Fuzzy Frequent Item Set Mining based on Recursive Elimination

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

Real life transaction data often miss some occurrences of items that are actually present. As a consequence some potentially interesting frequent item sets cannot be discovered, since with exact matching the number of supporting transactions may be smaller than the user-specified minimum. In order to allow approximate matching during the mining process, we propose an approach based on transaction editing. Our recursive algorithm relies on a step by step elimination of items from the transaction database together with a recursive processing of transaction subsets. This algorithm works without complicated data structures and allows us to find fuzzy frequent item sets easily.

1. Introduction

In many applications of frequent item set mining the considered transactions do not contain all items that are actually present. An example is the analysis of alarm sequences in telecommunication networks, where each alarm can be treated as an item. Unfortunately, the alarms often get delayed, lost, or repeated due to noise, transmission errors, failing links etc. If alarms do not get through or are delayed, they can be missing from the transaction (time window) its associated items (alarms) occur in. If we used exact matching in this case, the support of some item sets, which could be frequent if the items did not get lost, may be smaller than the user-specified minimum. This leads to a possible loss of potentially interesting frequent item sets.

In order to cope with such missing information, we introduce the notion of a "fuzzy" frequent item set. In contrast to other work on fuzzy association rules, where a fuzzy approach is used to deal with quantitative items/attributes, we use the term "fuzzy" to denote an item set that may not be found exactly in all supporting transactions, but only approximately. We propose an algorithm that relies on a step by step elimination of items together with a recursive processing of transaction subsets. Due to its simple data structure, transaction editing is straightforward, thus allowing effective fuzzy mining by inserting missing items.

2. Fuzzy frequent item sets

Let us briefly recall the problem of frequent item set mining: let $I = \{i_1, ..., i_n\}$ be the set of items in a database D consisting of transactions $T = (tid, X)$ where *tid* is a transaction identifier and $X \subseteq I$. A transaction $T = (tid, X)$ is said to *contain* an item set Y if $Y \subseteq X$. The support of Y, denoted as $s(Y)$, is the number of transactions in D containing item set Y. Given a transaction database D and a support threshold $s_{\min}$, an item set Y is called *frequent* if and only if $s(Y) \geq s_{\min}$. According to this classical definition, a transaction T contributes to the support of an item set Y either with 1 if T contains the all items in Y, or with 0 if not.

Motivated by the problem stated above, we define a "fuzzy frequent item set" by allowing approximate matching instead of the exact matching reviewed above. Preceding this, however, we introduce two additional notions.

1. Edit costs: The distance between two item sets can be defined as the costs of the cheapest sequence of edit operations needed to transform one item set into the other [8]. Here we consider only insertions, since they are very easy to implement with our algorithm (see below).[1] Different items can have 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 T in the original database is associated with a weight $w(T)$; the initial weight

[1] Note that deletions are implicit in the mining process anyway. Only replacements are an additional case we do not consider here.

of each transaction is 1. After each insertion of an item i into a transaction, its weight is "penalized" with the cost $c(i)$ associated with the insertion of this item. Formally, this can be described by a combination function. The new weight of a transaction T after editing is $w_{\{i\}}(T) = f(w(T), c(i))$, where f is a function that combines the weight $w(T)$ before editing and the insertion cost $c(i)$. There is a wide variety of combination functions that may be used, for instance, any t-norm. For simplicity, we use multiplication, i.e., $w_{\{i\}}(T) = w(T) \cdot c(i)$, but this is a more or less arbitrary choice. Note, however, that in this case lower values for the cost $c(i)$ mean higher costs as they penalize the weight more. Note also that the above definition can easily be extended to the insertion of multiple items as $w_{\{i_1,\ldots,i_m\}}(T) = w(t) \cdot \prod_{k=1}^{m} c(i_k)$. It should be clear that it is $w_{\emptyset}(T) = 1$ due to the initial weighting $w(T) = 1$.

How many insertions into a transaction are allowed can be limited by a user-specified lower bound $w_{\min}$ for the transaction weight. If the weight of a transaction falls below this threshold, it is not considered in further mining steps and thus no insertions can be done on it anymore.

Definition 1 Given a user-specified lower bound $w_{\min}$ for the transaction weight, a transaction $T = (tid, X)$ *fuzzy contains* an item set $Y \subseteq I$ if $w_{Y \setminus (X \cap Y)}(T) \geq w_{\min}$. In this case T contributes to the support of Y with $w_{Y \setminus (X \cap Y)}(T)$.

Definition 2 Given a database $D = \{T_1, \ldots, T_r\}$ of transactions $T_k = (k, X_i)$, $1 \leq k \leq r$, and a support threshold $s_{\min}$, an item set $Y \subseteq I$ is a *fuzzy frequent item set* if $\hat{s}(Y) \geq s_{\min}$, where $\hat{s}(Y) = \sum_{k=1}^{r} w_{Y \setminus (X_k \cap Y)}(T_k)$ is the *fuzzy support* of Y.

The basic idea of our approach is to try to "complete" transactions by inserting items during the mining process. Thus we allow for a certain number of mismatches, by which we account for possibly missing items. That is, a transaction still contributes to the support of an item set, though only to a reduced degree, if it contains only part of the items in the set. As an example consider the transaction database in Table 1 on the right, in particular, the 2nd transaction (*ecd*), the 5th (*cb*), the 8th (*ecb*) and the 9th (*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 5th transaction (*cb*) and the 9th (*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*".

In the following, we first present an algorithm called recursive elimination, which is extended to our fuzzy frequent item set mining. In order to ease understanding our algorithm, we describe it with exact matching first and transfer it to the fuzzy case later on. In the description we focus on implementation aspects, since the fuzzy version strongly relies on the data structures used in the implementation.

1	a d f		a d
2	c d e	g 1	e c d
3	b d	f 2	b d
4	a b c d	e 3	a c b d
5	b c	a 4	c b
6	a b d	c 5	a b d
7	b d e	b 7	e b d
8	b c e g	d 8	e c b
9	c d f		c d
10	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).

3. Recursive elimination

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]. Here we consider recursive elimination [5] (Relim for short), 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.

3.1. Preprocessing and data representation

Recursive elimination preprocesses the transaction database similar to several other algorithms for frequent item set mining: 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, the left of which shows 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 (integer num-

bers). 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 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.

As an illustration, Figure 1 shows, at the very top, how the (reduced) database of Table 1 is represented. E.g., the first list, corresponding to item *e*, contains the 2nd, 7th, and 8th transaction, with item *e* removed. The counter in an array element states the number of transactions starting with the corresponding item (3 for item *e*). Note that this counter is not always equal to the length of the associated list, although this is the case for the representation of the initial database. Differences result from (shrunk) transactions that contain no other items thus do not appear in the list.

3.2. 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 removed from the transaction, which is 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 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.)

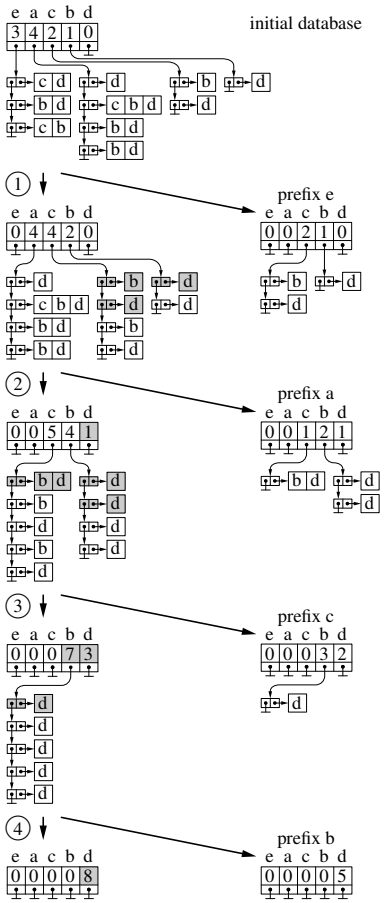

Figure 1. Procedure of the recursive elimination with the modification of the transaction lists (left) as well as the construction of the transaction lists for the recursion (right).

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 (also called a projection of a transaction database w.r.t. 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*.

A list element representing a (shrunk) transaction with only one item is neither reassigned nor copied, because the transaction is empty after the leading item is removed. For such elements only the counter in the lists array element is incremented. Such a situation occurs, for example, 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.

4. Fuzzy mining

For fuzzy frequent item set mining we extend Relim with the two notions introduced in Section 2—edit costs and transaction weights. To store a transaction weight we add a component to each list element described in Section 3.2.

The list array that represents a (reduced) transaction database is processed in basically the same way as before. The most important modification lies in the construction of the subset of transaction lists that represents the projected transaction database w.r.t. a specific item. Figure 2 shows how the extended algorithm (called Relx) works for the root level of the recursion. We consider the same transaction database as in Figure 1. In this example we use the same cost factor $c(i) = 0.5$ for all items. Hence a transaction weight is updated after an insertion according to $w' = w \cdot 0.5$. Furthermore, we assume $w_{\min} = 0$ and $s_{\min} = 3$.

Before a transaction list is processed, its support counter in the array element is checked. Note that this support counter is now even less an indicator of the number of the elements of the transaction list, 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). In the first step all item sets containing the item *e* are found by processing the leftmost list. Since the support counter is $3 \geq s_{\min}$, *e* is reported as a frequent item set. Reassigning the elements of this list to the lists on the right is the same as before (cf. 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, they all contain item *e*), but also the elements of the lists corresponding to items *a*, *c*, and *b* with a penalized weight of 0.5 (white list elements, $0.5 \geq w_{\min}$). Thus we virtually insert item *e* into the corresponding transactions. Obviously with such operations we have more transactions to process in the recursion and hence a (considerably) longer execution time is to be expected.

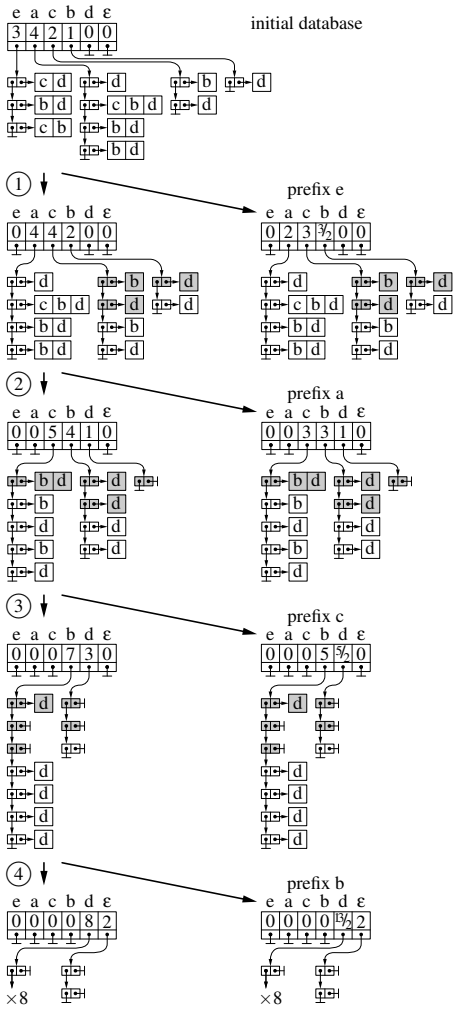

Figure 2. Procedure of the recursive elimination with the insertion of missing items. Since insertions of all items are possible, the subset (right) always has the same structure as the main set (left). The support values, however, differ due to the transaction weighting.

Due to the insertion, in the projected transaction database w.r.t. item *e* (i.e. the list array resulting from the copy step mentioned above), 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. Since $s(c) \geq s_{\min}$, we have that item *c* is frequent in this projected transaction database. Combining it with the associated prefix *e* yields the fuzzy frequent item set *ec* (up to now we found two frequent item sets—*e* and *ec*). This 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).

Note that there are now list elements referring to empty transactions. In contrast to exact mining, mining fuzzy fre-

census	number of sets	time/s
Relim (original data)	244	0.46
Relim (with deletions)	238	0.46
Relx (no insertion)	244	0.47
Relx ($w_{\min} = 0.4$)	1340	4.58
Relx ($w_{\min} = 0.2$)	2510	13.14

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

Table 2. Results on census and T10I4D100K.

quent item sets requires that we also reassign and copy a list element representing a (shrunk) transaction with only one item. Even though the transaction is empty after the leading item is removed, it has to be maintained as it can be processed further by inserting items. Therefore such a list element is 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, a new element labeled 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 has no leading item, it cannot be inserted into one of the lists corresponding to the items of the database. 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 how this new array element is used 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.

5. 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 (exact frequent item set mining) are labeled "Relim", those for fuzzy frequent item set mining "Relx". In all experiments we updated the weight of a transaction by 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 in Section 2, 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.

To check the performance on larger data sets, we tested our programs on the data sets census [3] and T10I4D100K [12], with a minimum support of 30% for census and 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 4; an additional factor is the computation of penalized weights, which takes place nevertheless).

Results produced by Relx with different thresholds for the transaction weight (allowing 0 to 2 insertions) show—not surprisingly—that with decreasing threshold the number of frequent item sets and the execution time increases. Frequent item sets that could not be found before are now discovered. Note, however, that the frequent item sets now have fractional support due to the transaction weighting. Note also 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.

Finally, we ran the program on data from which items had been deleted randomly to check the effectiveness of the proposed algorithm. Here we present only an example of the results. We randomly deleted 4% of item "hours=full-time" and 3% of the item "sex=female" from the census data set to simulate the missing items and then mined with a minimum support of 30%. We ran Relim on both the unmodified and the preprocessed data. With the former 244 frequent itemsets were found, while only 238 of them were detected in the later (cf. Table 2). That is, due to missing occurrences of two items, we lost 6 frequent itemsets. When we used Relx with an insertion cost of 0.5 for both items and a threshold of transaction weight of 0.4, we could find the complete item sets that were obtained by Relim from the

unmodified data. In fact, a superset of the original frequent item sets were reported. However, we have to accept this as an inherent feature of the insertion concept. Still the results are encouraging, and prove that our algorithm is capable of rediscovering frequent itemsets, which are lost with classical approaches due to missing information in the data.

6. Conclusions

Frequent item set mining on real-world data with missing information calls for fuzzy mining. Facing this challenge, we introduced a concept of fuzzy frequent item sets based on transaction editing. The algorithm we developed for mining fuzzy frequent item sets is based on deleting items, editing transactions, recursive processing, and reassigning transactions. The algorithm is very simple, works without complicated data structures, and performs reasonably well.

Some other work in this direction like in [6], [9], perform approximate matching only by counting the number of different items in the two item sets to be compared, and use an Apriori-like algorithm. The algorithm in [9], as reported, performed much slower (about 100 times or even more) when the authors tried to increase the allowed number of mismatches from 1 to 2. Compared to this our approach provides two main advantages: (1) Our approximate matching is based on a more general scheme—edit operations. It allows the individual treatment of every single item, which enables a better involvement of background knowledge. (2) It avoids scanning the original database many times (which is the case in Apriori-like algorithms). Instead it looks for (locally) frequent items in recursively projected databases and combines them with the associated prefix (which is the frequent item set found so far) to yield the frequent item sets. Thus it is more efficient, as can be seen in Table 2. The execution time in the case of allowing two insertions is only about thrice that for one insertion.

Up to now, we only investigated how to edit an item set by insertion. However, there are also other interesting editing operations. If we take the order of items into account, operations like exchanging the order of two items are definitely worth to be studied.

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. pages 307–328 in: 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

[3] C.L. Blake and C.J. Merz. *UCI Repository of Machine Learning Databases*. Dept. of Information and Computer Science, UC 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 Itemset Mining Implementations (FIMI 2003, Melbourne, FL)*. CEUR Workshop Proc. 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) http://fuzzy.cs.uni-magdeburg.de/˜borgelt/relim.html

[6] Y. Cheng, U. Fayyad, and P.S. Bradley. Efficient Discovery of Error-Tolerant Frequent Itemsets in High Dimensions. *Proc. 7th Int. Conf. on Knowledge Discovery and Data Mining (KDD'01, San Francisco, CA)*, 194–203. ACM Press, New York, NY, USA 2001

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

[8] 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

[9] J. Pei, A.K.H. Tung, and J. Han. Fault-Tolerant Frequent Pattern Mining: Problems and Challenges. *Proc. ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMK'01, Santa Babara, CA)*. Santa Babara, CA, May 2001

[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. *Proc. 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, Newport Beach, CA)*, 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