Tight Upper Bounds on the Number of Candidate Patterns

FLORIS GEERTS
University of Edinburgh
BART GOETHALS
University of Helsinki
and
JAN VAN DEN BUSSCHE
Limburgs Universitair Centrum

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 tight upper bounds, derived from a combinatorial result from the sixties by Kruskal and Katona. Our result is useful to secure existing algorithms from a combinatorial explosion of the number of candidate patterns.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; H.2.8 [**Database Management**]: Database Applications—*Data mining*; I.2.6 [**Artificial Intelligence**]: Learning—*Knowledge acquisition*

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: Data mining, frequent patterns, upper bounds

1. INTRODUCTION

The frequent pattern mining problem is by now well known [Agrawal et al. 1993]. 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

A preliminary report on this work was presented at the 2001 IEEE International Conference on Data Mining [Geerts et al. 2001].

Authors' addresses: F. Geerts, University of Edinburgh, Laboratory for Foundations of Computer Science, School of Informatics, Appleton Tower Room 206, Crichton Street, Edinburgh EH8 9LE, Scotland, UK; email: fgeerts@infoed.ac.uk; B. Goethals, ADReM, Department of Mathematics and Computer Science, University of Antwerp, Middelheimlaan 1, B-2020 Antwerpen, Belgium; email: bart.goethals@ua.ac.be; J. Van den Bussche, Limburgs Universitair Centrum, Department WNI, Universitair Campus, B-3590 Diepenbeek, Belgium; email: jan.vandenbussche@lnc.uc.ac.be.

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, the lattice of 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. During such a search, several collections of *candidate* patterns are generated and their supports computed until all frequent patterns have been found. Obviously, the size of a collection of candidate patterns must not exceed the amount of available main memory. Moreover, it is important to generate as few candidate patterns as possible, since computing the supports of a collection of patterns is a time consuming procedure. The main underlying property exploited by most algorithms is that support is monotone decreasing with respect to extension of a pattern. Hence, if a pattern is infrequent, all of its supersets must be infrequent.

The standard Apriori algorithm for solving this problem performs a breadth-first levelwise search by iteratively generating all (candidate) patterns for which all subsets are known to be frequent, after which their support is counted by performing a scan through the transaction database. This is repeated until no new candidate patterns can be generated [Agrawal et al. 1996]. Recently, it has been shown that this algorithm and its enhancements sometimes still outperform more recent algorithms [Zheng et al. 2001; Goethals and Zaki 2003].

Several variants on this algorithm try to improve the time spent on counting the support of all candidate patterns, for example [Brin et al. 1997; Toivonen 1996; Savasere et al. 1995; Agrawal and Srikant 1994a], but they are strongly dependent on the number of candidate patterns that are generated. More specifically, the main risk lies in the fact that the number of candidate patterns can grow exponentially. At the heart of 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 still need to be generated?*

A brute force method to answer the above question is to simply count the candidate patterns by generating them without storing them. When a combinatorial explosion occurs, however, this method takes a prohibitive amount of time. Indeed, the problem is precisely to predict a combinatorial explosion without suffering from it, neither in space, nor in time.

Our contribution is to theoretically study this problem, which enables us to provide hard and tight combinatorial upper bounds that can be computed efficiently. By computing any of these upper bounds after every iteration 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 optimization techniques, and point out the dangers of using existing heuristics for estimating the number of candidate patterns. 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. We will then generalize the given upper bounds such that they can be applied by a wider range of algorithms in Section 5. In Section 6, we discuss several issues concerning the implementation of the given upper bounds on top of Apriori-like algorithms. In Section 7, we consider three brute force counting methods, that simply generate all possible candidates in order to get the desired numbers we have been trying to bound. In Section 8, 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 article in Section 9.

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. [1995]. This algorithm uses a hashing scheme to collect upper bounds on the frequencies of the candidate patterns for the following iteration. Patterns for which it is already known that they will turn up infrequently can then be eliminated from further consideration and their supports need not be counted anymore. The effectiveness of this technique only showed for the first few iterations.

Since our upper bound can be used to eliminate passes at the end, both techniques can be combined.

Other strategies, discussed next, try to reduce the number of iterations. However, such a reduction often causes an increase in the number of candidate patterns that need to be explored during a single iteration. This tradeoff between the reduction of iterations and the number of candidate patterns is important since the time needed to process a single transaction is dependent on the number of candidates that are contained 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. [1995], 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 with an Apriori-like algorithm using a fast in-memory support counting mechanism. 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 far too many candidates.

The Sampling algorithm proposed by Toivonen [1996] 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, for a reasonably small probability of failure, the threshold must be drastically decreased, which can again cause a combinatorial explosion of the number of candidate patterns.

The DIC algorithm, proposed by Brin et al. [1997], 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 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. Again, this technique can be combined with our technique in the same algorithm.

Another type of algorithm generates frequent patterns using a depth-first search [Zaki et al. 1997; Agarwal et al. 2000, 2001; Han et al. 2000]. Generating patterns in a depth-first manner implies that the monotonicity property cannot be fully exploited anymore. Hence, many more candidate patterns will be generated and need to be counted as compared to the breadth-first algorithms. On the other hand, the depth-first approach facilitates easy and fast support counting by loading the database into main memory. If this is impossible, then several techniques exist to load only (not necessarily disjoint) parts of the database into memory [Savasere et al. 1995; Han et al. 2000]. However, these techniques significantly reduce the performance of such algorithms.

Other strategies try to find only a subset of all frequent patterns from which the support of all remaining frequent patterns can be derived. For example, a very interesting stream of research is primarily focused on finding only all closed frequent itemsets [Pasquier et al. 1999; Zaki and Hsiao 2002; Burdick et al. 2001; Pei et al. 2000]; that is, all frequent itemsets that do not have a superset with the same support. These algorithms typically first have to find the so called free frequent itemsets: those itemsets that do not have a subset with the same support [Boulicaut et al. 2003]. This collection of free frequent itemsets is, like all frequent itemsets, downward closed, hence similar algorithms to those that mine all frequent itemsets, must be used.

When some very long itemsets are frequent, it becomes simply infeasible to mine all frequent itemsets. Therefore, others try to find only the set of *maximal* frequent patterns: those frequent patterns that have no superset that is also frequent [Bayardo 1998; Lin and Kedem 1998; Burdick et al. 2001]. The main techniques of most of these algorithms still iteratively generate collections of candidate itemsets, hence, they are also susceptible to the risk of a combinatorial explosion. The bounds presented in this article will also be able to predict the maximal size of a frequent itemset, which makes it possible to detect long itemsets early in the process, and could for example, allow us to switch to maximal itemset mining before being caught in a combinatorial explosion.

The first heuristic specifically proposed to estimate the number of candidate patterns that can still be generated was used in the AprioriHybrid algorithm [Agrawal and Srikant 1994a, 1994b]. This algorithm uses Apriori in the initial iterations 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 iteration 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 iteration. If this size is small enough and there were fewer candidate patterns in the current iteration than the previous iteration, the heuristic decides to switch to AprioriTid.

This heuristic (like all heuristics) is not watertight, 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 1000 disjoint frequent patterns of size 5. If we merge these two datasets, we get $\binom{20}{3} + 1000\binom{5}{3} = 11140$ patterns of size 3 and $\binom{20}{4} + 1000\binom{5}{4} = 9845$ 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 decision is premature, however, because the number of new patterns in each pass will start growing exponentially afterwards.

Also, current state-of-the-art algorithms for frequent itemset mining, such as Opportunistic Project [Liu et al. 2002] and DCI [Orlando et al. 2002] use several techniques within the same algorithm and switch between these techniques using several simple, but not watertight heuristics. Both of these algorithms perform an Apriori-like levelwise generation in the first iterations, until the *active* part of the database can be stored into main memory. Again, the decision to perform this switch is mainly dependent on the number of candidate itemsets that can still be generated.

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 [Agrawal and Srikant 1994a; Agrawal et al. 1996], 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 3.1. A *candidate pattern* for L is a pattern (of size larger than k) of which all k-subsets are in L. For a given $p > 0$, we denote the set of all size-$k + p$ candidate patterns for L by $C_{k+p}(L)$.

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 [1968].)

LEMMA 3.2. *Given n and k, there exists a unique representation*

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

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

This representation is called the *k-canonical representation of n* and can be computed as follows: Find the integer m_k satisfying $\binom{m_k}{k} \leq n < \binom{m_k+1}{k}$, then find the integer m_{k-1} satisfying $\binom{m_{k-1}}{k-1} \leq n - \binom{m_k}{k} < \binom{m_{k-1}+1}{k-1}$, and so on, until $n - \binom{m_k}{k} - \binom{m_{k-1}}{k-1} - \cdots - \binom{m_r}{r}$ is zero. In order to compute this in practice, we only need the combinations $\binom{i}{j}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, k$. These can be computed by filling up Pascal's triangle, which needs $n \times k$ arithmetical operations.

We now establish:

THEOREM 3.3. *If*

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

in k-canonical representation, then

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

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

PROOF. Suppose, for the sake of contradiction, that

$$|C_{k+p}(L)| \geq \binom{m_k}{k+p} + \binom{m_{k-1}}{k-1+p} + \cdots + \binom{m_{s+1}}{s+p+1} + \binom{s+p}{s+p}.$$

Note that this is in $k + p$-canonical representation. A theorem by Kruskal and Katona [Frankl 1984; Katona 1968; Kruskal 1963] says that if the above assumption on the the cardinality of $C_{k+p}(L)$ holds, then the cardinality of L must be larger or equal than

$$\binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \cdots + \binom{m_{s+1}}{s+1} + \binom{s+p}{s}.$$

But this is impossible, because

$$\begin{aligned}
|L| &= \binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \cdots + \binom{m_{s+1}}{s+1} + \binom{m_s}{s} + \cdots + \binom{m_r}{r} \\
&\leq \binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \cdots + \binom{m_{s+1}}{s+1} + \sum_{1 \leq i \leq s} \binom{i+p-1}{i} \\
&< \binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \cdots + \binom{m_{s+1}}{s+1} + \sum_{0 \leq i \leq s} \binom{i+p-1}{i} \\
&= \binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \cdots + \binom{m_{s+1}}{s+1} + \binom{s+p}{s}.
\end{aligned}$$

The first inequality follows from the observation that $m_s \leq s + p - 1$ implies $m_i \leq i + p - 1$ for all $i = s, s-1, \ldots, r$. The last equality follows from a well-known binomial identity. □

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 3.4 (TIGHTNESS). *The upper bound provided by Theorem* 3.3 *is* tight: *for any given n and k there always exists an L with $|L| = n$ such that for any given p, $|C_{k+p}(L)| = KK_k^{k+p}(|L|)$.*

PROOF. Let us write a finite set of natural numbers as a string of natural numbers by writing its members in decreasing order. We can then compare two such sets by comparing their strings in lexicographic order. The resulting order on the sets is known as the *colexicographic* (or *colex*) order. An intuitive proof of the Kruskal-Katona theorem, based on this colex order, was given by Bollobás [1986]. Let

$$\binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \cdots + \binom{m_r}{r}$$

be the k-canonical representation of n. Then, Bollobás has shown that all $k-p$-subsets of the first n k-sets of natural numbers in colex order, are exactly the first

$$\binom{m_k}{k-p} + \binom{m_{k-1}}{k-1-p} + \cdots + \binom{m_s}{r-s}$$

$k-p$-sets of natural numbers in colex order, with s the smallest integer such that $s > p$. Using the same reasoning as above, we can conclude that all $k+p$-supersets of the first n k-sets of natural numbers in colex order are exactly the first $KK_k^{k+p}(n)$ $k+p$-sets of natural numbers in colex order. □

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

Example 3.5. 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 $\binom{5}{3} + \binom{3}{2}$, 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 3.6. *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 3.3, it is immediate that μ also tells us the last level before which KK becomes zero. Formally:

PROPOSITION 3.7.

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

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

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

4. IMPROVED UPPER BOUNDS

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** $p[1], p[2], \ldots, p[k], q[k]$
**from** $L_k\ p, L_k\ q$
**where** $p[1] = q[1], \ldots, p[k-1] = q[k-1],\ p[k] < q[k]$.
```

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

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

We can now simply combine this new upper bound with the KK upper bound, obtaining:

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

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:

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

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

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

THEOREM 4.1.

$$|C_{k+p}(L)| \leq KK^*_{k+p}(L).$$

PROOF. By induction on k. The base case $k = 1$ is clear. For $k > 1$, it suffices to show that for all $p > 0$

$$C_{k+p}(L) \subseteq \bigcup_{x \in I} C_{k+p-1}(L^x) + x. \qquad (1)$$

(For any set of patterns H, we denote $\{h \cup \{x\} \mid h \in H\}$ by $H + x$.)

From the above containment we can conclude

$$\begin{aligned}|C_{k+p}(L)| &\leq |\bigcup_{x\in I} C_{k+p-1}(L^x) + x| \\ &\leq \sum_{x\in I} |C_{k+p-1}(L^x) + x| \\ &= \sum_{x\in I} |C_{k+p-1}(L^x)| \\ &\leq \sum_{x\in I} KK^*_{k+p-1}(L^x)\end{aligned}$$

where the last inequality is by induction.

To show (1), we need to show that for every $p > 0$ and every $s \in C_{k+p}(L)$, $s - \{x\} \in C_{k+p-1}(L^x)$, where $x = \min s$. This means that every subset of $s - \{x\}$ of size $k - 1$ must be an element of L^x. Let $s - \{x\} - \{y_1, \ldots, y_p\}$ be such a subset. This subset is an element of L^x iff $s - \{y_1, \ldots, y_p\} \in L$ and $x = \min(s - \{y_1, \ldots, y_p\})$. The first condition follows from $s \in C_{k+p}(L)$, and the second condition is trivial. Hence the theorem. □

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 4.2. Let L consist of $\{5, 7, 8\}$ and $\{5, 8, 9\}$ plus all 19 3-subsets of $\{1, 2, 3, 4, 5\}$ and $\{3, 4, 5, 6, 7\}$. 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,

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

and

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

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 maxsize(L). In analogy with Proposition 3.7, we define:

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

We then have:

PROPOSITION 4.3.

$$\text{maxsize}(L) \leq \mu_k^*(L) \leq \mu_k(L).$$

We finally use Theorem 4.1 for improving the upper bound KK_k^{total} on the total number of candidate patterns. We define:

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

Then we have:

PROPOSITION 4.4. *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. GENERALIZED UPPER BOUNDS

The upper bounds presented in the previous sections work well for algorithms that generate and test candidate patterns of one specific size at a time. However, several algorithms generate and test patterns of different sizes within the same pass of the algorithm [Brin et al. 1997; Bayardo 1998; Toivonen 1996]. For example, if the given database does not fit into main memory or even is too large to be scanned multiple times, one can first try to find an approximation of the collection of frequent itemsets by mining only a sample of the database, which can be stored into main memory [Toivonen 1996]. After this, a lot of frequent itemsets might still not have been found. Nevertheless, if the sample was correctly chosen, and the support threshold lowered, as described in Toivonen [1996], there exists a good chance that one can generate all possible remaining candidates immediately, such that only a single additional pass over the massive database is needed. In order to prevent a combinatorial explosion of the number of candidate itemsets, we should be able to compute our upper bounds based on the frequency and infrequency information given by the, in the sample, generated collection of itemsets of different lengths. Also, one of the most successful algorithms that generates only all maximal itemsets, MaxMiner [Bayardo 1998], uses a look-ahead and support lower bounding technique such that at a given stage, itemsets of different sizes are known to be frequent or not. As also presented in that paper, these techniques can also be added into Apriori resulting in significant performance improvements. Again, it would be useful if our upper bounds could be adapted to take this valuable information into account.

Since our upper bound is solely based on the patterns of a certain length k, it does not use information about patterns of length larger than k. Nevertheless, these larger sets could give crucial information. More specifically, suppose we have generated all frequent patterns of size k, and we also already know in advance that a certain set of size larger than k is not frequent. Our upper bound on the total number of candidate patterns that can still be generated, would disregard this information. We will therefore generalize our upper bound such that it will also incorporate this additional information.

5.1 Generalized *KK*-Bound

From now on, L is some family of sets of patterns $L_k, L_{k+1}, \ldots, L_{k+q}$ that are known to be frequent, such that L_{k+p} contains patterns of size $k+p$, and all $k+p-1$-subsets of all patterns in L_{k+p} are in L_{k+p-1}. We denote by $|L|$ the sequence of numbers $|L_k|, |L_{k+1}|, \ldots, |L_{k+q}|$.

Similarly, let I be a family of sets of patterns $I_k, I_{k+1}, \ldots, I_{k+q}$ that are known to be infrequent, such that I_{k+p} contains patterns of size $k+p$ and all $k+p-1$-subsets of all patterns in I_{k+p} are in L_{k+p-1}. We denote by $|I|$ the sequence of numbers $|I_k|, |I_{k+1}|, \ldots, |I_{k+q}|$. Note that for each $p \geq 0$, L_{k+p} and I_{k+p} are disjoint.

Before we present the general upper bounds, we also generalize our notion of a candidate pattern.

Definition 5.1. A *candidate pattern for* (L, I) of size $k+p$ is a pattern that is not in L_{k+p} or I_{k+p}, all of its k-subsets are in L_k, and none of its subsets of size larger than k is included in $I_k \cup I_{k+1} \cup \cdots \cup I_{k+q}$. For a given p, we denote the set of all $k+p$-size candidate patterns for (L, I) by $C_{k+p}(L, I)$.

We note:

LEMMA 5.2.

$$C_{k+p}(L, I) = \begin{cases} C_{k+1}(L_k) \setminus (L_{k+1} \cup I_{k+1}) & \textit{if } p = 1; \\ C_{k+p}\big(C_{k+p-1}(L, I) \cup L_{k+p-1}\big) \setminus (L_{k+p} \cup I_{k+p}) & \textit{if } p > 1. \end{cases}$$

PROOF. The case $p = 1$ is clear. For $p > 1$, we show the inclusion in both directions.

$\supseteq$ For every set in $C_{k+p}\big(C_{k+p-1}(L, I) \cup L_{k+p-1}\big)$, we know that all of its k-subsets are always contained in a $k+p-1$ subset, and these are in $C_{k+p-1}(L, I) \cup L_{k+p-1}$. By definition, we know that for every set in $C_{k+p-1}(L, I)$, all of its k-subsets are in L_k. Also, for every set in L_{k+p-1}, all of its k-subsets are in L_k. By definition, for every set in $C_{k+p-1}(L, I)$, all of its $k+p-i$-subsets are not in I_{k+p-i}. Also, for every set in L_{k+p-1}, all of its $k+p-i$-subsets are in L_{k+p-i} hence they are not in I_{k+p-i} since they are disjoint. By definition, none of the patterns in $L_{k+p} \cup I_{k+p}$ are in $C_{k+p}(L, I)$.

$\subseteq$ It suffices to show that for every set in $C_{k+p}(L, I)$, every $k+p-1$-subset s is in $C_{k+p-1}(L, I) \cup L_{k+p-1}$. Obviously, this is true, since if it is not already in L_{k+p-1}, all k-subsets of s must still be in L_k, s cannot be in I_{k+p-1} and none of its subsets can be in any $I_{k+p-\ell}$ with $\ell > 1$. □

Hence, we define

$$gKK_k^{k+p}(|L|,|I|) := \begin{cases} KK_k^{k+1}(|L_k|) - |L_{k+1}| - |I_{k+1}| & \text{if } p = 1; \\ KK_{k+p-1}^{k+p}(gKK_k^{k+p-1}(|L|,|I|) + |L_{k+p-1}|) - |L_{k+p}| - |I_{k+p}| & \text{if } p > 1, \end{cases}$$

and obtain:

THEOREM 5.3.

$$|C_{k+p}(L,I)| \le gKK_k^{k+p}(|L|,|I|) \le KK_k^{k+p}(|L_k|) - |L_{k+p}| - |I_{k+p}|.$$

PROOF. The first inequality is clear by Lemma 5.2. The second inequality is by induction on p. The base case $p = 1$ is by definition. For $p > 1$, we have:

$$\begin{aligned} gKK_k^{k+p}(|L|,|I|) &= KK_{k+p-1}^{k+p}\big(gKK_k^{k+p-1}(|L|,|I|) + |L_{k+p-1}|\big) \\ &\quad - |L_{k+p}| - |I_{k+p}| \\ &\le KK_{k+p-1}^{k+p}\big(KK_k^{k+p-1}(|L_k|) - |I_{k+p-1}|\big) - |L_{k+p}| - |I_{k+p}| \\ &\le KK_{k+p-1}^{k+p}\big(KK_k^{k+p-1}(|L_k|)\big) - |L_{k+p}| - |I_{k+p}| \\ &= KK_k^{k+p}(|L_k|) - |L_{k+p}| - |I_{k+p}| \end{aligned}$$

where the first inequality is by induction and because of the monotonicity of *KK*, the second inequality also because of the monotonicity of *KK* and the last equality follows from

$$KK_k^{k+p}(|L_k|)) = KK_{k+p-1}^{k+p}\big(KK_k^{k+p-1}(|L_k|)\big). \qquad \square$$

Again, we can also generalize the upper bound on the maximal size of a candidate pattern, denoted by maxsize(L, I), and the upper bound on the total number of candidate patterns, both also incorporating (L, I):

$$g\mu(|L|,|I|) := k + \min\{p \mid gKK_k^{k+p}(|L|,|I|) = 0\} - 1$$

$$gKK_k^{\text{total}}(|L|,|I|) := \sum_{p\ge 1} gKK_k^{k+p}(|L|,|I|).$$

We obtain:

PROPOSITION 5.4.

$$\text{maxsize}(L,I) \le g\mu(|L|,|I|) \le \mu(|L|).$$

PROPOSITION 5.5. *The total number of candidate patterns that can be generated from (L, I) is bounded by $gKK_k^{\text{total}}(|L|,|I|)$. Moreover,*

$$gKK_k^{\text{total}}(|L|,|I|) \le KK_k^{\text{total}}(|L_k|).$$

Example 5.6. Suppose L_3 consists of all subsets of size 3 of the set $\{1, 2, 3, 4, 5, 6\}$. Now assume we already know that I_4 contains patterns $\{1, 2, 3, 4\}$ and

$\{3, 4, 5, 6\}$. The KK upper bound presented in the previous section would estimate the number of candidate patterns of sizes 4, 5, and 6 to be at most $\binom{6}{4} = 15$, $\binom{6}{5} = 6$, and $\binom{6}{6} = 1$ respectively. Nevertheless, using the additional information, gKK can already reduce these numbers to 13, 3, and 0. Also, μ would predict the maximal size of a candidate pattern to be 6, while $g\mu$ can already predict this number to be at most 5. Similarly, KK_{total} would predict the total number of candidate patterns that can still be generated to be at most 22, while gKK_{total} can already deduce this number to be at most 16.

5.2 Generalized KK^*-Bound

Using the generalized basic upper bound, we can now also generalize our improved upper bound KK^*. For an arbitrary item x, define the family of sets L^x as $L^x_k, L^x_{k+1}, \ldots, L^x_{k+q}$, and I^x as $I^x_k, I^x_{k+1}, \ldots, I^x_{k+q}$. We define:

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

We then have:

THEOREM 5.7.

$$|C_{k+p}(L, I)| \leq gKK^*_{k+p}(L, I) \leq KK^*_{k+p}(L_k) - |L_{k+p}| - |I_{k+p}|.$$

PROOF. The proof of the first inequality is similar to the proof of Theorem 4.1, but we now need to show that for all $p > 0$,

$$C_{k+p}(L, I) \subseteq \bigcup_{x\in I} C_{k+p-1}(L^x, I^x) + x.$$

Therefore, we need to show for every $s \in C_{k+p}(L, I)$, $s - \{x\} \in C_{k+p-1}(L^x, I^x)$, where $x = \min s$. First, this means that every subset of $s - \{x\}$ of size $k - 1$ must be in L^x_k. Let $s - \{x\} - \{y_1, \ldots, y_p\}$ be such a subset. This subset is an element of L^x_k if and only if $s - \{y_1, \ldots, y_p\} \in L_k$ and $x = \min(s - \{y_1, \ldots, y_p\})$. The first condition follows from $s \in C_{k+p}(L, I)$, and the second condition is trivial. Second, we need to show that $s - \{x\}$ is not in L^x_{k+p}. Since $s \in C_{k+p}(L, I)$, s is not in L_{k+p}, hence $s - \{x\}$ cannot be in L^x_{k+p}. Finally, we need to show that none of the subsets of $s - \{x\}$ of size greater than $k - 1$ are in $I^x_{k+1}, \ldots, I^x_{k+p-1}$. Let $s - \{x\} - \{y_1, \ldots, y_m\}$ be such a subset. Since $s \in C_{k+p}(L, I)$, $s - \{y_1, \ldots, y_m\}$ is not in I_{k+p-m}, hence $s - \{x\} - \{y_1, \ldots, y_m\}$ cannot be in I^x_{k+p-m}.

We prove the second inequality by induction on k. The base case $k = 1$ is clear. For all $k > 0$, we have

$$\begin{aligned}
&gKK^*_{k+p}(L, I)\\
&\quad = \min\left\{gKK_k^{k+p}(|L|, |I|), \sum_{x\in I} gKK^*_{k+p-1}(L^x, I^x)\right\}\\
&\quad \le \min\left\{KK_k^{k+p}(|L_k|) - |L_{k+p}| - |I_{k+p}|, \sum_{x\in I} KK^*_{k+p-1}(L^x_k) - |L^x_{k+p}| - |I^x_{k+p}|\right\}\\
&\quad = \min\left\{KK_k^{k+p}(|L|), \sum_{x\in I} KK^*_{k+p-1}(L^x)\right\} - |L_{k+p}| - |I_{k+p}|\\
&\quad = KK^*_{k+p}(L_k) - |L_{k+p}| - |I_{k+p}|
\end{aligned}$$

where the left hand side of the minimum in the inequality is by Theorem 5.3 and the right hand side is by induction. □

Again, we get an upper bound on maxsize(L, I):

$$g\mu^*(L, I) := k + \min\{p \mid gKK^*_{k+p}(L, I) = 0\} - 1,$$

and on the total number of candidate patterns that can still be generated:

$$gKK^*_{\text{total}}(L, I) := \sum_{p\ge 1} gKK^*_{k+p}(L, I).$$

We then have the following propositions analogous to 4.3 and 4.4:

PROPOSITION 5.8.

$$\text{maxsize}(L, I) \le g\mu^*(L, I) \le \mu^*(L).$$

PROPOSITION 5.9. *The total number of candidate patterns that can be generated from (L, I) is bounded by $gKK^*_{\text{total}}(L, I)$. Moreover,*

$$gKK^*_{\text{total}}(L, I) \le KK^*_{\text{total}}(L_k).$$

Example 5.10. Consider the same set of patterns as in the previous example. L_3 consists of all subsets of size 3 of the set $\{1, 2, 3, 4, 5, 6\}$ and $\{1, 2, 3, 4\}$ and $\{3, 4, 5, 6\}$ are included in I_4. The KK^* upper bound presented in the previous section would also estimate the number of candidate patterns of sizes 4, 5, and 6 to be at most $\binom{6}{4} = 15$, $\binom{6}{5} = 6$, and $\binom{6}{6} = 1$ respectively. Nevertheless, using the additional information, gKK^* can perfectly predict these numbers to be 13, 2, and 0. Again, μ^* would predict the maximal size of a candidate pattern to be 6, while $g\mu^*$ can already predict this number to be at most 5. Similarly, KK^*_{total} would predict the total number of candidate patterns that can still be generated to be at most 22, while gKK^*_{total} can already deduce this number to be at most 15.

6. EFFICIENT IMPLEMENTATION

For simplicity reasons, we will restrict ourselves to the explanation of how the improved upper bounds can be implemented. The proposed implementation can be easily extended to support the computation of the generalized upper bounds.

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. [1997]. This trie structure makes it cheap and straightforward to implement the computation of all upper bounds. Indeed, a top-level subtrie (rooted at some singleton pattern $\{x\}$) represents exactly the set L^x we defined in Section 4. Every top-level subtrie of this subtrie (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 3.6), KK^{ℓ}_{k-d} for any $\ell \leq \mu_{k-d}$ (cf. Theorem 3.3), hence also KK^{total}_{k-d} (cf. Proposition 3.8). 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+p}_{k-d} 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 conclude that the time needed to compute $KK^*_{k+p}(L)$, in terms of the number of arithmetical operations, is linearly proportional to the time needed to construct L in the first place.

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.

7. BRUTE FORCE COMPUTATION

As already mentioned in the introduction, instead of using any of the presented upper bound computations, one could also actually generate all possible candidate itemsets using the Apriori candidate generation technique and simply count them. Obviously, this brute force method doesn't give an upper bound, but the exact number of candidate itemsets. On the other hand, when a combinatorial explosion occurs, this method could take a prohibitive amount of space and time, which is exactly what the presented upper bounding techniques try to prevent. Nevertheless, when one is not necessarily interested in the exact number of possible candidate itemsets, but simply wants to know whether this number is above a certain threshold, it might still be feasible to compute in a brute force manner. Indeed, we can prevent a combinatorial explosion by simply stopping the generation and counting as soon as the number of generated candidate itemsets reaches the given threshold. Then, in the worst case, the only drawback is the time and space needed to generate exactly that maximum number of itemsets.

This brute force method can still be implemented in several different manners. A first option is to perform the exact breadth-first, levelwise candidate generation mechanism as is used by Apriori. Second, it might be more efficient to generate all candidate itemsets immediately in a depth-first manner and in colex order, such that all subsets of a candidate itemset have been generated earlier. While the breadth-first technique only has to store the itemsets of size k and $k + 1$ in each iteration, the depth-first technique might be faster, but it has the major drawback that it must store all generated candidates since they might be necessary to check monotonicity of any itemset that is generated later. This problem, however, can also be resolved. Suppose we have generated all frequent itemsets up to depth k and we are generating all possible candidate itemsets of size larger than k. At the generation of a candidate $k + p$-itemset, we normally test whether all its subsets of size $k + p - 1$ are also known to be frequent, or in this case, potentially frequent. We know, however, for $p > 1$, that those sets are candidate itemsets themselves, and they have been generated on the basis of the frequent k-itemsets. Hence, instead of testing whether all $k+p-1$ itemsets are (potentially) frequent, we can limit ourselves to testing whether all k-subsets of the $k + p$ candidate itemset are frequent. In that way, we do not have to store all depth-first generated candidates, but only those on the immediate recursion path to the current itemset. Although this third technique might use much less memory, it can consume a lot more time. Indeed, testing whether all immediate subsets of a $k + p$-itemset exist, takes only $k + p$ operations, while testing whether all k subsets of a $k + p$-itemset exist, takes $\binom{k+p}{k}$ operations.

All three of these brute force generating and counting methods will be compared to the computation of the presented upper bounds in the following section.

8. EXPERIMENTAL EVALUATION

All experiments were performed on a 400 MHz Sun Ultra Sparc with 512 MB main memory, running Sun Solaris 8. The algorithm was implemented in C++ and uses the GNU MP library for arbitrary-length integers (`http://directory.fsf.org/gnump.html`).

Data sets. We have experimented using three real data sets, of which two are publicly available, and one synthetic data set generated by the program provided by the Quest research group at IBM Almaden [Agrawal and Srikant 1994c]. The mushroom data set contains characteristics of various species of mushrooms, and was originally obtained from the UCI repository of machine learning databases [Blake and Merz 1998]. The BMS-WebView-1 data set contains several months worth of clickstream data from an e-commerce web site, and is made publicly available by Blue Martini Software [Kohavi et al. 2000]. The basket data set contains transactions from a Belgian retail store, but can

Table I. Database Characteristics

Data Set	#Items	#Transactions	MinSup	#Iterations	Time
T40I10D100K	1000	100000	700	18	1700 s
mushroom	120	8124	813	16	663 s
BMS-Webview-1	498	59602	36	15	86 s
basket	13103	41373	5	11	43 s

unfortunately not be made publicly available. Table I shows the number of items and the number of transactions in each data set. The table additionally shows the minimal support threshold we used in our experiments for each data set, together with the resulting number of iterations and the time (in seconds) which the Apriori algorithm needed to find all frequent patterns.

The results from the experiment with the real data sets were not immediately as good as the results from the synthetic data set. 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 (namely L^x), we win by incorporating the structure of the current collection of frequent patterns, but we also lose some information. Indeed, whenever we recursively restrict ourselves to a subtrie L^x, then for every candidate pattern s with $x = \min s$, we lose the information about exactly one subpattern in L, namely $s - x$.

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 [Bayardo 1998; Brin et al. 1997; Agarwal et al. 2001], is to force the most frequent items to appear in the most candidate patterns, by reordering the single item patterns in increasing order of frequency.

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

Efficiency. 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 currently known frequent sets. We have shown experimentally that T scales linearly with n. Moreover, the constant factor in our implementation is very small (around 0.00001). We ran several experiments using the different data sets and varying minimal support thresholds. After every pass of the algorithm, we registered the number of known frequent sets and the time spent to compute all upper bounds, resulting in 145 different data points. Figure 1 shows these results.

In Figure 2, we compare the performance of the three brute force candidate generation and count methods with the computation of the KK^*_{total} upper bound. For each level k, the total time spent on computing upper bounds until that level is shown. The line BF shows the time spent by the breadth-first method, the line DF shows this for the depth-first method, and the line DF_{mem} for the memory

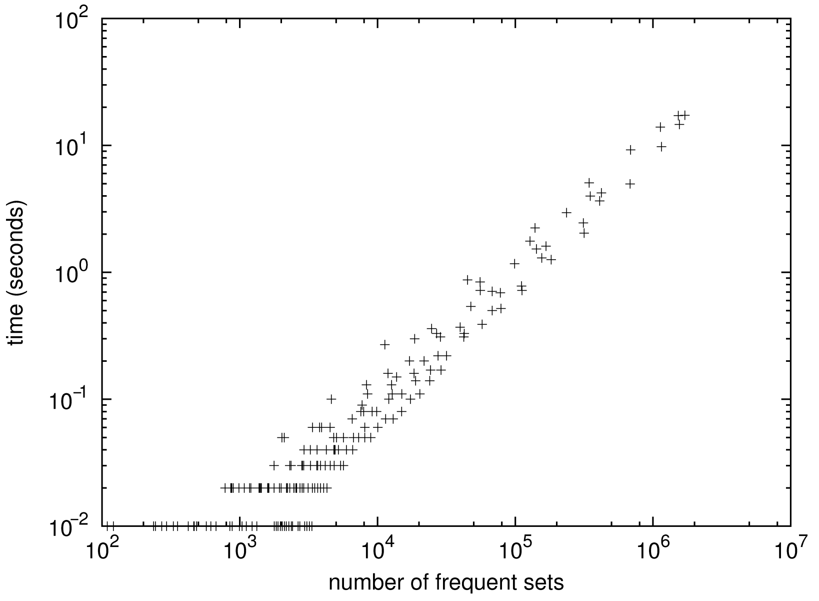
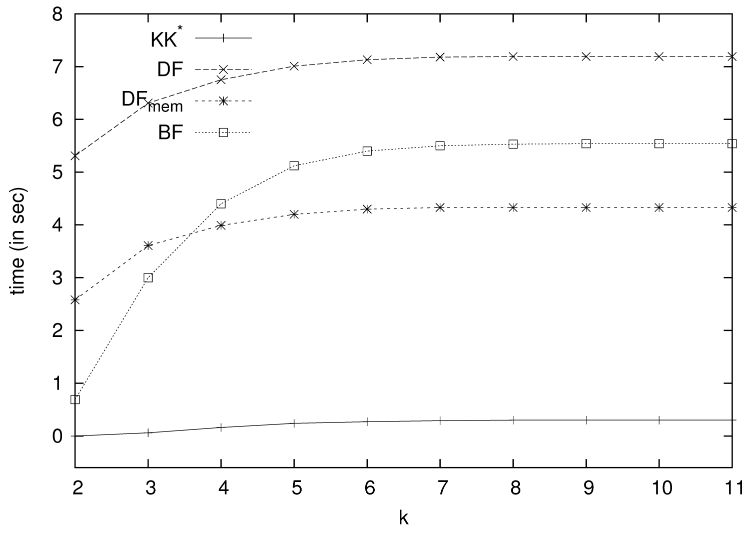


Fig. 1. Time needed to compute upper bounds is linear in the number of nodes.

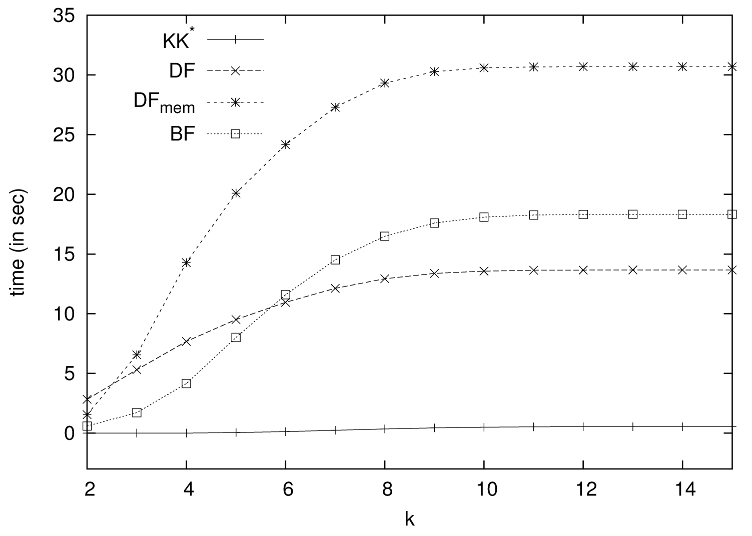
efficient depth-first method. For all three methods, the maximum number of candidate itemsets that was allowed to be generated was 500 000. In practice, however, a lot more candidate itemsets can reside in main memory, but higher thresholds would slow down these methods even more.

In Table II, we compare the memory consumption between the three brute force candidate generation and count methods with the computation of the KK^{*}_{total} upper bound. The values shown represent the total amount of memory used for the whole itemset mining algorithm, which is the same for all four implementations. Only the computation of the number of candidate itemsets differs, hence, is responsible for the differences in memory consumption.

As can be seen, the computation of the KK^{*}_{total} upper bound always gives the fastest answer, significantly faster than all three other brute force methods. Together with DF_{mem}, it also consumes the least amount of memory. Among the three brute force algorithms, it shows that DF is almost always the fastest, but it also consumes enormous amounts of memory as compared to the other techniques. On the other hand, while DF_{mem} also consumes the least memory, it is also the slowest of the three brute force techniques. The breadth-first method BF is always in between the two other breadth-first methods, for memory consumption as well as performance. None of the three methods come near the speed of the upper bound computation performance, of which again, it can be seen that the overhead of its computation is negligible. On the basket dataset, the methods behave somewhat differently. The high memory consumption is due to an enormous amount of candidate 3-itemsets, and DF_{mem} even outperforms DF by a few seconds because of the extra overhead by DF, which has to delete all generated candidates in a separate recursive call, while this is done immediately in DF_{mem}.



(a) basket



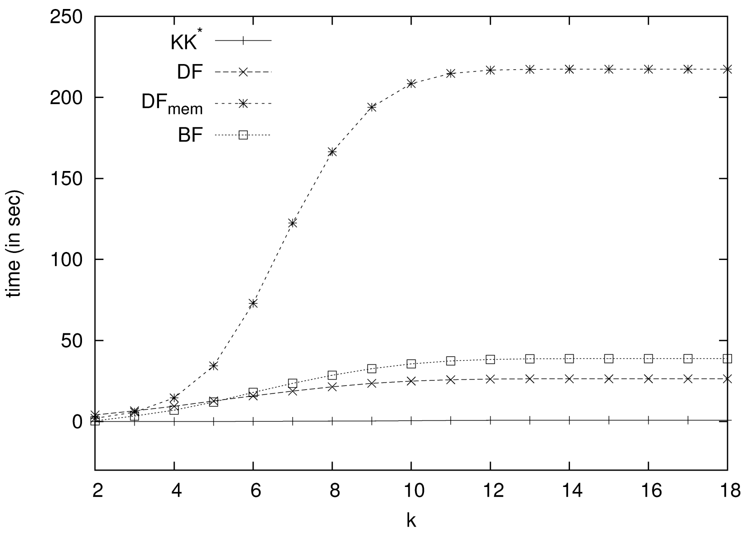
(b) BMS-Webview-1

Fig. 2. Time comparison with brute force computation.

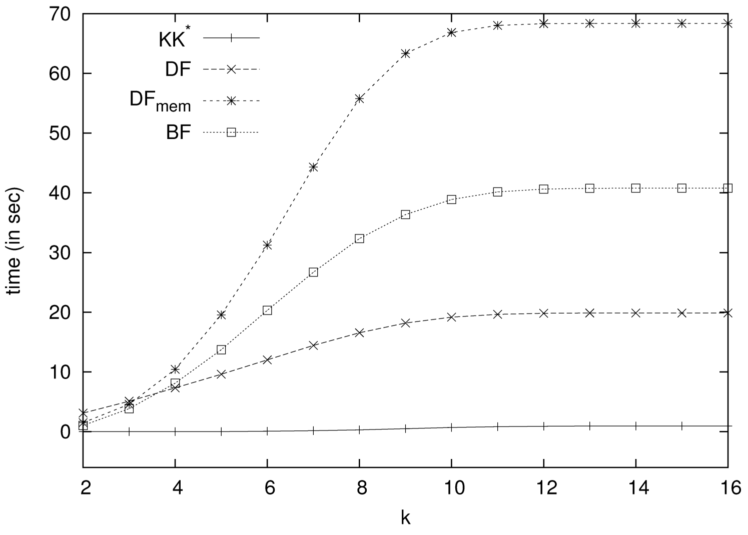
In the following experiments, we show that the presented upper bounds also give very accurate results with respect to the number of candidate patterns.

Upper bounds

—Figure 3 shows, after each level k, the computed upper bound KK and improved upper bound KK^* for the number of candidate patterns of size $k + 1$, as well as the actual number $|C_{k+1}|$ it turned out to be. We omitted the



(c) T40I10D100K

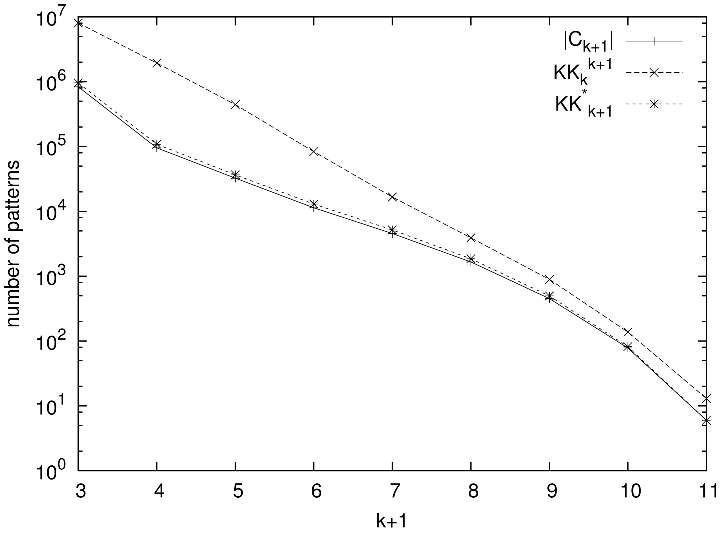


(d) mushroom

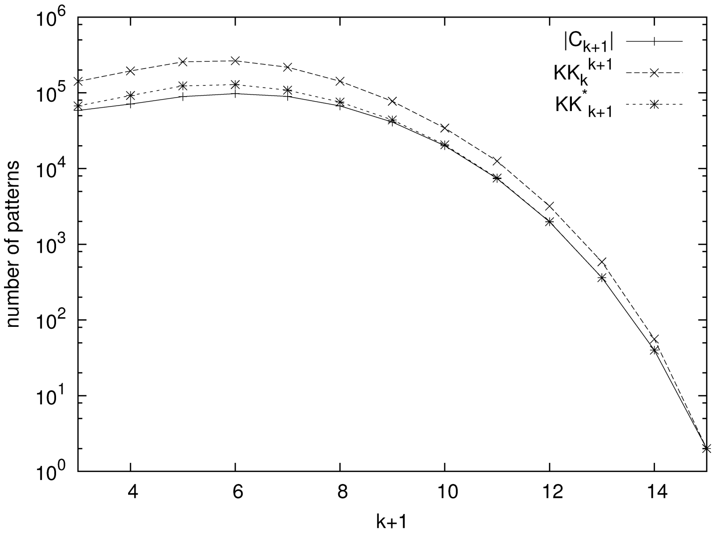
Fig. 2. Time comparison with brute force computation.

Table II. Memory Consumption Comparison with Brute Force Computation

Data Set	KK^*_{total}	BF	DF	DF$_{\text{mem}}$
T40I10D100K	27641	28984	43722	27641
mushroom	19663	27089	42066	19663
BMS-Webview-1	24163	34821	47215	24163
basket	58748	58778	61226	58778



(a) basket

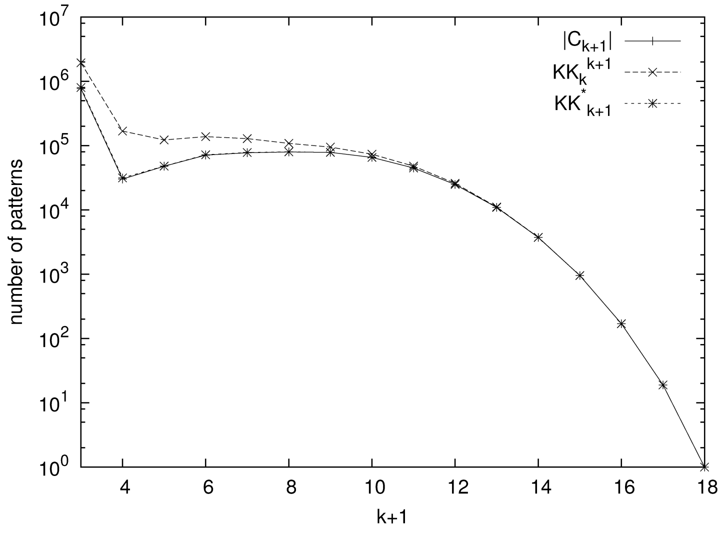


(b) BMS-Webview-1

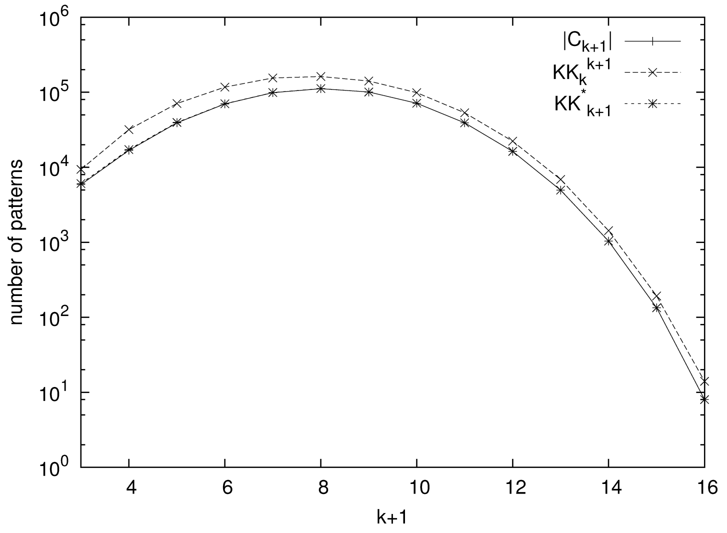
Fig. 3. Actual and estimated number of candidate patterns.

upper bound for $k+1=2$, since the upper bound on the number of candidate patterns of size 2 is simply $\binom{|L|}{2}$, with $|L|$ the number of frequent items.

—Figure 4 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 were effectively generated. Again, we omitted the upper bound for $k=1$, since this number is simply $2^{|L|}-|L|-1$, with $|L|$ the number of frequent items.



(c) T40I10D100K



(d) mushroom

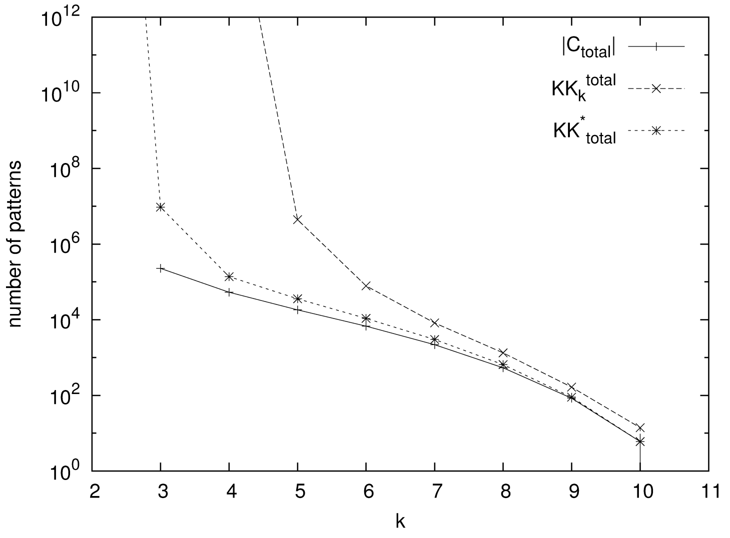
Fig. 3. Actual and estimated number of candidate patterns.

—Figure 5 shows the computed upper bounds μ and μ^* on the maximal size of a candidate pattern. Here we omitted the result for $k = 1$, since this number is exactly the number of frequent items.

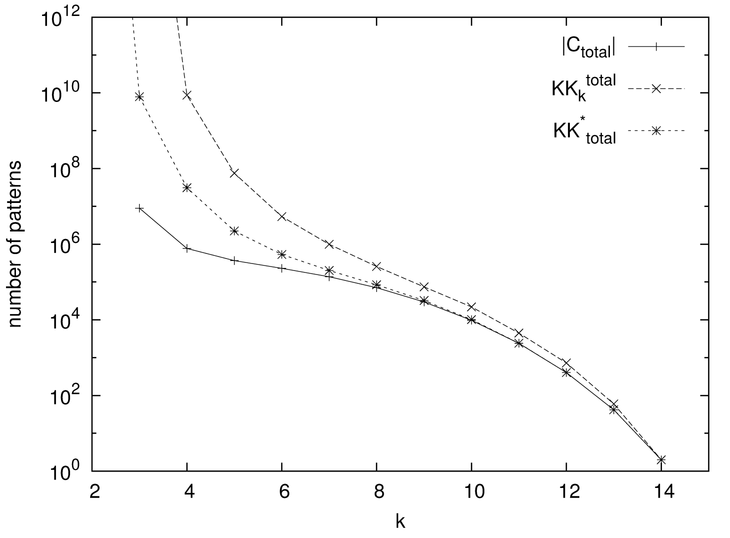
The results are pleasantly surprising:

—Note that the improvement of KK^* over KK, and of μ^* over μ, anticipated by our theoretical discussion, is indeed dramatic.

—Comparing the computed upper bounds with the actual numbers, we observe the high accuracy of the estimations given by KK^*. Indeed, the estimations



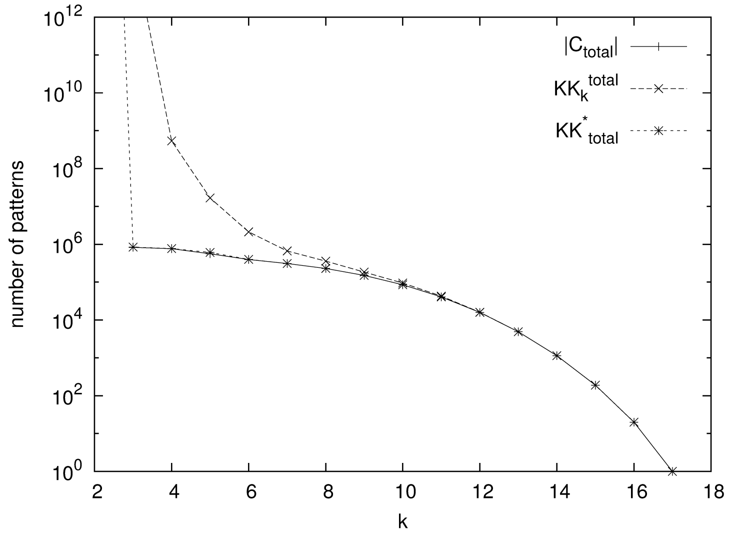
(a) basket



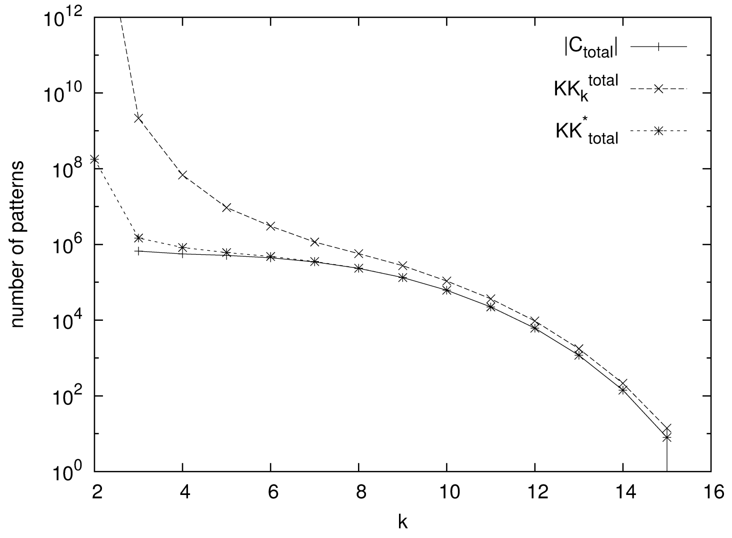
(b) BMS-Webview-1

Fig. 4. Actual and estimated total number of future candidate patterns.

of KK^*_{k+1} match almost exactly, the actual number of candidate patterns that has been generated at level $k + 1$. Also note that the number of candidate patterns in T40I10D100K is decreasing in the first four iterations and then increases again. This perfectly illustrates that the heuristic used for Apriori-Hybrid, as explained in the related work section, would not work on this data set. Indeed, any algorithm that exploits the fact that the current number of candidate patterns is small enough and there were fewer candidate patterns in the current iteration than in the previous iteration, would falsely interpret



(c) T40I10D100K

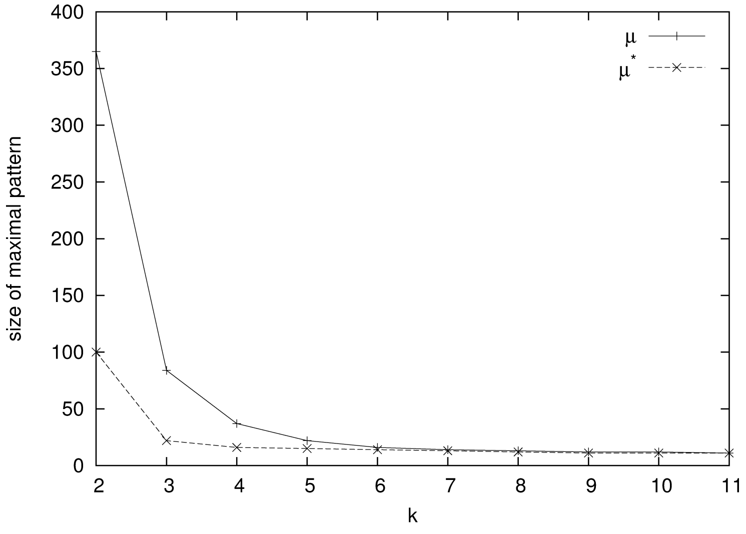


(d) mushroom

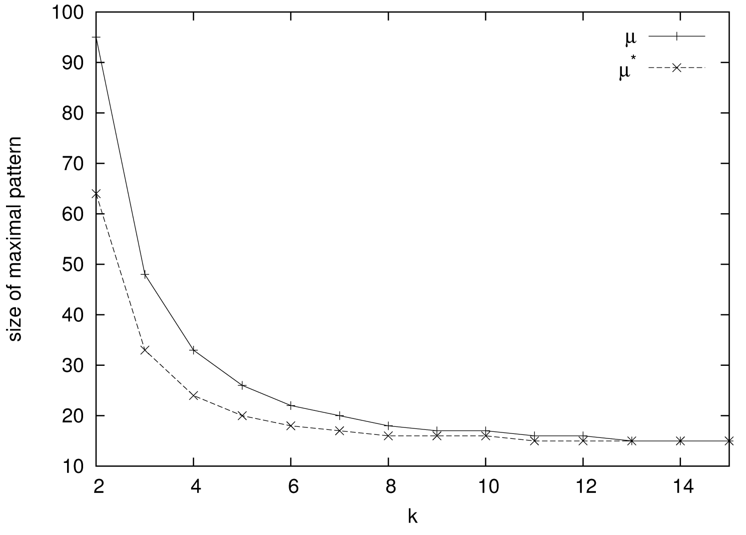
Fig. 4. Actual and estimated total number of future candidate patterns.

these observations, since the number of candidate patterns in the next iterations increases again. The presented upper bounds perfectly predict this increase.

—The upper bounds on the total number of candidate patterns are still very large when estimated in the first few passes, which is not surprising because at these initial stages, there is not much information yet. For the mushroom and the artificial data sets, the upper bound is almost exact when the frequent patterns of size 3 are known. For the basket data set, this result is



(a) basket



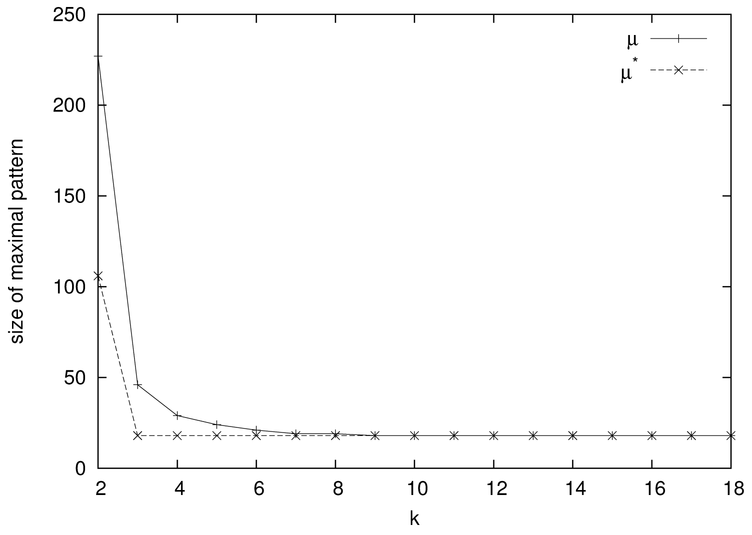
(b) BMS-Webview-1

Fig. 5. Estimated size of the largest possible candidate pattern.

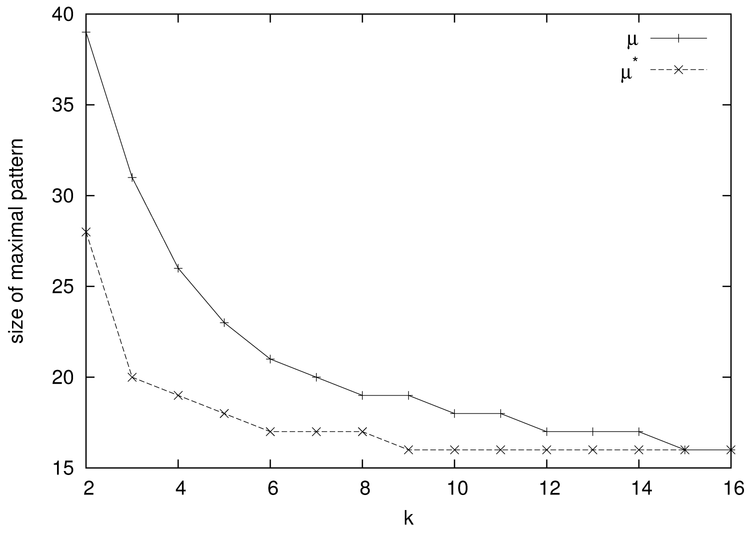
obtained when the frequent patterns of size 4 are known and size 6 for the BMS-Webview-1 data set.

—We also performed experiments for varying minimal support thresholds. The results obtained from these experiments were entirely similar to those presented above.

Combining iterations. As discussed in the Introduction, the proposed upper bound can be used to protect several improvements of the Apriori algorithm



(c) T40I10D100K

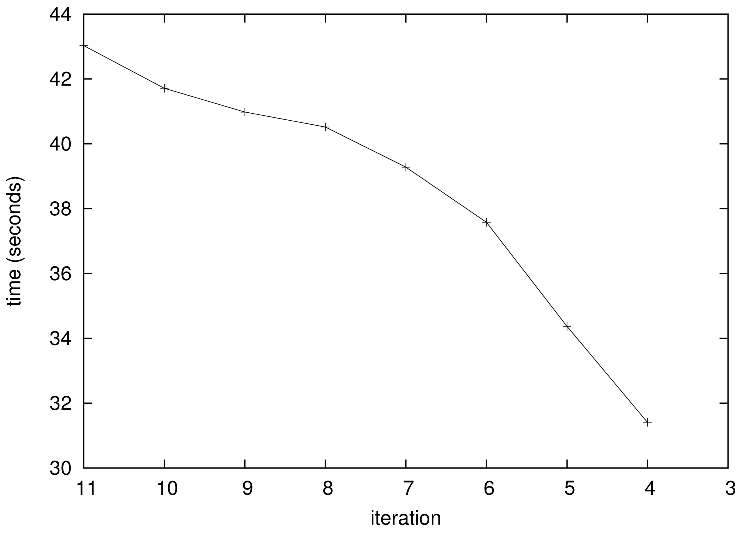


(d) mushroom

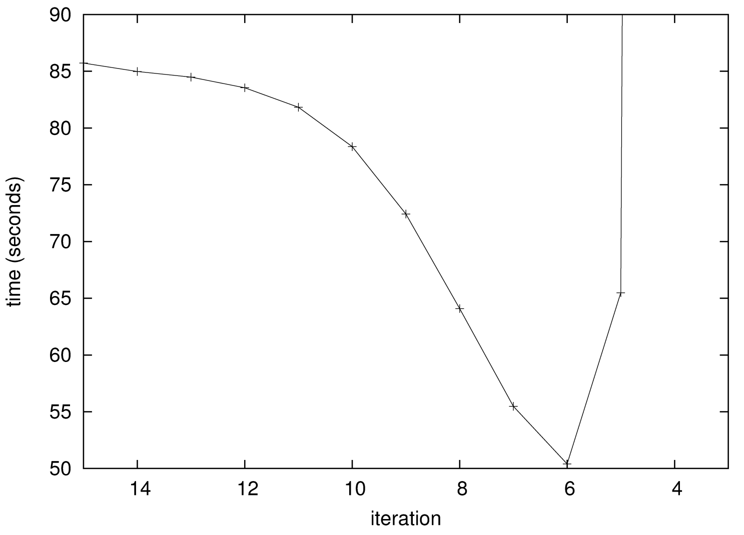
Fig. 5. Estimated size of the largest possible candidate pattern.

from generating too many candidate patterns. One such improvement tries to combine as many iterations as possible in the end, when only few candidate patterns can still be generated. We have incorporated this technique within our implementation of the Apriori algorithm.

We performed several experiments on each data set. Figure 6 illustrates the time spent by the adapted Apriori algorithm, when all iterations are combined after the iteration shown on the x-axis. More specifically, the x-axis shows the total number of iterations in which the algorithm completed, and the y-axis



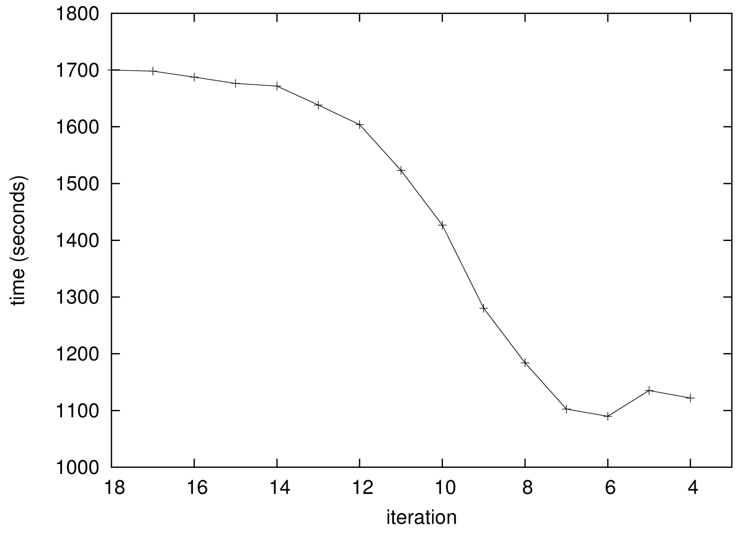
(a) basket



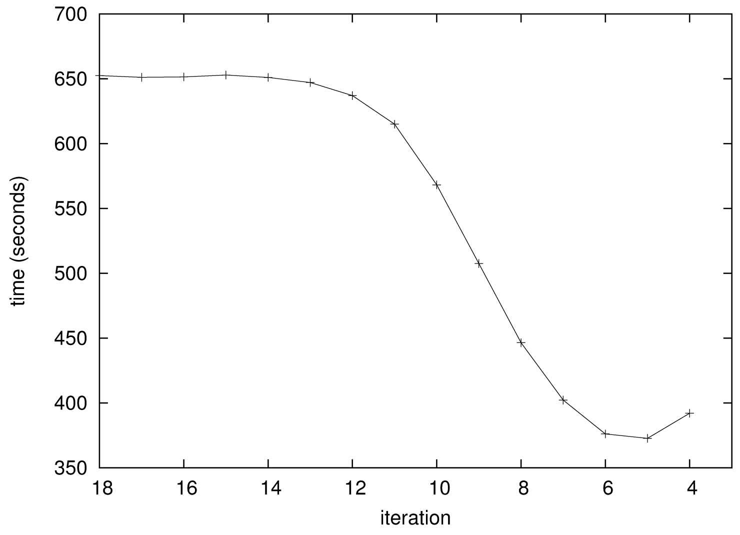
(b) BMS-Webview-1

Fig. 6. Combining iterations.

shows the total time the algorithm needed to complete. As can be seen, for all datasets, the algorithm can already combine all remaining iterations into one, very early in the algorithm. For example, if the maximum number of candidate pattern that is allowed to be generated is set to, for example, 5000000, then the BMS-Webview-1 dataset, which normally performs 15 iterations, would be reduced to 6 iterations (see Figure 4) and result in an optimal performance. If the algorithm already would have generated all remaining candidate patterns



(c) T40I10D100K



(d) mushroom

Fig. 6. Combining iterations.

in the fifth iteration, the number of candidate patterns that turned out to be infrequent was too large, such that the gain of reducing iterations has been consumed by the time needed to count all these candidate patterns. Nevertheless, it is still more effective than not combining any passes at all. If the generation of all candidate patterns occurs in even earlier iterations, although the upper bound predicted a too large number of candidate patterns, this number became indeed too large keep in main memory.

9. CONCLUSION

Motivated by several heuristics to reduce the number of database scans in the context of frequent pattern mining, we provide a hard and tight combinatorial upper bound on the number of candidate patterns and on the size of the largest possible candidate pattern, given a set of frequent patterns. Our findings are not restricted to a single algorithm, but can be applied to any frequent pattern mining algorithm which is based on the levelwise generation of candidate patterns. For example, using the standard Apriori algorithm, on which most frequent pattern mining algorithms are based, our experiments showed that these upper bounds can be used to considerably reduce the number of iterations of candidate generation, without taking the risk of a combinatorial explosion in the number of candidate patterns.

ACKNOWLEDGMENTS

We wish to thank Blue Martini Software for contributing the KDD Cup 2000 data, the machine learning repository librarians Catherine Blake and Chris Mertz for providing access to the mushroom data, and Tom Brijs for providing the Belgian retail market basket data.

REFERENCES

AGARWAL, R., AGGARWAL, C., AND PRASAD, V. 2000. Depth first generation of long patterns. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, R. Ramakrishnan, S. Stolfo, R. Bayardo, and I. Parsa, Eds. ACM Press, 108–118.

AGARWAL, R., AGGARWAL, C., AND PRASAD, V. 2001. A tree projection algorithm for generation of frequent itemsets. *J. Parallel Distrib. Comput. 61*, 3 (March), 350–371.

AGRAWAL, R., IMIELINSKI, T., AND SWAMI, A. 1993. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, P. Buneman and S. Jajodia, Eds. SIGMOD Record, vol. 22:2. ACM Press, 207–216.

AGRAWAL, R., MANNILA, H., SRIKANT, R., TOIVONEN, H., AND VERKAMO, A. 1996. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, Eds. MIT Press, 307–328.

AGRAWAL, R. AND SRIKANT, R. 1994a. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases*, J. Bocca, M. Jarke, and C. Zaniolo, Eds. Morgan Kaufmann, 487–499.

AGRAWAL, R. AND SRIKANT, R. 1994b. Fast algorithms for mining association rules. IBM Research Report RJ9839, IBM Alamaden Research Center, San Jose, California. June.

AGRAWAL, R. AND SRIKANT, R. 1994c. *Quest Synthetic Data Generator*. IBM Alamaden Research Center, `http://www.almaden.ibm.com/software/quest/Resources/index.shtml`.

BAYARDO, R. 1998. Efficiently mining long patterns from databases. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, L. Haas and A. Tiwary, Eds. SIGMOD Record, vol. 27:2. ACM Press, 85–93.

BLAKE, C. AND MERZ, C. 1998. *UCI Repository of machine learning databases*. University of California, Irvine, Dept. of Information and Computer Sciences, `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

BOLLOBÁS, B. 1986. *Combinatorics*. Cambridge University Press.

BOULICAUT, J.-F., BYKOWSKI, A., AND RIGOTTI, C. 2003. Free-sets: a condensed representation of Boolean data for frequency query approximation. *Data Mining and Knowledge Discovery 7*, 1, 5–22.

BRIN, S., MOTWANI, R., ULLMAN, J., AND TSUR, S. 1997. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. SIGMOD Record, vol. 26:2. ACM Press, 255–264.

Burdick, D., Calimlim, M., and Gehrke, J. 2001. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of the 17th International Conference on Data Engineering*. IEEE Computer Society, 443–452.

Frankl, P. 1984. A new short proof for the Kruskal–Katona theorem. *Discrete Mathematics 48*, 327–329.

Geerts, F., Goethals, B., and Van den Bussche, J. 2001. A tight upper bound on the number of candidate patterns. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, N. Cercone, T. Lin, and X. Wu, Eds. IEEE Computer Society, 155–162.

Goethals, B. and Zaki, M. J., Eds. 2003. *Proceedings of the Workshop on Frequent Itemset Mining Implementations (FIMI-03), Melbourne Florida, USA, November 19, 2003*. CEUR Workshop Proceedings, vol. 90. http://CEUR-WS.org/Vol-90/.

Han, J., Pei, J., and Yin, Y. 2000. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, W. Chen, J. Naughton, and P. Bernstein, Eds. SIGMOD Record, vol. 29:2. ACM Press, 1–12.

Katona, G. 1968. A theorem of finite sets. In *Theory Of Graphs*. Akadémia Kiadó, 187–207.

Kohavi, R., Brodley, C., Frasca, B., Mason, L., and Zheng, Z. 2000. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations 2*, 2, 86–98. http://www.ecn.purdue.edu/KDDCUP.

Kruskal, J. 1963. The number of simplices in a complex. In *Mathematical Optimization Techniques*. Univ. of California Press, 251–278.

Lin, D. and Kedem, Z. 1998. Pincer-search: A new algorithm for discovering the maximum frequent set. In *EDBT*, H.-J. Schek, F. Saltor, I. Ramos, and G. Alonso, Eds. Lecture Notes in Computer Science, vol. 1377. Springer, 105–119.

Liu, J., Pan, Y., Wang, K., and Han, J. 2002. Mining frequent item sets by opportunistic projection. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, D. Hand, D. Keim, and R. Ng, Eds. ACM Press, 229–238.

Orlando, S., Palmerini, P., Perego, R., and Silvestri, F. 2002. Adaptive and resource-aware mining of frequent sets. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, V. Kumar, S. Tsumoto, P. Yu, and N.Zhong, Eds. IEEE Computer Society, to appear.

Park, J., Chen, M.-S., and Yu, P. 1995. An effective hash based algorithm for mining association rules. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. SIGMOD Record, vol. 24:2. ACM Press, 175–186.

Pasquier, N., Bastide, Y., Taouil, R., and Lakhal, L. 1999. Discovering frequent closed itemsets for association rules. In *Proceedings of the 7th International Conference on Database Theory*, C. Beeri and P. Buneman, Eds. lncs, vol. 1540. Springer, 398–416.

Pei, J., Han, J., and Mao, R. 2000. Closet: An efficient algorithm for mining frequent closed itemsets. ACM SIGMOD'00 Workshop on Research Issues in Data Mining and Knowledge Discovery.

Savasere, A., Omiecinski, E., and Navathe, S. 1995. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21th International Conference on Very Large Data Bases*, U. Dayal, P. Gray, and S. Nishio, Eds. Morgan Kaufmann, 432–444.

Toivonen, H. 1996. Sampling large databases for association rules. In *Proceedings of the 22th International Conference on Very Large Data Bases*, T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, Eds. Kaufmann, 134–145.

Zaki, M. and Hsiao, C.-J. 2002. CHARM: An efficient algorithm for closed itemset mining. In *Proceedings of the Second SIAM International Conference on Data Mining*, R. Grossman, J. Han, V. Kumar, H. Mannila, and R. Motwani, Eds. SIAM.

Zaki, M., Parthasarathy, S., Ogihara, M., and Li, W. 1997. New algorithms for fast discovery of association rules. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, D. Heckerman, H. Mannila, and D. Pregibon, Eds. AAAI Press, 283–296.

Zheng, Z., Kohavi, R., and Mason, L. 2001. Real world performance of association rule algorithms. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, F. Provost and R. Srikant, Eds. ACM Press, 401–406.

Received July 2003; revised June 2004; accepted October 2004