Improving Direct Counting for Frequent Itemset Mining

Adriana Prado*, Cristiane Targa**, and Alexandre Plastino***

Department of Computer Science, Universidade Federal Fluminense,
Rua Passo da Pátria, 156 - Bloco E - 3º andar - Boa Viagem,
24210-240, Niterói, RJ, Brazil
{aprado, ctarga, plastino}@ic.uff.br
http://www.ic.uff.br

Abstract. During the last ten years, many algorithms have been proposed to mine frequent itemsets. In order to fairly evaluate their behavior, the *IEEE/ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)* has been recently organized. According to its analysis, kDCI++ is a state-of-the-art algorithm. However, it can be observed from the *FIMI'03* experiments that its efficient behavior does not occur for low minimum supports, specially on sparse databases. Aiming at improving kDCI++ and making it even more competitive, we present the kDCI-3 algorithm. This proposal directly accesses candidates not only in the first iterations but specially in the third one, which represents, in general, the highest computational cost of kDCI++ for low minimum supports. Results have shown that kDCI-3 outperforms kDCI++ in the conducted experiments. When compared to other important algorithms, kDCI-3 enlarged the number of times kDCI++ presented the best behavior.

1 Introduction

Association rules represent an important type of information extracted from data mining processes that describe interesting relationships among data items of a specific knowledge domain. *Market basket analysis* is the typical application of *association rule mining* (ARM) and consists in identifying relationships among products that significantly occur in customers buys. For instance, the rule "a customer who buys bean, in general buys rice", represented by $\{bean\} \Rightarrow \{rice\}$, would certainly be extracted from a market database in Brazil. Formally, an association rule, defined over a set of items $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$, is an implication of the form $X \Rightarrow Y$, where $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$, $X \neq \emptyset$, $Y \neq \emptyset$, and $X \cap Y = \emptyset$. We say that X is the antecedent and Y is the consequent of the rule.

Let $\mathcal{D}$ be a set of transactions (a transactional database) defined over $\mathcal{I}$, where each transaction t is a subset of $\mathcal{I}$ ($t \subseteq \mathcal{I}$). Then, the rule $X \Rightarrow Y$ holds in

* Work sponsored by CAPES Master scholarship.
** Work sponsored by CAPES Master scholarship.
*** Work sponsored by CNPq research grant 300879/00-8.

$\mathcal{D}$ with support s and confidence c if, respectively, $s\%$ of the transactions in $\mathcal{D}$ contain $X \cup Y$, and $c\%$ of the transactions in $\mathcal{D}$ that contain X also contain Y.

The ARM problem is commonly broken into two phases. Let $minsup$ and $minconf$ be, respectively, the user specified minimum support and confidence. The first phase, the *frequent itemset mining* (FIM) phase, consists in identifying all *frequent itemsets* (sets of items) that occur in at least $minsup\%$ of the transactions. The second phase outputs, for each identified frequent itemset Z, all association rules $A \Rightarrow B$ with confidence greater than or equal to $minconf$, such that $A \subset Z$, $B \subset Z$, and $A \cup B = Z$. The FIM phase demands more computational effort than the second one and has been intensively addressed [3].

1.1 Previous Work

During the last ten years, many algorithms have been proposed to efficiently mine frequent itemsets. Most of them are improved variations of Apriori [1]. In this strategy, the set F_1 containing all frequent itemsets of length 1 (1-itemsets) is initially identified. Then, at each iteration $k \geq 2$: (a) the set C_k of candidates of length k (k-candidates) is generated by combining all pairs included in F_{k-1} (frequent $(k-1)$-itemsets) that share a common $(k-2)$-prefix; (b) C_k is pruned in order to eliminate candidates that have at least one $(k-1)$-subset that is not frequent (all subsets of a frequent itemset are also frequent); (c) then, in the *counting phase*, the database $\mathcal{D}$ is read and, for each transaction t of $\mathcal{D}$, the support of all k-candidates contained in t is incremented. After reading the whole database, F_k is identified. If this set is empty, the termination condition is reached. In the Apriori algorithm, the candidates are stored in a hash-tree.

Subsequent proposed algorithms have improved Apriori by reducing: the number of database scans [11, 12], the computational cost of the counting phase [2, 7–9], and the number and size of transactions to be scanned [7–9].

Another class of FIM algorithms mines frequent itemsets by adopting a depth-first approach. The Eclat algorithm [14] uses an in-memory vertical layout of the database and an intersection-based approach to determine the supports of the candidates. The FP-growth algorithm [5] does not generate candidates. It builds a compact in-memory representation of the database, called FP-tree (*frequent pattern tree*), from which the support of all frequent itemsets are derived. According to experiments presented in [5], FP-growth is very efficient. However, it did not show good performance on sparse databases [15].

OpportuneProject [6], PatriciaMine [10] and FPgrowth* [4] are recent FIM algorithms that have improved the ideas adopted by FP-growth. OpportuneProject is able to choose between two different data structures according to the database features. PatriciaMine uses only one data structure (*Patricia trie*) to represent both dense and sparse databases together with optimizations that reduce the cost of tree traversal and that of physical database projections. FPgrowth* introduces an array-based technique also aiming at reducing FP-trees traversals.

DCI (*Direct Count & Intersect*) [8] and its recent version kDCI++ [7] are apriori-like algorithms. During their first iterations, they exploit database pruning techniques, inspired in [9], and efficient data structures to store candidates.

When the vertical layout of the pruned database fits into main memory, the supports of the candidates are obtained by an intersection-based technique. Both of them can adapt their behavior according to the database features. Moreover, kDCI++ uses a counting inference strategy based on that presented in [2].

A recent and efficient proposed method called LCM-freq [13] mines all frequent itemsets from frequent closed itemsets (an itemset is a closed itemset if none of its supersets have the same support).

1.2 Motivation

As pointed out in [3], every new proposed FIM algorithm is many times evaluated by limited experimental tests. In order to fairly evaluate the behavior of these algorithms, the *IEEE/ICDM Workshop on Frequent Itemset Mining Implementation (FIMI'03)* has been recently organized. All the accepted FIM implementations can now be found in the *FIMI repository* (available at http://fimi.cs.helsinki.fi/) together with an extensive performance evaluation.

According to the *FIMI'03 Workshop* experiments, kDCI++ is a state-of-the-art algorithm. However, it can be observed from its analysis [3] that its highly efficient behavior is not true for low values of support, specially on sparse databases.

Aiming at evaluating kDCI++ under these specific circumstances, we performed experiments on different combinations of databases and minimum supports also used in [2–8, 10, 13, 15]. We observed that the third iteration presented a very high computational cost compared to the other iterations.

Table 1 presents, for different databases and low minimum supports, the total execution times (in seconds) of kDCI++ and its third iteration execution times (in seconds). The databases are described in Section 4. The last column represents the ratio between the third iteration execution time and the total execution time. We observed that, on average, the third iteration represented 65% (ranged from 34% to 83%) of the total execution time of kDCI++ runs. This is due to the huge number of 3-candidates that must be evaluated. Indeed, as already observed in [7, 8], the third iteration may represent a bottleneck in apriori-like algorithms.

In this work, in order to improve the kDCI++ algorithm, we propose a strategy, called kDCI-3, that enables the direct counting at the third iteration, one of its most time consuming iterations for low minimum supports, specially on sparse databases. This proposal is based on a directly accessible data structure to store candidates, which allows a more efficient 3-candidate counting.

The paper is organized as follows. In Section 2, we review the kDCI++ algorithm. In Section 3, we present the kDCI-3 algorithm. The experimental results are reported and discussed in Section 4. Finally, in Section 5, some concluding remarks are made and future work is pointed out.

Table 1. Execution times of kDCI++

Database (*minsup* - %)	Total time	3^{rd} iteration time	(%)
T20I10N1KP5C0.25D200K (0.1)	544	353	65
T20I10N1KP5C0.25D200K (0.3)	51	34	67
T10I5N1KP5C0.25D200K (0.01)	298	220	74
T10I5N1KP5C0.25D200K (0.03)	104	70	67
T30I15N1KP5C0.25D200K (0.25)	655	541	83
T30I15N1KP5C0.25D200K (0.5)	139	116	83
T25I10D10K (0.1)	26	21	81
T25I10D10K (0.2)	4	1.9	47
T40I10D100K (0.5)	386	271	70
T40I10D100K (0.75)	133	101	76
T10I4D100K (0.01)	307	103	34
T10I4D100K (0.03)	36	28	78
T30I16D400K (0.4)	775	487	63
T30I16D400K (0.6)	234	161	69
BMS-POS (0.1)	422	156	37
BMS-POS (0.3)	47	22	47

2 The kDCI++ Algorithm

The kDCI++ algorithm [7] is an apriori-like strategy. During its first iterations, it exploits directly accessible data structures together with database pruning techniques that reduce the number and size of transactions to be scanned.

For $k = 2$, kDCI++ builds a *prefix table* P_2 of $\binom{|F_1|}{2}$ entries. Each entry of P_2 is a counter that represents a 2-candidate and accumulates its support.

To directly access the entry (counter) associated with a generic candidate c=(c_1,c_2), where $c_1 < c_2$, kDCI++ maps c into a pair $\{x_1,x_2\}$ where x_1=$\mathcal{T}(c_1)$, x_2=$\mathcal{T}(c_2)$, and $\mathcal{T}$ is a strictly monotonous increasing function defined by $\mathcal{T}$: $F_1 \rightarrow \{1,\ldots,|F_1|\}$. Equation 1 is thus adopted by kDCI++ in order to find the entry of P_2 that represents c=$\{c_1, c_2\}$, called here $EP_2(c_1, c_2)$. This equation is derived considering that the counters associated with pairs $\{1,x_2\}$, $2 \leq x_2 \leq |F_1|$ are stored in the first ($|F_1|$-1) positions of P_2, the counters associated with pairs $\{2,x_2\}$, $3 \leq x_2 \leq |F_1|$, are stored in the next ($|F_1|$-2) positions, and so on.

$$EP_2(c_1, c_2) = \sum_{i=1}^{x_1-1} (|F_1| - i) + (x_2 - x_1) = |F_1|(x_1 - 1) - \frac{x_1(x_1-1)}{2} + x_2 - x_1 \ . \quad (1)$$

At the end of the counting phase, if the corresponding entry of c is greater than or equal to *minsup*, c is included in F_2.

When $k > 2$, another data structure was proposed. A *prefix table* P_k of $\binom{|M_k|}{2}$ entries is built, where M_k is the set of items at iteration k that were not pruned during execution progress. Each entry of P_k contains a pointer to the

beginning of a memory section that stores the ordered k-candidates having the same 2-prefix. The entry of P_k that represents the prefix $\{c_1, c_2\}$ of a candidate c=$\{c_1, c_2, \ldots, c_k\}$ is obtained similarly to the second iteration.

To obtain the support of each k-candidate, for each transaction t, kDCI++ determines all possible 2-prefixes of all k-itemsets in t. Then, for each 2-prefix, the entry i of P_k is obtained and the section of ordered candidates that must be evaluated will be delimited by $P_k[i]$ and $P_k[i+1]$.

At each iteration $k \geq 2$, kDCI++ checks whether the vertical layout of the pruned database fits into main memory. The vertical layout can be seen as a set of $|M_k|$ bit vectors of size $|T_k|$, where T_k is the set of not pruned transactions at that iteration. If the bit vector associated with an item i has its j^{th} bit equal to 1, item i is present in the j^{th} transaction. If the database is small enough, its vertical representation is built and the supports of the $(k+1)$-candidates is thus obtained by the intersections of the k bit vectors associated with their items.

3 The kDCI-3 Algorithm

Aiming at improving the performance of kDCI++, in this section, we propose the kDCI-3 algorithm, which uses an efficient direct counting technique to determine the support of all 3-candidates.

The data structure used by kDCI-3 during the third iteration is based on the *prefix table* P_2 and on a new *array* C which represents all 3-candidates.

In the kDCI-3 algorithm, an entry of P_2 related to the frequent 2-itemset $\{c_1,c_2\}$ stores two values. The first one, represented by $Pt(c_1, c_2)$, is a pointer to the first entry of the contiguous section in C associated with all 3-candidates (in lexicographic order) having the same 2-prefix $\{c_1,c_2\}$. This value is null if there is no 3-candidate with that 2-prefix. The second value is the order, represented by $Ord(c_1, c_2)$, in which the frequent 2-itemset $\{c_1,c_2\}$ is represented in P_2. $Ord(c_1, c_2)$ is null if $\{c_1,c_2\}$ is not a frequent 2-itemset. To access P_2, kDCI-3 utilizes Equation 1, similarly to kDCI++.

The entry of C which corresponds to a generic 3-candidate c=$\{c_1,c_2,c_3\}$, called here $EC(c_1, c_2, c_3)$ and defined by Equation 2, can be found from the pointer $Pt(c_1, c_2)$ together with an offset value. This offset is defined by the number of frequent 2-itemsets represented between $\{c_1,c_2\}$ and $\{c_1,c_3\}$ in P_2.

$$EC(c_1, c_2, c_3) = Pt(c_1, c_2) + ((Ord(c_1, c_3) - Ord(c_1, c_2)) - 1)\ . \qquad (2)$$

Figure 1 illustrates how kDCI-3 identifies the entry associated with a given 3-candidate. In this example, F_1=$\{0,1,3,7,9\}$ and F_2=$\{\{0,1\},\{0,3\},\{0,7\},\{0,9\},\{1,3\},\{1,9\},\{3,9\}\}$. The set of 3-candidates (C_3), represented by C, is generated in lexicographic order by the combination of the frequent 2-itemsets that share a common 1-prefix. For instance, the frequent 2-itemset $\{0,1\}$ will be combined with all subsequent frequent 2-itemsets in P_2 that have 0 as their first item, in order to generate all 3-candidates with $\{0,1\}$ as their 2-prefix. Since the frequent 2-itemset $\{0,9\}$ is the third one after $\{0,1\}$, it is easy to conclude that the candidate $\{0,1,9\}$ is the third candidate in C having $\{0,1\}$ as their 2-prefix. Then, the

3-candidate {0,1,9}, generated by {0,1} and {0,9}, is located in C two entries after the first 3-candidate sharing the prefix {0,1}. Indeed, this two entries are given by $((Ord(0,9) - Ord(0,1)) - 1)$ and the first 3-candidate sharing the prefix {0,1} is identified by $Pt(0,1)$.

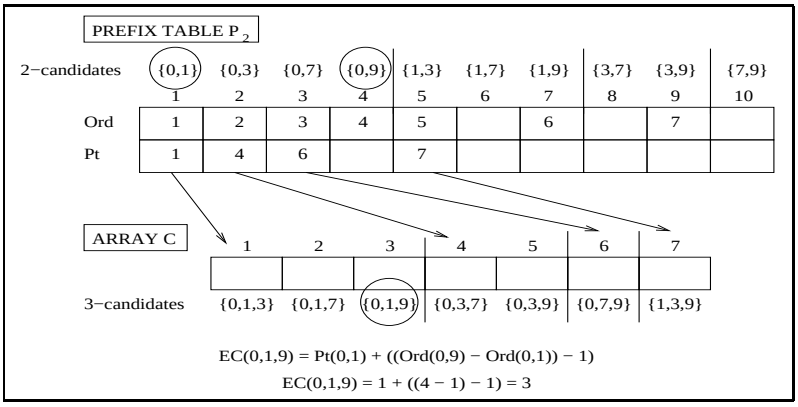

Fig. 1. Data structure used by kDCI-3 to access 3-candidates

To obtain the support of each 3-candidate, for each transaction t, kDCI-3 determines all the possible 3-itemsets included in t. After that, for each 3-itemset $\{c_1,c_2,c_3\}$ in t, it identifies the entries of P_2 that represents the 2-itemsets $\{c_1,c_2\}$ and $\{c_1,c_3\}$. If the identified entries represent frequent 2-itemsets, the entry associated with $\{c_1,c_2,c_3\}$, in C, is found from Equation 2 and is then incremented. At the end of the candidate counting, if an entry in C is greater than or equal to *minsup*, its corresponding candidate is included in F_3.

4 Performance Evaluation

The computational experiments reported in this work have been carried out on a 600 MHz Pentium III PC with 256 MB of RAM memory under the RedHat Linux 9.0 (kernel version 2.4.20) operating system.

The databases used in the experiments are described in Table 2. The first three databases were generated by the IBM dataset generator [1]. The databases T40I10100K, T10I4D100K and BMS-POS (real database) were downloaded from the *FIMI repository*, and finally, T25I10D10K and T30I16D400K were downloaded from the DCI site (*http://miles.cnuce.cnr.it/~palmeri/datam/DCI*).

This section is organized as follows. In Subsection 4.1, we compare the effectiveness of the new technique used by kDCI-3 with that adopted by kDCI++ during the third iteration. In Subsection 4.2, we compare the performance of kDCI-3 with recent *FIM* algorithms evaluated in the *FIMI'03 Workshop*.

Table 2. Databases used in the experiments and its corresponding characteristics

Database	Items	Transactions	Avg. Length	References
T20I10N1KP5C0.25D200K	1000	197,437	20.1	[3]
T10I5N1KP5C0.25D200K	1000	192,889	10.3	[3]
T30I15N1KP5C0.25D200K	1000	199,093	29.6	[3]
T25I10D10K	1001	9,219	27.7	[2, 5, 7, 8]
T40I10D100K	1000	100,000	39.6	[4, 10, 13]
T10I4D100K	1000	100,000	10.1	[10, 13]
T30I16D400K	1000	397,487	29.7	[7, 8, 10]
BMS-POS	1657	515,597	6.5	[3, 6, 10, 13, 15]

4.1 Effectiveness of the New Direct Counting Technique

For kDCI++, we used the source code available at the DCI site, now also available at the *FIMI repository*. For kDCI-3, we adapted the source code of kDCI++ in order to implement kDCI-3 features and to allow a fair comparison between them.

Table 3 shows, for the same combinations of databases and low minimum supports presented in Table 1, the total execution times (in seconds) and the third iteration execution times (in seconds) of kDCI++ and kDCI-3, respectively. The fourth column represents the ratio between the third iteration execution time of kDCI-3 and that of kDCI++. The seventh column represents the ratio between the total execution time of kDCI-3 and that of kDCI++.

Table 3. Execution times of kDCI++ and kDCI-3

	3^{rd} iteration times			Total times		
Database (*minsup* - %)	kDCI++	kDCI-3	(%)	kDCI++	kDCI-3	(%)
T20I10N1KP5C0.25D200K (0.1)	353	40	11	544	191	35
T20I10N1KP5C0.25D200K (0.3)	34	10	29	51	27	53
T10I5N1KP5C0.25D200K (0.01)	220	12	5	298	90	30
T10I5N1KP5C0.25D200K (0.03)	70	7	10	104	48	46
T30I15N1KP5C0.25D200K (0.25)	541	86	16	655	209	32
T30I15N1KP5C0.25D200K (0.5)	116	31	27	139	60	43
T25I10D10K (0.1)	21	4	19	26	9	35
T25I10D10K (0.2)	1.9	1.3	68	4	3.7	93
T40I10D100K (0.5)	271	69	25	386	208	54
T40I10D100K (0.75)	101	39	39	133	72	54
T10I4D100K (0.01)	103	6	6	307	202	66
T10I4D100K (0.03)	28	3	11	36	16	44
T30I16D400K (0.4)	487	132	27	755	478	63
T30I16D400K (0.6)	161	67	42	234	154	66
BMS-POS (0.1)	156	9	6	422	216	51
BMS-POS (0.3)	22	5	23	47	25	53

Due to the large number of 3-candidates generated in these tests (up to 11,051,970 on T10I4D100K (0.01%)), kDCI++ presented a poor performance in the third iteration. We can note that kDCI-3 efficiently reduced the execution time of this time consuming phase. We can observe, for instance, that in the first line of Table 3, while kDCI++ executed the third iteration in 353 seconds, kDCI-3 executed it in 40 seconds. This represents a reduction of 89%. Since the third iteration represents in this run a great part of the total execution time (as observed in Subsection 1.2), kDCI-3 total execution time was 35% of that of kDCI++ (a reduction of 65%).

The elapsed times of kDCI-3 were, for the third iteration, on average, 23% (from 5% to 68%) of that of kDCI++ (a reduction, on average, of 77%). We can also observe the impact produced by the proposed direct counting technique during the third iteration in the total execution times. The relative total execution times of kDCI-3 ranged from 30% to 66% (with a discrepant value of 93%) representing, on average, 51% of the total execution times of kDCI++.

4.2 Performance Comparison

This section reports a performance comparison among kDCI-3 and four preeminent algorithms: kDCI++, PatriciaMine, FPgrowth*, and LCM-freq. According to [3], kDCI++ and PatriciaMine are considered state-of-the-art algorithms. We also selected FPgrowth* and LCM-freq since they showed a good behavior in the *FIMI'03* experiments, specially for low supports. For PatriciaMine, FPgrowth* and LCM-freq, we used the source codes available at the *FIMI repository*.

Figure 2 shows the relative execution times of kDCI-3 and the other algorithms. In each picture, the value 1 (in y-axis) represents the relative execution time of the worst algorithm for the correspondent minimum support. The value in brackets beside the minimum support (in x-axis) is the absolute execution time of the worst algorithm (in seconds).

We observe that kDCI-3 was better than kDCI++ for almost all combinations of databases and minimum supports. In graph (e), for the highest support (0.9%), and in graph (f) for the three highest supports (0.5%, 0.4%, and 0.3%), kDCI++ was slightly better than kDCI-3 due to a not relevant number of 3-candidates.

For lower values of minimum support in graphs (e) and (f), when the number of 3-candidates increases, kDCI-3 performs much better than kDCI++. In some cases, as pointed out in [7], the bad behavior of kDCI++ can be justified by the size of C_3, which can lead to "a lot of useless work to determine the support of many candidate itemsets which are not frequent".

In graphs (g) and (h), kDCI++ was always the worst algorithm. For all values of support, kDCI-3 was better than kDCI++, making it more competitive.

In graphs (a)-(d), we observe that for higher minimum supports, kDCI-3 runs faster than kDCI++, which was the best algorithm in these situations. For lower minimum supports, however, kDCI++ performance decreases, but kDCI-3 is still better than kDCI++, also making it more competitive. In these graphs (and also in graph(e)), for some values of support, kDCI++ was the worst algorithm (or almost the worst), while the improvements of kDCI-3 made it the best one.

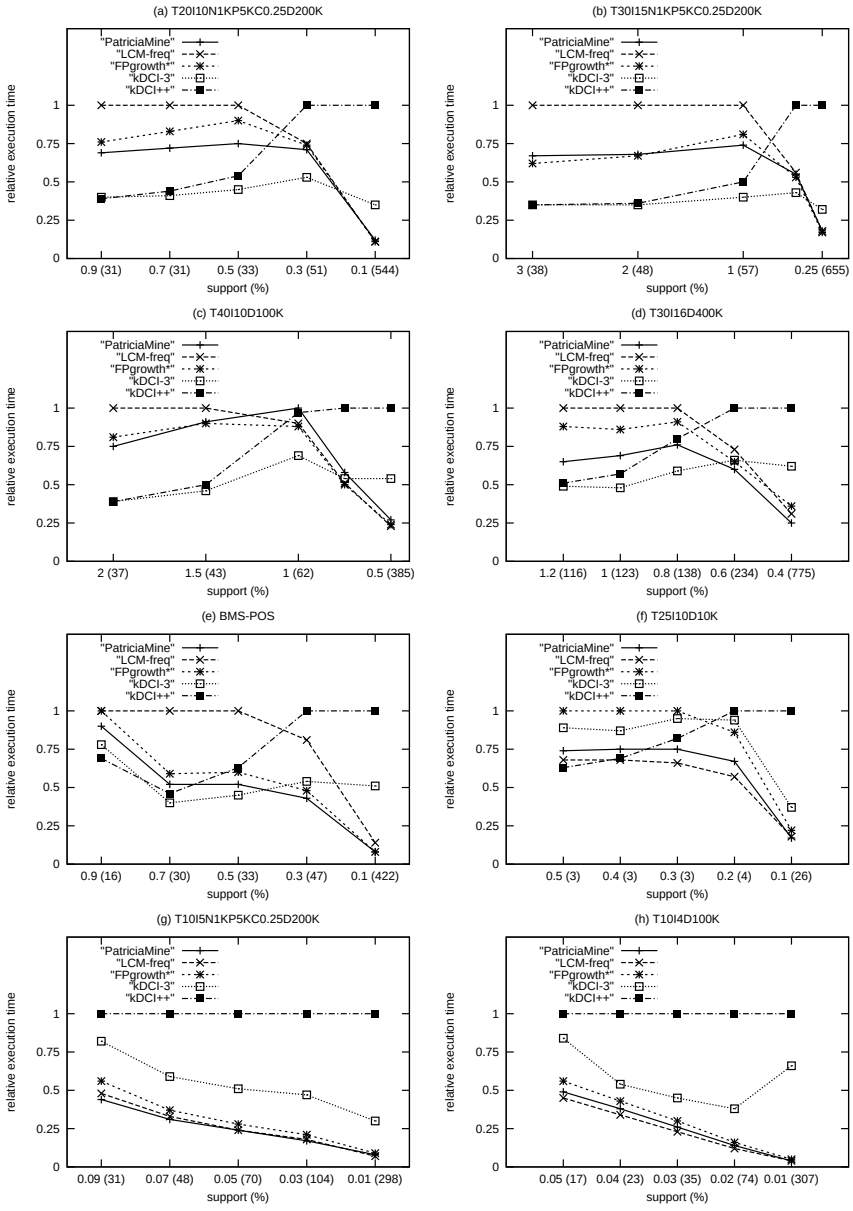

Fig. 2. Relative execution times of kDCI-3 and other algorithms

5 Conclusions

In this work we proposed kDCI-3, an extension of the kDCI++ algorithm – a recent and important method for the FIM problem. kDCI-3 provided an efficient direct counting of candidates when dealing with a huge number of 3-candidates, a consequence of low minimum support values, specially in sparse databases.

From the conducted computational experiments, we observed that kDCI-3 presented a very efficient behavior. kDCI-3 reduced the execution times of the third iteration of kDCI++ and, consequently, its total elapsed times in almost all executed experiments. We believe that these results represent an important contribution that improves a state-of-the-art algorithm.

Based on the encouraging observed results, as future work, we intend to investigate the use of the proposed direct counting of candidates in other iterations.

References

1. R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *20th VLDB Conference*, 1994.
2. Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L.Lakhal. Mining Frequent Patterns with Counting Inference. In *ACM SIGKDD Explorations*, v.2, n.2, 2000.
3. B. Goethals and M. J. Zaki. Advances in Frequent Itemset Mining Implementations: Introduction to FIMI'03. In *IEEE ICDM FIMI Workshop*, 2003.
4. G. Grahne, J. Zhu. Efficiently Using Prefix Trees in Mining Frequent Itemsets. In *IEEE ICDM FIMI Workshop*, 2003.
5. J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *ACM SIGMOD Conference*, 2000.
6. J. Liu, Y. Pan, K. Wang, and J. Han. Mining Frequent Item Sets by Opportunistic Projection. In *8th ACM SIGKDD Conference*, 2002.
7. S. Orlando, P. Palmerimi, R. Perego, C. Lucchese, and F. Silvestri. kDCI++: A Multi–Strategy Algorithm for Discovering Frequent Sets in Large Databases. In *IEEE ICDM FIMI Workshop*, 2003.
8. S. Orlando, P. Palmerimi, and R. Perego. Adaptive and Resource–Aware Mining of Frequent Sets. In *IEEE ICDM Conference*, 2002.
9. J. S. Park, M. Chen, and P. S. Yu. An Effective Hash-Based Algorithm for Mining Association Rules. In *ACM SIGMOD Conference*, 1995.
10. A. Pietracaprina and D. Zandolin. Mining Frequent Itemsets using Patricia Tries. In *IEEE ICDM FIMI Workshop*, 2003.
11. A. Savasere, E. Omiecinski, and S. Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. In *21th VLDB Conference*, 1995.
12. H. Toivonen. Sampling Large Databases for Association Rules. In *22th VLDB Conference*, 1996.
13. T. Uno, T. Asai, Y. Uchida, and H. Arimura. LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets. In *IEEE ICDM FIMI Workshop*, 2003.
14. M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. In *3rd ACM SIGKDD Conference*, 1997.
15. Z. Zheng, R. Kohavi, and L. Mason. Real World Performance of Association Rule Algorithms. In *7th ACM SIGKDD Conference*, 2001.