Probabilistic Iterative Expansion of Candidates in Mining Frequent Itemsets

Attila Gyenesei and Jukka Teuhola
Turku Centre for Computer Science, Dept. of Inf. Technology, Univ. of Turku, Finland
Email: {gyenesei,teuhola}@it.utu.fi

Abstract

A simple new algorithm is suggested for frequent itemset mining, using item probabilities as the basis for generating candidates. The method first finds all the frequent items, and then generates an estimate of the frequent sets, assuming item independence. The candidates are stored in a trie where each path from the root to a node represents one candidate itemset. The method expands the trie iteratively, until all frequent itemsets are found. Expansion is based on scanning through the data set in each iteration cycle, and extending the subtries based on observed node frequencies. Trie probing can be restricted to only those nodes which possibly need extension. The number of candidates is usually quite moderate; for dense datasets 2-4 times the number of final frequent itemsets, for non-dense sets somewhat more. In practical experiments the method has been observed to make clearly fewer passes than the well-known Apriori method. As for speed, our non-optimised implementation is in some cases faster, in some others slower than the comparison methods.

1. Introduction

We study the well-known problem of finding *frequent itemsets* from a transaction database, see [2]. A *transaction* in this case means a set of so-called *items*. For example, a supermarket basket is represented as a transaction, where the purchased products represent the items. The database may contain millions of such transactions. The frequent itemset mining is a task, where we should find those subsets of items that occur at least in a given minimum number of transactions. This is an important basic task, applicable in solving more advanced data mining problems, for example discovering *association rules* [2]. What makes the task difficult is that the number of potential frequent itemsets is exponential in the number of distinct items.

In this paper, we follow the notations of Goethals [7]. The overall set of items is denoted by I. Any subset $X \subseteq I$ is called an itemset. If X has k items, it is called a k-itemset. A transaction is an itemset identified by a *tid*. A transaction with itemset Y is said to *support* itemset X, if $X \subseteq Y$. The *cover* of an itemset X in a database D is the set of transactions in D that support X. The *support* of itemset X is the size of its cover in D. The *relative frequency* (probability) of itemset X with respect to D is

$$P(X, D) = \frac{Support(X, D)}{|D|} \qquad (1)$$

An itemset X is frequent if its support is greater than or equal to a given threshold σ. We can also express the condition using a relative threshold for the frequency: $P(X, D) \geq \sigma_{rel}$, where $0 \leq \sigma_{rel} \leq 1$. There are variants of the basic 'all-frequent-itemsets' problem, namely the *maximal* and *closed* itemset mining problems, see [1, 4, 5, 8, 12]. However, here we restrict ourselves to the basic task.

A large number of algorithms have been suggested for frequent itemset mining during the last decade; for surveys, see [7, 10, 15]. Most of the algorithms share the same general approach: generate a set of *candidate itemsets*, count their frequencies in D, and use the obtained information in generating more candidates, until the complete set is found. The methods differ mainly in the order and extent of candidate generation. The most famous is probably the *Apriori* algorithm, developed independently by Agrawal et al. [3] and Mannila et al. [11]. It is a representative of *breadth-first* candidate generation: it first finds all frequent 1-itemsets, then all frequent 2-itemsets, etc. The core of the method is clever pruning of candidate k-itemsets, for which there exists a non-frequent k-1-subset. This is an application of the obvious *monotonicity* property: All subsets of a frequent itemset must also be frequent. Apriori is essentially based on this property.

The other main candidate generation approach is *depth-first* order, of which the best-known representatives are Eclat [14] and FP-growth [9] (though the 'candidate' concept in the context of FP-growth is disputable). These two are generally considered to be among the fastest algorithms for frequent itemset mining. However, we shall mainly use Apriori as a reference method, because it is technically closer to ours.

Most of the suggested methods are *analytical* in the sense that they are based on logical inductions to restrict the number of candidates to be checked. Our approach (called *PIE*) is *probabilistic*, based on relative item frequencies, using which we compute estimates for itemset frequencies in candidate generation. More precisely, we generate iteratively improving approximations (candidate itemsets) to the solution. Our general endeavour has been to develop a relatively simple method, with fast basic steps and few iteration cycles, at the cost of somewhat increased number of candidates. However, another goal is that the method should be robust, i.e. it should work reasonably fast for all kinds of datasets.

2. Method description

Our method can be characterized as a generate-and-test algorithm, such as Apriori. However, our candidate generation is based on probabilistic estimates of the supports of itemsets. The testing phase is rather similar to Apriori, but involves special book-keeping to lay a basis for the next generation phase.

We start with a general description of the main steps of the algorithm. The first thing to do is to determine the frequencies of all items in the dataset, and select the frequent ones for subsequent processing. If there are m frequent items, we internally identify them by numbers 0, …, m-1. For each item i, we use its probability (relative frequency) $P(i)$ in the generation of candidates for frequent itemsets.

The candidates are represented as a *trie* structure, which is normal in this context, see [7]. Each node is labelled by one item, and a path of labels from the root to a node represents an itemset. The root itself represents the empty itemset. The paths are sorted, so that a subtrie rooted by item i can contain only items > i. Note also that several nodes in the trie can have the same item label, but not on a single path. A complete trie, storing all subsets of the whole itemset, would have 2^m nodes and be structurally a *binomial tree* [13], where on level j there are $\binom{m}{j}$ nodes, see Fig. 1 for $m = 4$.

The trie is used for book-keeping purposes. However, it is important to avoid building the complete trie, but only some upper part of it, so that the nodes (i.e. their root paths) represent reasonable candidates for frequent sets. In our algorithm, the first approximation for candidate itemsets is obtained by computing estimates for their probabilities, assuming independence of item occurrences. It means that, for example, for an itemset $\{x, y, z\}$ the estimated probability is the product $P(x)P(y)P(z)$. Nodes are created in the trie from root down along all paths as long as the path-related probability is not less that the threshold σ_{rel}. Note that the probability values are monotonically non-increasing on the way down. Fig. 2

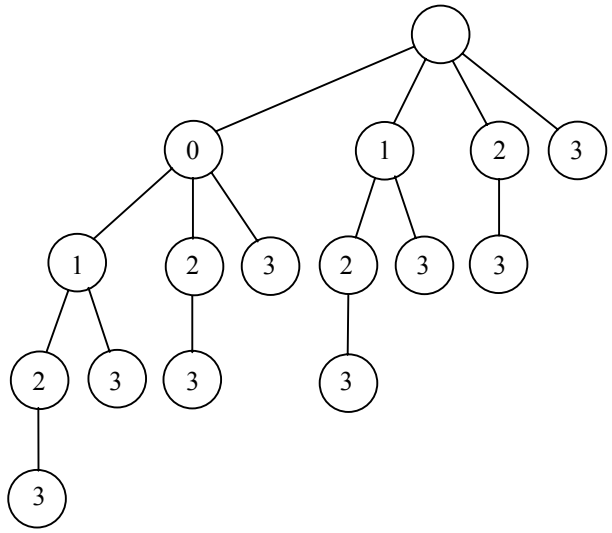

Figure 1. The complete trie for 4 items.

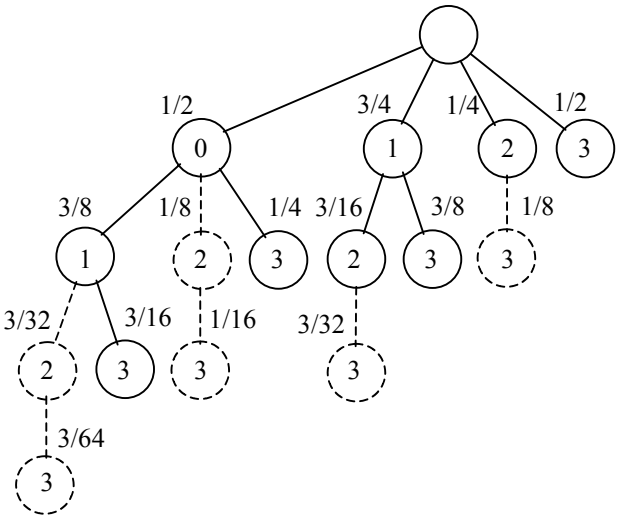

Figure 2. An initial trie for the transaction set {(0, 3), (1, 2), (0, 1, 3), (1)}, with minimum support threshold σ = 1/6. The virtual nodes with probabilities < 1/6 are shown using dashed lines.

shows an example of the initial trie for a given set of transactions (with m = 4). Those nodes of the complete trie (Fig. 1) that do not exist in the actual trie are called *virtual nodes*, and marked with dashed circles in Fig. 2.

The next step is to read the transactions and count the true number of occurrences for each node (i.e. the related path support) in the trie. Simultaneously, for each visited node, we maintain a counter called *pending support* (*PS*), being the number of transactions for which at least one

virtual child of the node would match. The pending support will be our criterion for the *expansion* of the node: If $PS(x) \geq \sigma$, then it is possible that a virtual child of node x is frequent, and the node must be expanded. If there are no such nodes, the algorithm is ready, and the result can be read from the trie: All nodes with support $\geq \sigma$ represent frequent itemsets.

Trie expansion starts the next cycle, and we iterate until the stopping condition holds. However, we must be very careful in the expansion: which virtual nodes should we materialize (and how deep, recursively), in order to avoid trie 'explosion', but yet approach the final solution? Here we apply item probabilities, again. In principle, we could take advantage of all information available in the current trie (frequencies of subsets, etc.), as is done in the Apriori algorithm and many others. However, we prefer simpler calculation, based on global probabilities of items.

Suppose that we have a node x with pending support $PS(x) \geq \sigma$. Assume that it has virtual child items $v_0, v_1, \ldots, v_{s-1}$ with global probabilities $P(v_0), P(v_1), \ldots, P(v_{s-1})$. Every transaction contributing to $PS(x)$ has a match with at least one of $v_0, v_1, \ldots, v_{s-1}$. The *local probability* (*LP*) for a match with v_i is computed as follows:

$$\begin{aligned} &LP(v_i) \\ &= P(v_i \text{ matches} \mid \text{One of } v_0, v_1, \ldots \text{ matches}) \\ &= \frac{P((v_i \text{ matches}) \wedge (\text{One of } v_0, v_1 \ldots \text{ matches}))}{P(\text{One of } v_0, v_1, \ldots \text{ matches})} \\ &= \frac{P(v_i \text{ matches})}{P(\text{One of } v_0, v_1, \ldots \text{ matches})} \\ &= \frac{P(v_i)}{1-(1-P(v_0))(1-P(v_1))\ldots(1-P(v_s))} \end{aligned} \quad (2)$$

Using this formula, we get an *estimated support* $ES(v_i)$:

$$ES(v_i) = LP(v_i) PS(Parent(V_i)) \quad (3)$$

If $ES(v_i) \geq \sigma$, then we conclude that v_i is expected to be frequent. However, in order to guarantee a finite number of iterations in the worst case, we have to relax this condition a bit. Since the true distribution may be very skewed, almost the whole pending support may belong to only one virtual child. To ensure convergence, we apply the following condition for child expansion in the k^{th} iteration,

$$ES(v_i) \geq \alpha^k \sigma \quad (4)$$

with some constant α between 0 and 1. In the worst case this will eventually (when k is high enough) result in expansion, to get rid of a *PS*-value $\geq \sigma$. In our tests, we used the heuristic value α = average probability of frequent items. The reasoning behind this choice is that it speeds up the local expansion growth by one level, on the average (k levels for α^k). This acceleration restricts the number of iterations efficiently. The largest extensions are applied only to the 'skewest' subtries, so that the total size of the trie remains tolerable. Another approach to choose α would be to do a statistical analysis to determine confidence bounds for *ES*. However, this is left for future work.

Fig. 3 shows an example of trie expansion, assuming that the minimum support threshold $\sigma = 80$, $\alpha = 0.8$, and $k = 1$. The item probabilities are assumed to be P(y) = 0.7, P(z) = 0.5, and P(v) = 0.8. Node t has a pending support of 100, related to its two virtual children, y and z. This means that 100 transactions contained the path from root to t, plus either *or both* of items y and z, so we have to test for expansion. Our formula gives y a local probability $LP(y) = 0.7 / (1-(1-0.7)(1-0.5)) \approx 0.82$, so the estimated support is $82 > \alpha \cdot \sigma = 64$, and we expand y. However, the local probability of z is only ≈ 0.59, so its estimated support is 59, and it will not be expanded.

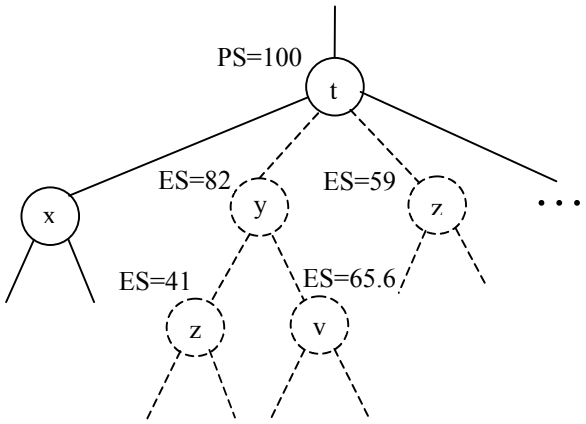

Figure 3. An example of expansion for probabilities P(y) = 0.7, P(z) = 0.5, and P(v) = 0.8.

When a virtual node (y) has been materialized, we immediately test also its expansion, based on its *ES*-value, recursively. However, in the recursive steps we cannot apply formula (2), because we have no evidence of the children of y. Instead, we apply the unconditional probabilities of z and v in estimation: $LP(z) = 82 \cdot 0.5 = 41 < \alpha \cdot \sigma = 64$, and $LP(v) = 82 \cdot 0.8 = 65.6 > 64$. Node v is materialized, but z is not. Expansion test continues down from v. Thus, both in initialization of the trie and in its expansion phases, we can create several new levels (i.e. longer candidates) at a time, contrary to e.g. the base version of Apriori. It is true that also Apriori can be modified to create several candidate levels at a time, but at the cost of increased number of candidates.

After the expansion phase the iteration continues with the counting phase, and new values for node supports and pending supports are determined. The two phases alternate

until all pending supports are less than σ. We have given our method the name 'PIE', reflecting this Probabilistic Iterative Expansion property.

3. Elaboration

The above described basic version does a lot of extra work. One observation is that as soon as the pending support of some node x is smaller than σ, we can often 'freeze' the whole subtrie, because it will not give us anything new; we call it 'ready'. The readiness of nodes can be checked easily with a recursive process: A node x is ready if $PS(x) < \sigma$ and all its real children are ready. The readiness can be utilized to reduce work both in counting and expansion phases. In counting, we process one transaction at a time and scan its item subsets down the trie, but only until the first ready node on each path. Also the expansion procedure is skipped for ready nodes. Finally, a simple stopping condition is when the root becomes ready.

Another tailoring, not yet implemented, relates to the observation that most of the frequent itemsets are found in the first few iterations, and a lot of I/O effort is spent to find the last few frequent sets. For those, not all transactions are needed in solving the frequency. In the counting phase, we can distinguish between relevant and irrelevant transactions. A transaction is irrelevant, if it does not increase the pending support value of any non-ready node. If the number of relevant transactions is small enough, we can store them separately (in main memory or temporary file) during the next scanning phase.

Our implementation of the trie is quite simple; saving memory is considered, but not as the first preference. The child linkage is implemented as an array of pointers, and the frequent items are renumbered to 0, …, m-1 (if there are m frequent items) to be able to use them as indices to the array. A minor improvement is that for item i, we need only m-i-1 pointers, corresponding to the possible children i+1, …, m-1.

The main restriction of the current implementation is the assumption that the trie fits in the main memory. Compression of nodes would help to some extent: Now we reserve a pointer for every possible child node, but most of them are null. Storing only non-null pointers saves memory, but makes the trie scanning slower. Also, we could release the ready nodes as soon as they are detected, in order to make room for expansions. Of course, before releasing, the related frequent itemsets should be reported. However, a fully general solution should work for any main memory and trie size. Some kind of external representation should be developed, but this is left for future work.

A high-level pseudocode of the current implementation is given in the following. The recursive parts are not coded explicitly, but should be rather obvious.

Algorithm PIE – Probabilistic iterative expansion of candidates in frequent itemset mining

---

Input: A transaction database D, the minimum support threshold σ.
Output: The complete set of frequent itemsets.

---

1. // Initial steps.
2. **scan** *D* and collect the set *F* of frequent items;
3. α := average probability of items in *F*;
4. *iter* := 0;

5. // The first generation of candidates, based on
   // item probabilities.
6. **create** a PIE-trie *P* so that it contains all such ordered subsets $S \subseteq F$ for which $\Pi(\text{Prob}(s \in S)) \cdot |D| \geq \sigma$; // Frequency test
7. **set** the status of all nodes of *P* to not-ready;

8. // The main loop: alternating count, test and
   // expand.
9. **loop**
10. // Scan the database and check readiness.
11. **scan** *D* and count the support and pending support values for non-ready nodes in *P*;
12. *iter* := *iter* + 1;
13. **for each** node $p \in P$ **do**
14. **if** *pending_support*(*p*) < σ **then**
15. **if** *p* is a leaf **then** set *p* ready
16. **else if** the children of *p* are ready **then**
17. **set** *p* ready;
18. **if** *root*(*P*) is ready **then exit loop**;

19. // Expansion phase: Creation of subtries on
    // the basis of observed pending supports.
20. **for each** non-ready node *p* **in** *P* **do**
21. **if** *pending_support*(*p*) ≥ σ **then**
22. **for each** virtual child *v* of *p* **do**
23. **compute** *local_prob*(*v*) by formula (2);
24. *estim_support*(*v*) := *local_prob*(*v*) · *pending_support*(*p*);
25. **if** *estim_support*(*v*) $\geq \alpha^{iter}\sigma$ **then**
26. **create** node *v* as the child of *p*;
27. **add** such ordered subsets $S \subseteq F \backslash \{1..v\}$ as descendant paths of *v*, for which $\Pi(\text{Prob}(s \in S)) \cdot \textit{estim\_support}(v) \geq \alpha^{iter}\sigma$;

28. // Gather up results from the trie
29. **return** the paths for nodes *p* in *P* such that *support*(*p*) ≥ σ;
30. **end**

---

4. Experimental results

For verifying the usability of our PIE algorithm, we used four of the test datasets made available to the Workshop on Frequent Itemset Mining Implementations (FIMI'03) [6]. The test datasets and some of their properties are described in Table 1. They represent rather different kinds of domains, and we wanted to include both dense and non-dense datasets, as well as various numbers of items.

Table 1. Test dataset description

Dataset	#Transactions	#Items
Chess	3 196	75
Mushroom	8 124	119
T40I10D100K	100 000	942
Kosarak	900 002	41 270

For the PIE method, the interesting statistics to be collected are the number of candidates, depth of the trie, and the number of iterations. These results are given in Table 2 for selected values of σ, for the 'Chess' dataset. We chose values of σ that keep the number of frequent itemsets reasonable (extremely high numbers are probably useless for any application). The table shows also the number of frequent items and frequent sets, to enable comparison with the number of candidates. For this dense dataset, the number of candidates varies between 2-4 times the number of frequent itemsets. For non-dense datasets the ratio is usually larger. Table 2 shows also the values of the 'security parameter' α, being the average probability of frequent items. Considering I/O performance, we can see that the number of iteration cycles (= number of file scans) is quite small, compared to the base version of the Apriori method, for which the largest frequent itemset dictates the number of iterations. This is roughly the same as the trie depth, as shown in Table 2.

The PIE method can also be characterized by describing the development of the trie during the iterations. The most interesting figures are the number of nodes and the number of ready nodes, given in Table 3. Especially the number of ready nodes implies that even though we have rather many candidates (= nodes in the trie), large parts of them are not touched in the later iterations.

Table 3. Development of the trie for dataset 'Chess', with three different values of σ.

σ	Iteration	#Frequent sets found	#Nodes	#Ready nodes
2600	1	4 720	4 766	2 021
	2	6 036	9 583	9 255
	3	6 134	10 296	10 173
	4	6 135	10 516	10 516
2400	1	15 601	15 760	5 219
	2	20 344	34 995	25 631
	3	20 580	47 203	46 952
	4	20 582	47 515	47 515
2200	1	44 022	44 800	1 210
	2	58 319	112 370	64 174
	3	59 176	206 292	196 782
	4	59 181	216 931	216 922
	5	59 181	216 943	216 943

For speed comparison, we chose the Apriori and FP-growth implementations, provided by Bart Goethals [6]. The results for the four test datasets and for different minimum support thresholds are shown in Table 4. The processor used in the experiments was a 1.5 GHz Pentium 4, with 512 MB main memory. We used a g++ compiler, using optimizing switch –O6. The PIE algorithm was coded in C.

Table 2. Statistics from the PIE algorithm for dataset 'Chess'.

σ	#Frequent items	#Frequent sets	Alpha	#Candidates	Trie depth	#Iterations	#Apriori's iterations
3 000	12	155	0.970	400	6	3	6
2 900	13	473	0.967	1 042	8	4	7
2 800	16	1 350	0.953	2 495	8	4	8
2 700	17	3 134	0.947	5 218	9	4	8
2 600	19	6 135	0.934	10 516	10	4	9
2 500	22	11 493	0.914	18 709	11	4	10
2 400	23	20 582	0.907	47 515	12	4	11
2 300	24	35 266	0.900	131 108	13	4	12
2 200	27	59 181	0.877	216 943	14	5	13

Table 4. Comparison of execution times (in seconds) of three frequent itemset mining programs for four test datasets.

(a) Chess

σ	#Freq. sets	Apriori	FP-growth	PIE
3 000	155	0.312	0.250	0.125
2 900	473	0.469	0.266	0.265
2 800	1 350	0.797	0.297	1.813
2 700	3 134	1.438	0.344	6.938
2 600	6 135	3.016	0.438	14.876
2 500	11 493	10.204	0.610	26.360
2 400	20 582	21.907	0.829	78.325
2 300	35 266	42.048	1.156	203.828
2 200	59 181	73.297	1.766	315.562

(b) Mushroom

σ	#Freq. sets	Apriori	FP-growth	PIE
5 000	41	0.375	0.391	0.062
4 500	97	0.437	0.406	0.094
4 000	167	0.578	0.438	0.141
3 500	369	0.797	0.500	0.297
3 000	931	1.062	0.546	1.157
2 500	2 365	1.781	0.610	6.046
2 000	6 613	3.719	0.750	27.047
1 500	56 693	55.110	1.124	153.187

(c) T40I10D100K

σ	#Freq. sets	Apriori	FP-growth	PIE
20 000	5	2.797	6.328	0.797
18 000	9	2.828	6.578	1.110
16 000	17	3.001	7.250	1.156
14 000	24	3.141	8.484	1.187
12 000	48	3.578	14.750	1.906
10 000	82	4.296	23.874	4.344
8 000	137	7.859	41.203	11.796
6 000	239	20.531	72.985	29.671
4 000	440	35.282	114.953	68.672

(c) Kosarak

σ	#Freq. sets	Apriori	FP-growth	PIE
20 000	121	27.970	30.141	5.203
18 000	141	28.438	31.296	6.110
16 000	167	29.016	32.765	7.969
14 000	202	29.061	33.516	9.688
12 000	267	29.766	34.875	12.032
10 000	376	34.906	37.657	18.016
8 000	575	35.891	41.657	30.453
6 000	1 110	39.656	51.922	70.376

We can see that in some situations the PIE algorithm is the fastest, in some others the slowest. This is probably a general observation: the performance of most frequent itemset mining algorithms is highly dependent on the data set and threshold. It seems that PIE is at its best for sparse datasets (such as T40I10D100K and Kosarak), but not so good for very dense datasets (such as 'Chess' and 'Mushroom'). Its speed for large thresholds probably results from the simplicity of the algorithm. For smaller thresholds, the trie gets large and the counting starts to consume more time, especially with a small main memory size.

One might guess that our method is at its best for *random* data sets, because those would correspond to our assumption about independent item occurrences. We tested this with a dataset of 100 000 transactions, each of which contained 20 random items out of 30 possible. The results were rather interesting: For all tested thresholds for minimum support, we found *all* the frequent itemsets in the first iteration. However, verification of the completeness required one or two additional iterations, with a clearly higher number of candidates, consuming a majority of the total time. Table 5 shows the time and number of candidates both after the first and after the final iteration. The stepwise growth of the values reveals the levelwise growth of the trie. Apriori worked well also for this dataset, being in most cases faster than PIE. Results for FP-growth (not shown) are naturally much slower, because randomness prevents a compact representation of the transactions.

We wish to point out that our implementation was an initial version, with no special tricks for speed-up. We are convinced that the code details can be improved to make the method still more competitive. For example, buffering of transactions (or temporary files) were not used to enhance the I/O performance.

5. Conclusions and future work

A probability-based approach was suggested for frequent itemset mining, as an alternative to the 'analytic' methods common today. It has been observed to be rather robust, working reasonably well for various kinds of datasets. The number of candidate itemsets does not 'explode', so that the data structure (trie) can be kept in the main memory in most practical cases.

The number of iterations is smallest for random datasets, because candidate generation is based on just that assumption. For skewed datasets, the number of iterations may somewhat grow. This is partly due to our simplifying premise that the items are independent. This point could be tackled by making use of the conditional probabilities obtainable from the trie. Initial tests did not show any significant advantage over the basic approach, but a more

Table 5. Statistics from the PIE algorithm for a random dataset.

σ	#Freq. sets	PIE						Apriori	
		After iteration 1.			After the last iteration (final)				
		#Freq. sets	Time (sec.)	#Cand.	#Cand.	#Iter-ations	Time (sec.)	#Iter-ations	Time (sec.)
50 000	30	30	0.500	30	464	2	2.234	2	3.953
44 000	42	42	2.016	465	509	3	2.704	3	5.173
43 800	124	124	1.875	465	1 247	3	10.579	3	6.015
43 700	214	214	1.876	465	1 792	3	20.250	3	7.235
43 600	331	331	1.891	465	2 775	3	37.375	3	9.657
43 500	413	413	1.860	465	3 530	3	48.953	3	11.876
40 000	465	465	1.844	465	4 443	2	62.000	3	13.875
28 400	522	522	60.265	4 525	4 900	3	64.235	4	15.016
28 300	724	724	61.422	4 525	5 989	3	82.140	4	15.531
28 200	1 270	1 270	61.469	4 525	8 697	3	115.250	4	19.265
28 100	2 223	2 223	61.734	4 525	13 608	3	167.047	4	31.266
28 000	3 357	3 357	60.969	4 525	19 909	3	219.578	4	69.797

sophisticated probabilistic analysis might imply some ways to restrict the number of candidates. The exploration of these elaborations, as well as tuning the buffering, data structure, and parameters, is left for future work.

References

[1] R. Agrawal, C. Aggarwal, and V.V.V. Prasad, "Depth First Generation of Long Patterns", In R. Ramakrishnan, S. Stolfo, R. Bayardo, and I. Parsa (eds.), *Proc. of the Int. Conf. on Knowledge Discovery and Data Mining*, ACM, Aug. 2000, pp. 108-118.

[2] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules Between Sets of Items in Large Databases", In P. Buneman and S. Jajodia (eds.), *Proc. of ACM SIGMOD Int. Conf. of Management of Data*, May 1993, pp. 207-216.

[3] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases", In J.B. Bocca, M. Jarke, and C. Zaniolo (eds.), *Proc. of the 20th VLDB Conf.*, Sept. 1994, pp. 487-499.

[4] R.J. Bayardo, "Efficiently Mining Long Patterns from Databases", In L.M. Haas and A. Tiwary (eds.), *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, June 1998, pp. 85-93.

[5] D. Burdick, M. Calimlim, and J. Gehrke, "MAFIA: a Maximal Frequent Itemset Algorithm for Transactional Databases", *Proc. of IEEE Int. Conf. on Data Engineering*, April 2001, pp. 443-552.

[6] Frequent Itemset Mining Implementations (FIMI'03) Workshop website, *http://fimi.cs.helsinki.fi*, 2003.

[7] B. Goethals, "Efficient Frequent Pattern Mining", *PhD thesis*, University of Limburg, Belgium, Dec. 2002.

[8] K. Gouda and M.J. Zaki, "Efficiently Mining Maximal Frequent Itemsets", In N. Cercone, T.Y. Lin, and X. Wu (eds.), *Proc. of 2001 IEEE International Conference on Data Mining*, Nov. 2001, pp. 163-170.

[9] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns Without Candidate Generation", In W. Chen, J. Naughton, and P.A. Bernstein (eds.), *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, 2000, pp. 1-12.

[10] J. Hipp, U. Güntzer, and N. Nakhaeizadeh, "Algorithms for Association Rule Mining - a General Survey and Comparison", *ACM SIGKDD Explorations* 2, July 2000, pp. 58-65.

[11] H. Mannila, H. Toivonen, and A.I. Verkamo, "Efficient Algorithms for Discovering Association Rules", In U.M. Fayyad and R. Uthurusamy (eds.), *Proc. of the AAAI Workshop on Knowledge Discovery in Databases*, July 1994, pp. 181-192.

[12] J. Pei, J. Han, and R. Mao, "Closet: An Efficient Algorithm for Mining Frequent Closed Itemsets", *Proc. of ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, May 2000*, pp. 21-30.

[13] J. Vuillemin, "A Data Structure for Manipulating Priority Queues", *Comm. of the ACM*, 21(4), 1978, pp. 309-314.

[14] M.J. Zaki, "Scalable Algorithms for Association Mining", *IEEE Transactions on Knowledge and Data Engineering* 12 (3), 2000, pp. 372-390.

[15] Z. Zheng, R. Kohavi, and L. Mason, "Real World Performance of Association Rule Algorithms", In F. Provost and R. Srikant (eds.), *Proc. of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2001, pp. 401-406.