

A tight upper bound on the number of candidate patterns

Floris Geerts, Bart Goethals, and Jan Van den Bussche
Limburgs Universitair Centrum, Belgium

Abstract

In the context of mining for frequent patterns using the standard levelwise algorithm, the following question arises: given the current level and the current set of frequent patterns, what is the maximal number of candidate patterns that can be generated on the next level? We answer this question by providing a tight upper bound, derived from a combinatorial result from the sixties by Kruskal and Katona. Our result is useful to reduce the number of database scans.

1 Introduction

The frequent pattern mining problem [3] is by now well known. We are given a set of items \mathcal{I} and a database \mathcal{D} of subsets of \mathcal{I} called transactions. A *pattern* is some set of items; its *support* in \mathcal{D} is defined as the number of transactions in \mathcal{D} that contain the pattern; and a pattern is called *frequent* in \mathcal{D} if its support exceeds a given minimal support threshold. The goal is now to find all frequent patterns in \mathcal{D} .

The search space of this problem, all subsets of \mathcal{I} , is clearly huge. Instead of generating and counting the supports of all these patterns at once, several solutions have been proposed to perform a more directed search through all patterns. However, this directed search enforces several scans through the database, which brings up another great cost, because these databases tend to be very large, and hence they do not fit into main memory.

The standard Apriori algorithm [4] for solving this problem is based on its monotonicity property, that all subsets of a frequent pattern must be frequent. A pattern is thus considered potentially frequent, also called a *candidate* pattern, if its support is yet unknown, but all of its subsets are already known to be frequent. In every step of the algorithm, all candidate patterns are generated and their supports are then counted by performing a complete scan of the transaction database. This is repeated until no new candidate patterns can be generated. Hence, the number of scans through the database equals the maximal size of a candidate pattern. Several improvements on the Apriori algorithm try

to reduce the number of scans through the database by estimating the number of candidate patterns that can still be generated.

At the heart of all these techniques lies the following purely combinatorial problem, that must be solved first before we can seriously start applying them: *given the current set of frequent patterns at a certain pass of the algorithm, what is the maximal number of candidate patterns that can be generated in the passes yet to come?*

Our contribution is to solve this problem by providing a hard and tight combinatorial upper bound. By computing our upper bound after every pass of the algorithm, we have at all times a watertight guarantee on the size of what is still to come, on which we can then base various optimization decisions, depending on the specific algorithm that is used.

In the next Section, we will discuss existing techniques to reduce the number of database scans, and point out the dangers of using existing heuristics for this purpose. Using our upper bound, these techniques can be made watertight. In Section 3, we derive our upper bound, using a combinatorial result from the sixties by Kruskal and Katona. In Section 4, we show how to get even more out of this upper bound by applying it recursively. In Section 5, we discuss several issues concerning the implementation of the given upper bounds on top of Apriori-like algorithms. In Section 6, we give experimental results, showing the effectiveness of our result in estimating, far ahead, how much will still be generated in the future. Finally, we conclude the paper in Section 7.

Due to space limitations, the proofs of the theorems in this paper have been omitted.

2 Related Work

Nearly all frequent pattern mining algorithms developed after the proposal of the Apriori algorithm, rely on its levelwise candidate generation and pruning strategy. Most of them differ in how they generate and count candidate patterns.

One of the first optimizations was the DHP algorithm proposed by Park et al. [19]. This algorithm uses a hashing scheme to collect upper bounds on the frequencies of the

candidate patterns for the following pass. Patterns of which it is already known they will turn up infrequent can then be eliminated from further consideration. The effectiveness of this technique only showed for the first few passes. Since our upper bound can be used to reduce the number of passes and hence after the first few passes, both techniques can be combined in the same algorithm.

Other strategies, discussed next, try to reduce the number of passes. However, such a reduction of passes often causes an increase in the number of candidate patterns that need to be explored during a single pass. This tradeoff between the reduction of passes and the number of candidate patterns is important since the time needed to process a transaction is dependent on the number of candidates that are covered in that transaction, which might blow up exponentially. Our upper bound can be used to predict whether or not this blowup will occur.

The Partition algorithm, proposed by Savasere et al. [20], reduces the number of database passes to two. Towards this end, the database is partitioned into parts small enough to be handled in main memory. The partitions are then considered one at a time and all frequent patterns for that partition are generated using an Apriori-like algorithm. At the end of the first pass, all these patterns are merged to generate a set of all potential frequent patterns, which can then be counted over the complete database. Although this method performs only two database passes, its performance is heavily dependent on the distribution of the data, and could generate much too many candidates.

The sampling algorithm proposed by Toivonen [22] performs at most two scans through the database by picking a random sample from the database, then finding all frequent patterns that probably hold in the whole database, and then verifying the results with the rest of the database. In the cases where the sampling method does not produce all frequent patterns, the missing patterns can be found by generating all remaining potentially frequent patterns and verifying their frequencies during a second pass through the database. The probability of such a failure can be kept small by decreasing the minimal support threshold. However, this can again cause a combinatorial explosion of the number of candidate patterns.

The DIC algorithm, proposed by Brin et al. [7], tries to reduce the number of passes over the database by dividing the database into intervals of a specific size. First, all candidate patterns of size 1 are generated. The frequencies of the candidate sets are then counted over the first interval of the database. Based on these relative frequencies, candidate patterns of size 2 are generated and are counted over the next interval together with the patterns of size 1. In general, after every interval k , candidate patterns of size $k + 1$ are generated and counted. The algorithm stops if no more candidates can be generated. Although this method drasti-

cally reduces the number of scans through the database, its performance is also heavily dependent on the the distribution of the data, and hence it could again generate too many candidates.

Recently, the first successful attempts to generate frequent patterns using a depth-first search were proposed by Agarwal et al. [1, 2] and by Han et al. [11]. Generating patterns in a depth-first manner implies that the monotonicity property cannot be exploited anymore. Hence, a lot more patterns will be generated and need to be counted, compared to the breadth-first algorithms. The FPgrowth algorithm from Han et al. solves this problem by loading a compressed form of the database in main memory using the proposed FPtree. This memory-resident FPtree benefits from a very fast counting mechanism of all generated patterns.¹ Obviously, it is not always possible to load the compressed form of the database into main memory.

Other strategies try to push certain constraints into the candidate pattern generation as deeply as possible to reduce the number of candidate patterns that must be generated [9, 15, 18, 21]. Still others try to find only the set of *maximal* frequent patterns, i.e. those frequent patterns that have no superset which is also frequent [6, 16, 23]. Of course, these techniques do not give us all frequencies of all frequent patterns as required by the general pattern mining problem we consider in this paper.

The first heuristic specifically proposed to estimate the number of candidate patterns that can still be generated was used in the AprioriHybrid algorithm [4, 5]. This algorithm uses Apriori in the initial passes and switches to AprioriTid if it expects it to run faster. This AprioriTid algorithm does not use the database at all for counting the support of candidate patterns. Rather, an encoding of the candidate patterns used in the previous pass is employed for this purpose. The AprioriHybrid algorithm switches to AprioriTid when it expects this encoding of the candidate patterns to be small enough to fit in main memory. The size of the encoding grows with the number of candidate patterns. Therefore, it calculates the size the encoding would have in the current pass. If this size is small enough and there were fewer candidate patterns in the current level than the previous pass, the heuristic decides to switch to AprioriTid.

This heuristic (like all heuristics) is not waterproof, however. Take, for example, two disjoint datasets. The first dataset consists of all subsets of a frequent pattern of size 20. The second dataset consists of all subsets of 1 000 disjoint frequent patterns of size 5. If we merge these two datasets, we get $\binom{20}{3} + 1\,000\binom{5}{3} = 11\,140$ patterns of size 3 and $\binom{20}{4} + 1\,000\binom{5}{4} = 9\,845$ patterns of size 4. If we have enough memory to store the encoding for all these patterns, then the heuristic decides to switch to AprioriTid. This de-

¹Note that the patterns in the FPtree are represented in the so called header tables.

cision is premature, however, because the number of new patterns in each pass will start growing exponentially afterwards.

Another improvement of the Apriori algorithm, which is part of the folklore, tries to combine as many iterations as possible in the end, when only few candidate patterns can still be generated. The potential of such a combination technique was realized early on [4, 17], but the modalities under which it can be applied were never further examined. Our work does exactly that.

3 The basic upper bounds

In all that follows, L is some family of patterns of size k .

Definition 1. A candidate pattern for L is a pattern (of size larger than k) of which all k -subsets are in L . For a given $\ell > k$, we denote the set of all size- ℓ candidate patterns for L by $C_\ell(L)$.

For any $p \geq 1$, we will provide an upper bound on $|C_{k+p}(L)|$ in terms of $|L|$. The following lemma is central to our approach: (a simple proof was given by Katona [13])

Lemma 1. Given n and k , there exists a unique representation

$$n = \binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \dots + \binom{m_r}{r},$$

with $r \geq 1$, $m_k > m_{k-1} > \dots > m_r$, and $m_i \geq i$ for $i = r, r+1, \dots, k$.

This representation is called the k -canonical representation of n and can be computed as follows: The integer m_k satisfies $\binom{m_k}{k} \leq n < \binom{m_k+1}{k}$, the integer m_{k-1} satisfies $\binom{m_{k-1}}{k-1} \leq n - \binom{m_k}{k} < \binom{m_{k-1}+1}{k-1}$, and so on, until $n - \binom{m_k}{k} - \binom{m_{k-1}}{k-1} - \dots - \binom{m_r}{r}$ is zero.

We now establish the following theorem. The proof is based on a combinatorial result of Kruskal and Katona [8, 13, 14].

Theorem 2. If

$$|L| = \binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \dots + \binom{m_r}{r}$$

in k -canonical representation, then

$$|C_{k+p}(L)| \leq \binom{m_k}{k+p} + \binom{m_{k-1}}{k-1+p} + \dots + \binom{m_{s+1}}{s+p+1},$$

where s is the smallest integer such that $m_s < s+p$. If no such integer exists, we set $s = r-1$.

Notation We will refer to the upper bound provided by the above theorem as $KK_k^{k+p}(|L|)$ (for Kruskal-Katona). The subscript k , the level at which we are predicting, is important, as the only parameter is the cardinality $|L|$ of L , not L itself. The superscript $k+p$ denotes the level we are predicting.

Proposition 3 (Tightness). The upper bound provided by Theorem 2 is tight: for any given n and k there always exists an L with $|L| = n$ such that for any given p , $|C_{k+p}(L)| = KK_k^{k+p}(|L|)$.

Analogous tightness properties hold for all upper bounds we will present in this paper, but we will no longer explicitly state this.

Example 1. Let L be the set of 13 patterns of size 3:

$$\begin{aligned} & \{ \{3, 2, 1\}, \\ & \{4, 2, 1\}, \{4, 3, 1\}, \{4, 3, 2\}, \\ & \{5, 2, 1\}, \{5, 3, 1\}, \{5, 3, 2\}, \\ & \{5, 4, 1\}, \{5, 4, 2\}, \{5, 4, 3\}, \\ & \{6, 2, 1\}, \{6, 3, 1\}, \{6, 3, 2\} \}. \end{aligned}$$

The 3-canonical representation of 13 is $\binom{5}{3} + \binom{3}{2}$ and hence the maximum number of candidate patterns of size 4 is $KK_3^4(13) = \binom{5}{4} + \binom{3}{3} = 6$ and the maximum number of candidate patterns of size 5 is $KK_3^5(13) = \binom{5}{5} = 1$. This is tight indeed, because

$$C_4(L) = \{ \{4, 3, 2, 1\}, \{5, 3, 2, 1\}, \{5, 4, 2, 1\}, \\ \{5, 4, 3, 1\}, \{5, 4, 3, 2\}, \{6, 3, 2, 1\} \}$$

and

$$C_5(L) = \{ \{5, 4, 3, 2, 1\} \}.$$

Estimating the number of levels The k -canonical representation of $|L|$ also yields an upper bound on the maximal size of a candidate pattern, denoted by $\text{maxsize}(L)$. Recall that this size equals the number of iterations the standard Apriori algorithm will perform. Indeed, since $|L| < \binom{m_k+1}{k}$, there cannot be a candidate pattern of size m_k+1 or higher, so:

Proposition 4. If $\binom{m_k}{k}$ is the first term in the k -canonical representation of $|L|$, then $\text{maxsize}(L) \leq m_k$.

We denote this number m_k by $\mu_k(|L|)$. From the form of KK_k^{k+p} as given by Theorem 2, it is immediate that μ also tells us the last level before which KK becomes zero. Formally:

Proposition 5. $\mu_k(|L|) = \min\{p \mid KK_k^{k+p}(|L|) = 0\} + k - 1$.

Estimating all levels As a result of the above, we can also bound, at any given level k , the *total* number of candidate patterns that can be generated, as follows:

Proposition 6. *The total number of candidate patterns that can be generated from a set L of k -patterns is at most*

$$KK_k^{\text{total}}(L) := \sum_{p=1}^{\mu_k(|L|)} KK_k^{k+p}(|L|).$$

4 Getting the most out of it

The upper bound KK on itself is neat and simple as it takes as parameters only two numbers: the current size k , and the number $|L|$ of current frequent patterns. However, in reality, when we have arrived at a certain level k , we do not merely have the cardinality: we have the actual set L of current k -patterns! For example, if the frequent patterns in the current pass are all disjoint, our current upper bound will still estimate their number to a certain non-zero figure. However, by the pairwise disjointness, it is clear that no further patterns will be possible at all. In sum, because we have richer information than a mere cardinality, we should be able to get a better upper bound.

To get inspiration, let us recall that the candidate generation process of the Apriori algorithm works in two steps. In the *join* step, we join L with itself to obtain a superset of C_{k+1} . The union $p \cup q$ of two patterns $p, q \in L$ is inserted in C_{k+1} if they share their $k - 1$ smallest items:

```

insert into  $C_{k+1}$ 
select  $p[1], p[2], \dots, p[k], q[k]$ 
from  $L_k$   $p, L_k$   $q$ 
where  $p[1] = q[1], \dots, p[k-1] = q[k-1], p[k] < q[k]$ 

```

Next, in the *prune* step, we delete every pattern $c \in C_{k+1}$ such that some k -subset of c is not in L .

Let us now take a closer look at the join step from another point of view. Consider a family of all frequent patterns of size k that share their $k - 1$ smallest items, and let its cardinality be n . If we now remove from each of these patterns all these shared $k - 1$ smallest items, we get exactly n distinct single-item patterns. The number of pairs that can be formed from these single items, being $\binom{n}{2}$, is exactly the number of candidates the join step will generate for the family under consideration. We thus get an obvious upper bound on the total number of candidates by taking the sum of all $\binom{n_f}{2}$, for every possible family f .

This obvious upper bound on $|C_{k+1}|$, which we denote by $obvious_{k+1}(L)$, can be recursively computed in the following manner. Let I denote the set of items occurring in L . For an arbitrary item x , define the set L^x as

$$L^x = \{s - \{x\} \mid s \in L \text{ and } x = \min s\}.$$

Then

$$obvious_{k+1}(L) := \begin{cases} \binom{|L|}{2} & \text{if } k = 1; \\ \sum_{x \in I} obvious_k(L^x) & \text{if } k > 1. \end{cases}$$

This upper bound is much too crude, however, because it does not take the prune step into account, only the join step. The join step only checks two k -subsets of a potential candidate instead of all $k + 1$ k -subsets.

However, we can generalize this method such that more subsets will be considered. Indeed, instead of taking a family of all frequent patterns sharing their $k - 1$ smallest items, we can take all frequent patterns sharing only their k' smallest items, for some $k' \leq k - 1$. If we then remove these k' shared items from each pattern in the family, we get a new set L' of n patterns of size $k - k'$. If we now consider the set C' of candidates (of size $k - k' + 1$) for L' , and add back to each of them the previously removed k' items, we obtain a pruned set of candidates of size $k + 1$, where instead of just two (as in the join step), $k - k' + 1$ of the k -subsets were checked in the pruning. Note that we can get the estimate $KK_{k-k'}^{k-k'+1}(|L'|)$ on the cardinality of C' from our upper bound Theorem 2.

Doing this for all possible values of k' yields an improved upper bound on $|C_{k+1}|$, which we denote by $improve d_{k+1}(L)$, and which is computed by refining the recursive procedure for the obvious upper bound as follows:

$$improve d_{k+1}(L) := \begin{cases} \binom{|L|}{2} & \text{if } k = 1; \\ \min\{KK_k^{k+1}(|L|), \sum_{x \in I} improved_k(L^x)\} & \text{if } k > 1. \end{cases}$$

Actually, as in the previous section, we can do this not only to estimate $|C_{k+1}|$, but also more generally to estimate $|C_{k+p}|$ for any $p \geq 1$. Henceforth we will denote our general improved upper bound by $KK_{k+p}^*(L)$. The general definition is as follows:

$$KK_{k+p}^*(L) := \begin{cases} \binom{|L|}{p+1} & \text{if } k = 1; \\ \min\{KK_k^{k+p}(|L|), \sum_{x \in I} KK_{k+p-1}^*(L^x)\} & \text{if } k > 1. \end{cases}$$

(For the base case, note that $\binom{|L|}{p+1}$, when $k = 1$, is nothing but $KK_k^{k+p}(|L|)$.)

By definition, KK_{k+p}^* is always smaller than KK_k^{k+p} . We can prove formally that it is still an upper bound on the number of candidate patterns of size $k + p$:

Theorem 7. $|C_{k+p}(L)| \leq KK_{k+p}^*(L)$.

A natural question is why we must take the minimum in the definition of KK^* . The answer is that the two terms of which we take the minimum are incomparable. The example of an L where all patterns are pairwise disjoint, already mentioned in the beginning of this section, shows that, for example, $KK_k^{k+1}(|L|)$ can be larger than the summation $\sum_{x \in I} KK_k^*(L^x)$. But the converse is also possible: consider $L = \{\{1, 2\}, \{1, 3\}\}$. Then $KK_2^3(L) = 0$, but the summation yields 1.

Example 2. Let L consist of all 19 3-subsets of $\{1, 2, 3, 4, 5\}$ and $\{3, 4, 5, 6, 7\}$ plus the sets $\{5, 7, 8\}$ and $\{5, 8, 9\}$. Because $21 = \binom{6}{3} + \binom{2}{2}$, we have $KK_3^4(21) = 15$, $KK_3^5(21) = 6$ and $KK_3^6(21) = 1$. On the other hand,

$$\begin{aligned} KK_4^*(L) &= KK_3^*(L^1) + KK_3^*(L^2) \\ &\quad + KK_3^*(L^3) + KK_3^*(L^4) \\ &\quad + KK_2^*((L^5)^6) + KK_2^*((L^5)^7) \\ &\quad + KK_2^*((L^5)^8) + KK_2^*((L^5)^9) \\ &\quad + KK_3^*(L^6) + KK_3^*(L^7) \\ &\quad + KK_3^*(L^8) + KK_3^*(L^9) \\ &= 4 + 1 + 4 + 1 + 0 + \dots + 0 \\ &= 10 \end{aligned}$$

and

$$\begin{aligned} KK_5^*(L) &= KK_4^*(L^1) + KK_4^*(L^2) \\ &\quad + KK_4^*(L^3) + KK_4^*(L^4) \\ &\quad + KK_3^*((L^5)^6) + KK_3^*((L^5)^7) \\ &\quad + KK_3^*((L^5)^8) + KK_3^*((L^5)^9) \\ &\quad + KK_4^*(L^6) + KK_4^*(L^7) \\ &\quad + KK_4^*(L^8) \\ &\quad + KK_4^*(L^9) \\ &= 1 + 0 + 1 + 0 + 0 + \dots + 0 \\ &= 2. \end{aligned}$$

Indeed, we have 10 4-subsets of $\{1, 2, 3, 4, 5\}$ and $\{3, 4, 5, 6, 7\}$, and the two 5-sets themselves.

We can also improve the upper bound $\mu_k(|L|)$ on $\text{maxsize}(L)$. Indeed, in analogy with Proposition 5, we define:

$$\mu^*(L) := \min\{p \mid KK_{k+p}^*(L) = 0\} + k - 1.$$

We then have:

Proposition 8. $\text{maxsize}(L) \leq \mu^*(L) \leq \mu(L)$.

We finally use Theorem 7 for improving the upper bound KK_k^{total} on the total number of candidate patterns. Indeed,

define:

$$KK_{\text{total}}^*(L) = \sum_{p=1}^{\mu^*(L)} KK_{k+p}^*(L)$$

Then we have:

Proposition 9. *The total number of candidate patterns that can be generated from a set L of k -patterns is bounded by $KK_{\text{total}}^*(L)$. Moreover, $KK_{\text{total}}^*(L) \leq KK_k^{\text{total}}(L)$.*

5 Efficient Implementation

To evaluate our upper bounds we implemented an optimized version of the Apriori algorithm using a trie data structure to store all generated patterns, similar to the one described by Brin et al. [7]. This trie structure makes it cheap and straightforward to implement the computation of all upper bounds. Indeed, a top-level subtree (rooted at some singleton pattern $\{x\}$) represents exactly the set L^x we defined in Section 4. Every top-level subtree of this subtree (rooted at some two-element pattern $\{x, y\}$) then represents $(L^x)^y$, and so on. Hence, we can compute the recursive bounds while traversing the trie, after the frequencies of all candidate patterns are counted, and we have to traverse the trie once more to remove all candidate patterns that turned out to be infrequent. This can be done as follows.

Remember, at that point, we have the current set of frequent patterns of size k stored in the trie. For every node at depth d smaller than k , we compute the $k-d$ -canonical representation of the number of descendants this node has at depth k , which can be used to compute μ_{k-d} (cf. Proposition 4), KK_{k-d}^ℓ for any $\ell \leq \mu_{k-d}$ (cf. Theorem 2) and hence also KK_{k-d}^{total} (cf. Proposition 6). For every node at depth $k-1$, its KK^* and μ^* values are equal to its KK and μ values respectively. Then compute for every $p > 0$, the sum of the $KK_{k-d+p-1}^*$ values of all its children, and let KK_{k-d+p}^* be the smallest of this sum and KK_{k-d}^{k-d+p} until this minimum becomes zero, which also gives us the value of μ^* . Finally, we can compute KK_{total}^* for this node. If this is done for every node, traversed in a depth-first manner, then finally the root node will contain the upper bounds on the number of candidate patterns that can still be generated, and on the maximum size of any such pattern. The soundness and completeness of this method follows directly from the theorems and propositions of the previous sections.

We should also point out that, since the numbers involved can become exponentially large (in the number of items), an implementation should take care to use arbitrary-length integers such as provided by standard mathematical packages. Since the length of an integer is only logarithmic in its value, the lengths of the numbers involved will remain polynomially bounded.

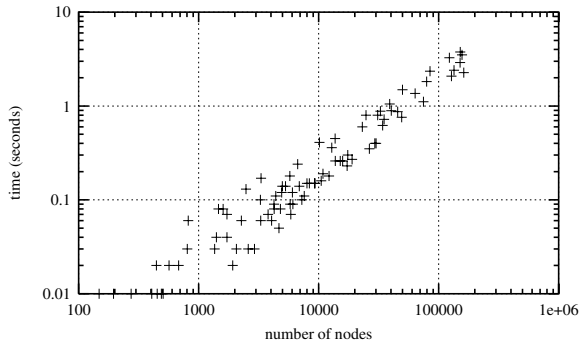


Figure 1. Time needed to compute upper bounds compared to the number of nodes.

The cost for the computation of the upper bounds is negligible compared to the cost of the complete algorithm. Indeed, the time T needed to calculate the upper bounds is largely dictated by the number n of nodes in the trie. We have shown experimentally that T scales linearly with n . Moreover, the constant factor in our implementation is very small (around 0.0025). We ran several experiments using two different datasets and varying minimal support thresholds. After every pass of the algorithm, we registered the number of nodes in the trie and the time spent to compute all upper bounds, resulting in 145 different datapoints. Figure 1 shows these results.

6 Experimental Evaluation

Data sets We have experimented using several synthetic datasets generated by the program provided by the Quest research group at IBM Almaden. For different settings of the parameters of the generator, the resulting figures were very analogous. We also experimented using a real market basket dataset from a Belgian retail store containing 41 373 transactions. The store carries 13 103 products. The results from this experiment were not immediately as good as the results from the synthetic datasets. The reason for this, however, turned out to be the bad ordering of the items, as explained next.

Reordering From the form of L^x , it can be seen that the order of the items can affect the recursive upper bounds. By computing the upper bound only for a subset of all frequent patterns, we win by incorporating the structure of the current collection of frequent patterns, but we also lose some information, because whenever we recursively restrict ourselves to a subtree L^x , then for every candidate pattern s with $x = \min s$, we lose the information about exactly one subpattern in L , namely $s - x$, because all patterns in L^x

come from patterns in L that contain x . We therefore would like to make it likely that many of these excluded patterns are frequent. A good heuristic, which has already been used for several other optimizations in frequent pattern mining [2, 6, 7] is to reorder the single item patterns in increasing order of frequency, such that the excluded subpatterns have the highest probability of being frequent (under the assumption that the occurrence of an event in a transaction is independent of the occurrence of any other event in the transaction).

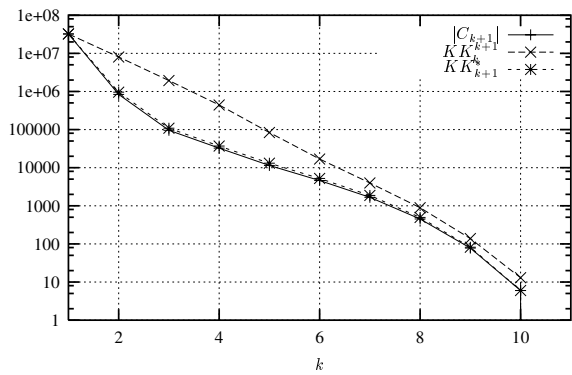
After reordering the items in the real life dataset, using this heuristic, the results became very analogous with the results using the synthetic datasets.

Results Figures 2(a), 2(b), and 2(c) show experimental results on the real life dataset.

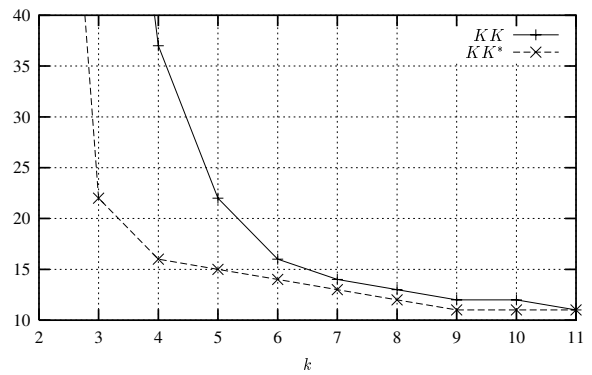
- Figure 2(a) shows, after each level k , the computed upper bound KK and improved upper bound KK^* for the number of candidate patterns at the next level, as well as the actual number $|C_{k+1}|$ it turned out to be.
- Figure 2(b) shows the computed upper bounds μ and μ^* on the maximal size of a candidate pattern. In this experiment, this maximum turned out to be 11.
- Figure 2(c) shows the upper bounds on the total number of candidate patterns that could still be generated, compared to the actual number of candidate patterns, $|C_{\text{total}}|$, that effectively still were generated when we combined all further passes into a single one.
- Figures 2(d), 2(e) and 2(f) show the previous experiment, now performed over a synthetic dataset containing 100 000 transactions over 1 000 items, using a varying minimal support threshold of 0.025%, 0.075% and 0.125% respectively.

Discussion The results are pleasantly surprising:

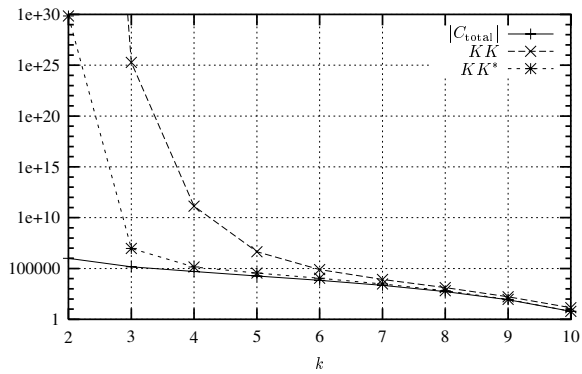
- First, note that the improvement of KK^* over KK , and of μ^* over μ , anticipated by our theoretical discussion, is indeed dramatic.
- Second, comparing the computed upper bounds with the actual numbers, we observe the high accuracy of the estimations given by KK^* . Indeed, the estimations of KK^*_{k+1} match almost exactly the actual number of candidate patterns that has been generated at level $k + 1$.
- The upper bounds on the total number of candidate patterns are still very large when estimated in the first two passes, which is not surprising because at these initial stages, there is not much information yet. From the



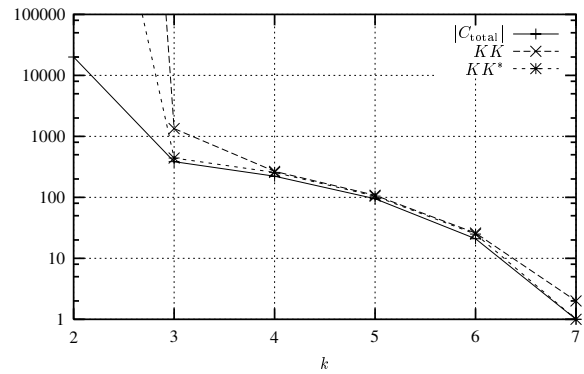
(a)



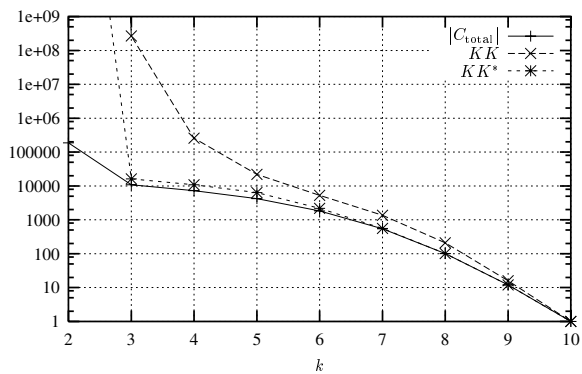
(b)



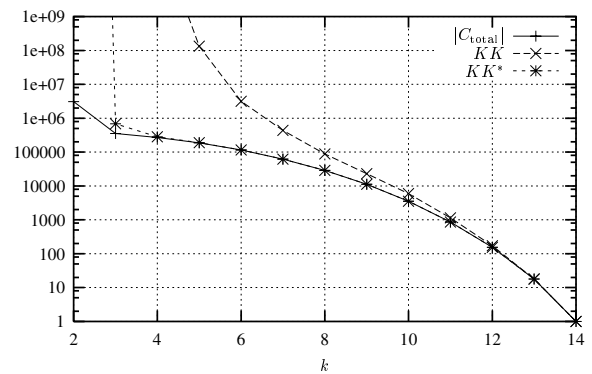
(c)



(d)



(e)



(f)

Figure 2. Experimental results

fourth pass on, however, when the frequent patterns of size 4 are known, the estimations become almost exact.

- Since the totals become manageable from there on, an optimizer using our estimates would decide to combine all iterations 5–11 in a single one, and would have generated at most 142 237 more candidate patterns.
- Entirely similar results are obtained when varying the minimal support thresholds.

References

- [1] R. Agarwal, C. Aggarwal, and V. Prasad. Depth first generation of long patterns. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM Press, 2000.
- [2] R. Agarwal, C. Aggarwal, and V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, March 2001.
- [3] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, volume 22:2 of *SIGMOD Record*, pages 207–216. ACM Press, 1993.
- [4] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. Fast discovery of association rules. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. MIT Press, 1996.
- [5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. IBM Research Report RJ9839, IBM Almaden Research Center, San Jose, California, June 1994.
- [6] R. Bayardo, Jr. Efficiently mining long patterns from databases. In Haas and Tiwary [10], pages 85–93.
- [7] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In J. Peckham, editor, *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, volume 26:2 of *SIGMOD Record*, pages 255–264. ACM Press, 1997.
- [8] P. Frankl. A new short proof for the Kruskal–Katona theorem. *Discrete Mathematics*, 48:327–329, 1984.
- [9] B. Goethals and J. Van den Bussche. On supporting interactive association rule mining. In Y. Kambayashi, M. Mohania, and A. Tjoa, editors, *Data Warehousing and Knowledge Discovery*, volume 1874 of *Lecture Notes in Computer Science*, pages 307–316. Springer, 2000.
- [10] L. Haas and A. Tiwary, editors. *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, volume 27:2 of *SIGMOD Record*. ACM Press, 1998.
- [11] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In W. Chen, J. Naughton, and P. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, volume 29:2 of *SIGMOD Record*, pages 1–12. ACM Press, 2000.
- [12] D. Heckerman, H. Mannila, and D. Pregibon, editors. *Proceedings of the Third International Conference on Knowledge Discovery & Data Mining*. AAAI Press, 1997.
- [13] G. Katona. A theorem of finite sets. In *Theory Of Graphs*, pages 187–207. Akadémia Kiadó, 1968.
- [14] J. Kruskal. The number of simplices in a complex. In *Mathematical Optimization Techniques*, pages 251–278. Univ. of California Press, 1963.
- [15] L. Lakshmanan, R. Ng, J. Han, and A. Pang. Optimization of constrained frequent set queries with 2-variable constraints. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, volume 28:2 of *SIGMOD Record*, pages 157–168. ACM Press, 1999.
- [16] D. Lin and Z. Kedem. Pincer-search: A new algorithm for discovering the maximum frequent set. In H.-J. Schek, F. Saltor, I. Ramos, and G. Alonso, editors, *Proceedings of the 6th International Conference on Extending Database Technology*, volume 1377 of *Lecture Notes in Computer Science*, pages 105–119. Springer, 1998.
- [17] H. Mannila, H. Toivonen, and A. Verkamo. Efficient algorithms for discovering association rules. In R. U. U.M. Fayyad, editor, *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases*, pages 181–192. AAAI Press, 1994.
- [18] R. Ng, L. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained association rules. In Haas and Tiwary [10], pages 13–24.
- [19] J. Park, M.-S. Chen, and P. Yu. An effective hash based algorithm for mining association rules. In D. S. M.J. Carey, editor, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, volume 24:2 of *SIGMOD Record*, pages 175–186. ACM Press, 1995.
- [20] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In U. Dayal, P. Gray, and S. Nishio, editors, *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 432–444. Morgan Kaufmann, 1995.
- [21] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In Heckerman et al. [12], pages 66–73.
- [22] H. Toivonen. Sampling large databases for association rules. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 134–145. Morgan Kaufmann, 1996.
- [23] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In Heckerman et al. [12], pages 283–286.