequently together—a task that can also be modeled as finding frequent item sets. Unfortunately, the alarms often get delayed, lost, or repeated due to the complex system, which can render this task fairly difficult. In this paper we consider mining fuzzy frequent item sets in such a noisy environment, even though our approach is not limited to this particular application. We present an algorithm to mine data with missing information, which is based on a step by step elimination of items together with a recursive processing of transaction subsets and the insertion of missing items.

We first motivate our study from the considered application domain, i.e. telecommunication alarm analysis. Then we introduce the recursive elimination algorithm for fuzzy frequent item sets based on the insertion of missing items, followed by experimental results we obtained with some test data sets. Finally, we draw conclusions and discuss possible future work.

2 Mining Fuzzy Frequent Item Sets

2.1 Motivation

The problem of finding frequent collections of alarms by analyzing data about alarms that occurred in telecommunication networks is also known as the discovery of frequent episodes in event sequences [8]. In general, an event (here: alarm) sequence can be seen as a sequence of items, in which each item is associated with a time of occurrence. A frequent episode is a collection of events that frequently occur together.

Since the events of an episode should occur close to each other in time, [8] introduced a time window that moves along the event sequence to build a sequence of partially overlapping windows. Each window captures a specific slice of the event sequence. Thus how close in time is "close enough" for a collection of events to qualify as an episode is defined by the width of the time window in which the events must occur.

This way of processing the data has the additional advantage that the problem of finding frequent episodes in event sequences is transformed into the well-studied problem of finding frequent item sets in a set of transactions, because the events in a time window can be treated as a transaction: each event is an item and the support of an episode is the number of windows in which the episode occurred. The mining algorithm used in [8] is based on an Apriori-like candidate generation and test scheme, which relies on exact event matching.

As already mentioned above, it happens quite often in telecommunication networks that some alarms get delayed, repeated, or lost because of noise, transmission errors, failing links etc. Repetition is usually not too much of a problem, since no information is lost, but if alarms do not get through or are delayed, they are missing from the time window its associated alarms occur in. To handle such situations, we rely on the notion of a "fuzzy" frequent item set. However, in contrast to other work on fuzzy association rules, where a fuzzy approach is used to deal with quantitative items/attributes, in this paper the term "fuzzy" means an item set that may not be found exactly in all supporting transactions, but only approximately.

The rationale underlying our approach is as follows: if we do exact matching, a time window that does not contain an event will not be considered as supporting an episode containing that event. However, it may actually support the episode, only the alarm corresponding to the missing event item got lost or was delayed so that it arrived outside the specified time window. As a consequence, the support of a potentially interesting item set may lie below the user-specified minimum support, simply because the complete episode occurs too rarely.

In order to handle such situations we try to "complete" transactions by inserting missing items during the mining process. In this way we allow a certain number of mismatches, by which we account for possibly lost alarms. That is, a transaction still contributes to the support of an item set, though only to a reduced degree, if it contains only a part of the items in the set. How the transaction is weighted if an insertion is necessary to make it match a given item set is described in detail in Section 2.3. However, to convey a better understanding of our approach, we describe it with exact matching first.

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

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

2.2 Recursive Elimination

Recursive elimination [5] (Relim for short) is an algorithm for finding frequent item sets, which uses data structures very similar to those of H-Mine [10], even though it was developed independently and finds the frequent item sets in a different order. Inspired by the FP-growth algorithm, but working without a prefix tree representation, Relim processes the transactions directly, organizing them merely into singly linked lists.

2.2.1 Preprocessing and Data Representation

Recursive elimination preprocesses the transaction database similar to several other algorithms for frequent item set mining, like e.g. Apriori or FP-growth: in an initial scan it determines the frequencies of the items (support of single element item sets). All infrequent items—that is, all items that appear in fewer transactions than a user-specified minimum number—are discarded from the transactions, since they can obviously never be part of a frequent item set. In addition, the items in each transaction are sorted in *ascending* order w.r.t. their frequencies in the database. Although the algorithm does not require this specific order, experiments showed that it leads to much shorter execution times than a random order.

This preprocessing is demonstrated in Table 1, on the left of which is an example transaction database. 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 minimum support of 3 transactions, items f and g can be discarded. After doing so and sorting the items in each transaction ascendingly w.r.t. their frequencies we obtain the reduced database shown in Table 1 on the right.

Relim uses very simple data structures: each transaction is represented as an array 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 implicitly 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 item e, contains the second, seventh and eighth transaction, with the item e removed. The counter in the array element states the number of transactions beginning with the corresponding item. Note 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.

2.2.2 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 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. 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

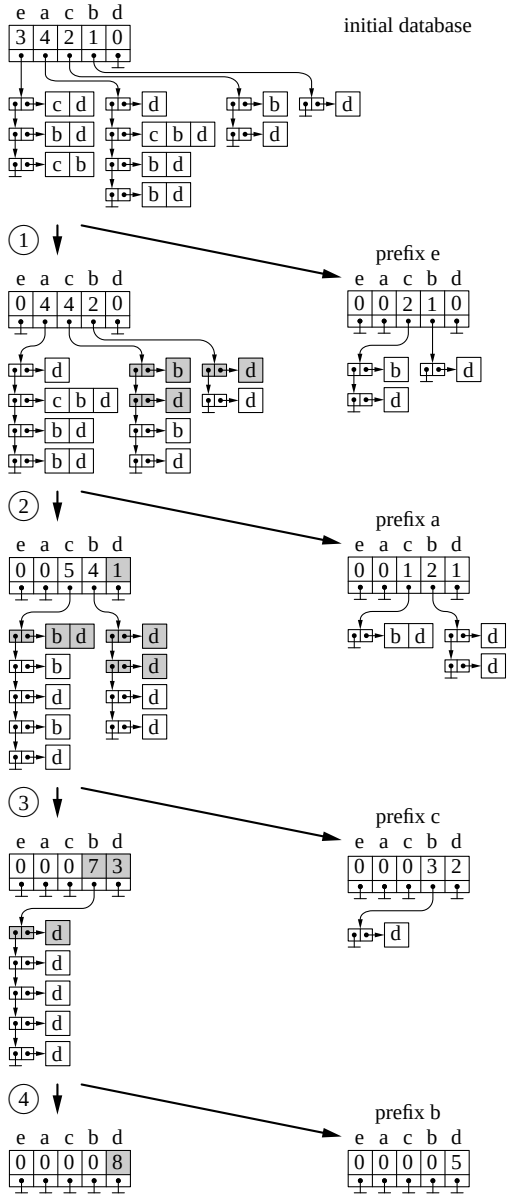

Figure 1: Procedure of the recursive elimination with the modification of the transaction lists (left) as well as the construction of the transaction lists for the recursion (right).

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 is empty after the removal of the leading item. Instead only the counter in the lists array element is incremented as an indicator of such 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.

2.3 Fuzzy Mining

For fuzzy frequent item set mining we extend the recursive elimination scheme presented above in the following ways:

1. **Edit cost**
   The distance of two item sets can be defined as the sum of the costs of the cheapest sequence of edit operations needed to transform one item set into the other [9]. In our case, we only consider "insertion" as the only edit operation. Insertions are very easy to implement with the Relim algorithm as we demonstrate below.

   Note that different items can be associated with different insertion costs. For example, in telecommunication networks different alarms can have a different probability of getting lost. Usually alarms originating in lower levels of the module hierarchy get lost more easily than alarms originating in higher levels. Therefore the former can be associated with lower insertion costs than the latter. The insertion of a certain item may also be completely inhibited by assigning a very high insertion cost.

2. **Transaction weight**
   Each transaction in the original database is associated with a "weight"; the initial weight of each transaction is 1. To store this weight we add a component to each list element mentioned in Section 2.2.2. After each insertion of an item into a transaction, its weight is "penalized" with the cost associated with this insertion.

   For instance, in Table 1 on the right consider the second transaction (ecd), the fifth (cb), the eighth (ecb) and the ninth (cd). If we want to determine the support of the item set "ec", the second and the eighth transaction contribute to the support with a weight of 1 each. However, the fifth transaction (cb) and the ninth (cd) can also be made to contain the item set "ec" if we insert item e into them. Due to this insertion, they should not contribute with full weight, though, but only to some degree. Therefore these two transactions are counted with penalized weights for the support of item set "ec".

Formally, the weighting can be described as follows: if we denote the weight of a transaction by w and represent the cost of inserting an item i by a number $c(i)$, then the new weight w' of the transaction after editing is

$$w' = \mathbf{f}(w, c(i)),$$

where $\mathbf{f}$ is a function that combines the weight before editing and the insertion cost, so that the new weight is penalized by the last insertion. There is a wide variety of combination functions that may be used, for instance, any t-norm. For simplicity, we use multiplication, i.e., $w' = w \cdot c(i)$, but this is a more or less arbitrary choice. Note, however, that in this case lower values for $c(i)$ mean higher costs as they penalize the weight more.

How many insertions into a transaction are allowed can be limited by a user-specified lower bound for the transaction weight. If the weight of a transaction falls below this threshold, it is not considered in the next recursion and thus no insertions can be done on it anymore.

3. **Construction of the subset of transaction lists**
   The recursive processing of the array list that represents a (reduced) transaction database remains basically the same as before. The most important modification lies in the construction of the subset of transaction lists, i.e., inserting the copy of the list elements into an initially empty second array of transaction lists.

   Recall that this second array collects the subset of transactions that contain a specific item. Therefore, to find fuzzy frequent item sets, we copy not only the elements of the transaction list that contain this specific item, but also some elements of the transaction lists associated with other items, into which the item is then inserted. Which elements we have to copy from the other lists is determined as follows: for each transaction not containing the item under consideration the penalized weight after insertion is computed as described above. If this penalized weight exceeds the user-specified minimum weight, the transaction is copied to the new lists array and associated there with the penalized weight.

   Note that with such operations we have more transactions to process in the recursion and thus a (considerably) longer execution time is to be expected.

   Before a transaction list is processed, its support counter in the array element is checked. Note that this support counter is even less an indicator of the number of the elements of the transaction list now, as it states the sum of the weights of list elements, several of which may differ from the initial weight of 1 (cf. Figure 2).

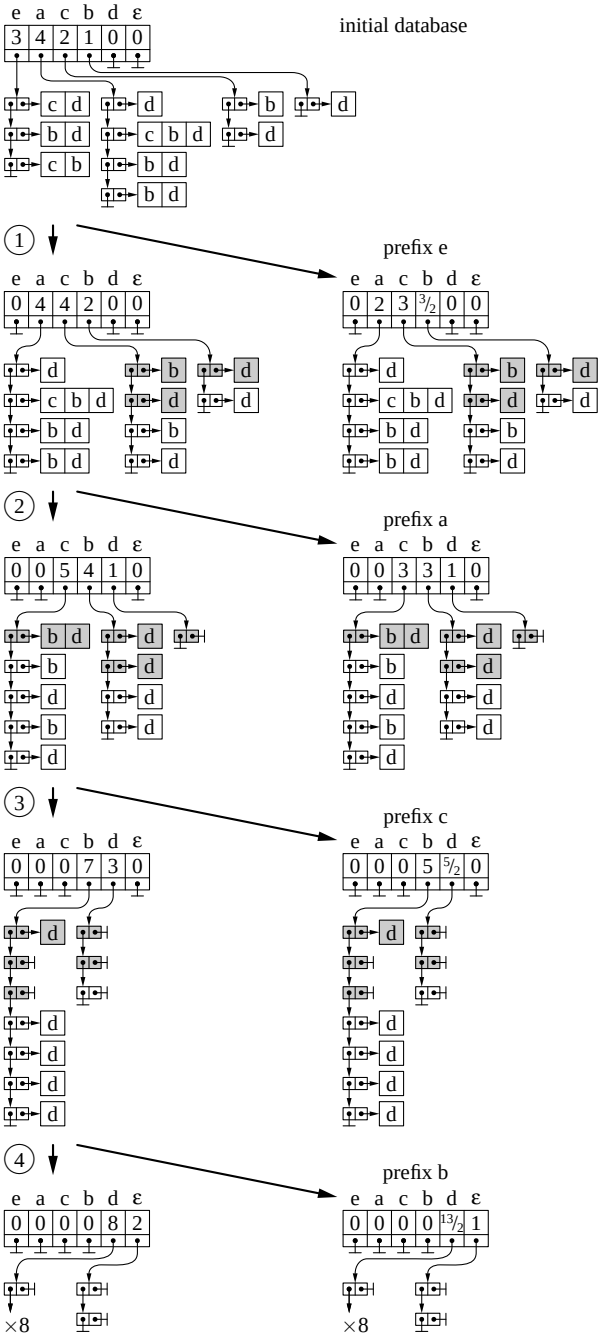

Figure 2: Procedure of the recursive elimination with the insertion of missing items. Since insertions of all items are possible, the subset (right) always has the same structure as the main set (left). The support values, however, differ due to the transaction weighting.

Figure 2 demonstrates the procedure of mining fuzzy frequent item sets by recursive elimination with the insertion of missing items for the root level of the recursion. We use the same transaction database as in Figure 1. In this example we use the same cost factor $c(i) = 0.5$ for all items, and thus the weight is updated after an insertion according to $w' = w \cdot 0.5$.

In the first step all item sets containing the item e are found by processing the leftmost list. Reassigning the elements of this list to the lists on the right is the same as before (see Figure 1 left), but the copies inserted into a second list array are different now (compare Figure 1 right and Figure 2 right): we copy not only the elements of the leftmost list with a weight of 1 (grey list elements), but also the elements of the lists corresponding to items a, c, and b with a penalized weight of 0.5 (white list elements, assuming that 0.5 is greater than the lower bound for the transaction weight). That is, we virtually inserted item e into the corresponding transactions.

Note that the structure of this list array (Figure 2 right), which collects the subset of transactions sharing item e, and that of the main set (Figure 2 left) are the same, since this is the root level of the recursion. In deeper levels, where some

census	number of sets	time/s
Relim	244	0.38
Relx (no insertion)	244	0.47
Relx ($t = 0.4$)	1340	4.58
Relx ($t = 0.2$)	2510	13.14

T10I4D100K	number of sets	time/s
Relim	10	0.01
Relx (no insertion)	10	0.03
Relx ($t = 0.4$)	55	0.14
Relx ($t = 0.2$)	68	0.39

Table 2: Results on census and T10I4D100K.

transactions may be discarded due to the lower bound for the transaction weight, this structural identity may not hold.

Note also that the support values differ for the subset (right) and the main set (left) due to the transaction weighting. Because of the insertion, the support counter of the list corresponding to item a is $4 \cdot 0.5 = 2$ and that of the list corresponding to item c is $(2 \cdot 1) + (2 \cdot 0.5) = 3$, and so on. This second list array is then processed recursively before working on the next list, i.e., the one for item a in the main set (Figure 2 left).

It also has to be pointed out that there are now empty elements in the transaction lists. As mentioned in Section 2.2.2, a list element representing a (shrunk) transaction that contains only one item is neither reassigned nor copied, only the counter in the lists array element is incremented. When mining fuzzy frequent item sets, however, we have to reassign and copy such an element as well. Even though the transaction is empty after the leading item has been removed, it still can be processed further by inserting items. Such a list element is therefore reassigned/copied—as an empty transaction—to the list associated with the only item it contains.

An example of this can be seen when the list corresponding to item a is processed (step 2). The first list element refers to a (shrunk) transaction that contains only item d. Thus the counter for item d is incremented and an empty element is kept in the list associated with item d (both reassignment and copy), since items c and/or b could be inserted later.

In addition, in Figure 2 a new element labeled with ε is added to the array of transaction lists. This new list is needed when an empty transaction has to be reassigned/copied. Since an empty transaction obviously has no leading item, it cannot be inserted into one of the lists corresponding to the items of the transaction database. However, it cannot be discarded either, because in later processing items may be inserted into it, and then it has to be considered in the corresponding recursion. Formally, ε can be seen as an additional pseudo-item, which is contained in all transactions, but which is not to be reported as part of a frequent item set.

An example of the use of this new array element can be seen when the list corresponding to item b is processed (step 4). In this list there are two empty transactions, which are reassigned and copied to the additional lists array element labeled with ε. Even though these transactions are now empty, they have to be considered when processing item d, because this item may be inserted into them.

3 Experimental Results

To evaluate our algorithm, we implemented it in C and ran experiments on a laptop with a 1.8 GHz Intel Pentium Mobile processor and 1 GB main memory using Windows XP Professional SP2. Results obtained with the original program for normal frequent item sets mining are labeled “Relim”, those for fuzzy frequent item sets mining “Relx” (which are actually the names of the corresponding programs). In all experiments we updated the weight of a transaction by simply multiplying it with an insertion cost factor (if necessary).

In an initial test, we used the very simple transaction database shown in Table 1, using a minimum support of 30%, to check the basic functionality of the approach. When mining this database with exact matching (i.e. without insertions) 11 frequent item sets are found. With fuzzy matching based on uniform insertion costs of 0.5 for all items and a threshold of 0.4 for the transaction weight (thus allowing exactly one insertion), 23 item sets are found. The item set “ec”, which we used as an example above, is found with fuzzy matching, but not with exact matching. On the other hand, if the insertion of item e is ruled out by setting $c(e) = 0$, the item set “ec” is not found anymore. This is the behavior we want.

To check the performance on larger data sets, we tested our programs on the data sets census [3] and T10I4D100K [12]. We used a minimum support of 30% for census and of 5% for T10I4D100K. The insertion cost factor was chosen to be 0.5 for all items in both cases. The number of frequent item sets discovered and the corresponding execution time (in seconds) are shown in Table 2. If insertions were inhibited, the number of sets reported by Relx coincides with that of Relim, proving the sanity of the implementation. However, as was to be expected, Relx needs more time as it has to invest additional effort into managing empty transactions (cf. Section 2.3; an additional factor is the computation of penalized weights, which takes place nevertheless).

The results produced by Relx with different values for the threshold of the transaction weight (that is, allowing 0, 1, or 2 insertions) show—not surprisingly—that with decreasing threshold the number of frequent item sets as well as the execution time increases. That is, frequent item sets that could not be found before are now discovered. Note, however, that the resulting frequent item sets now have fractional support, which is an effect of the transaction weighting. It is pleasing to observe that the execution times are still bearable, even though the insertions make it necessary to process a much higher number of transactions in the recursion.

4 Conclusions

In frequent item sets mining on real-world data—like, for example, alarm sequence analysis in telecommunication networks—there is a need for fuzzy mining, because alarms can get lost or delayed and thus may be missing from the corresponding transactions. In order to face this challenge, we developed an approach for mining fuzzy frequent item sets, namely recursive elimination with the insertion of missing items. This approach is based on deleting items, editing item sets, recursive processing, and reassigning transactions. The algorithm is very simple, works without complicated data

structures, and performs reasonably well, as can be seen from the experiments reported in the preceding section.

Up to now, we only investigated how to edit an item set by insertion. However, there are also other interesting editing operations, like replacing an item with another (which for some items could have different costs than deletion and subsequent insertion). Furthermore, we plan to take the time order of the alarms into account, which currently gets lost in the transformation into simple transactions. If, however, the sequence information is kept, operations like exchanging the order of two items become possible and are definitely worth to be studied.

5 Program

The implementation of the recursive elimination algorithm and the extended version for mining fuzzy frequent item sets 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/relim.html

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. Keeping Things Simple: Finding Frequent Item Sets by Recursive Elimination. (unpublished manuscript)
http://fuzzy.cs.uni-magdeburg.de/~borgelt/relim.html

[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] H. Mannila, H. Toivonen, and A.I. Verkamo. Discovery of frequent episodes in event sequences. In: *Report C-1997-15*. University of Helsinki, Finland.

[9] P. Moen. Attribute, Event Sequence, and Event Type Similarity Notions for Data Mining. *Ph.D. Thesis, Report A-2000-1*. Department of Computer Science, University of Helsinki, Finland 2000

[10] 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

[11] 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

[12] 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