Surprising results of trie-based FIM algorithms

Ferenc Bodon*
bodon@cs.bme.hu
Department of Computer Science and Information Theory,
Budapest University of Technology and Economics

Abstract

Trie is a popular data structure in frequent itemset mining (FIM) algorithms. It is memory-efficient, and allows fast construction and information retrieval. Many trie-related techniques can be applied in FIM algorithms to improve efficiency. In this paper we propose new techniques for fast management, but more importantly we scrutinize the well-known ones especially those which can be employed in APRIORI. The theoretical claims are supported by results of a comprehensive set of experiments, based on hundreds of tests that were performed on numerous databases, with different support thresholds. We offer some surprising conclusions, which at some point contradict published claims.

1. Introduction

Frequent itemset mining is the most researched field of frequent pattern mining. Techniques and algorithms developed here are often used in search for other types of patterns (like sequences, rooted trees, boolean formulas, graphs). The original problem was to discover association rules [2], where the main step was to find frequently occurring itemsets. Over one hundred FIM algorithms were proposed – the majority claiming to be the most efficient. The truth is that no single most efficient algorithm exists; there is no published method that outperforms every other method on every dataset with every support threshold [11]. However, there are three algorithms that play central role due to their efficiency and the fact that many algorithms are modifications or combinations of these basic methods. These algorithms are *APRIORI* [3], *Eclat* [25] and *FP-growth* [12].

Those who employed one of the basic algorithms as a search strategy, tended to employ the whole set of procedures and data structures as well, which is trie (prefix tree) in the case of APRIORI and FP-growth. Therefore it is useful and instructive to analyze tries, and clarify those details that have an effect on run-time or memory need. In this paper we will see, that small details can have a large influence on efficiency and taking a closer look at them brings up new questions and new solutions.

The rest of the paper is organized as follows. The problem is presented in Section 2, trie and its role in FIM is described in Section 3. Section 4 introduces accelerating techniques one-by-one. Surprising experimental results and the explanations are given in Section 5.

2. Problem statement

Frequent itemset mining is a special case of *frequent pattern mining*. Let us first describe this general case. We assume that the reader is familiar with the basics of poset theory. We call a poset $(P, \preceq)$ *locally finite*, if every interval $[x, y]$ is finite, i.e. the number of elements z, such that $x \preceq z \preceq y$ is finite. The element x *covers* y, if $y \preceq x$ and for any $y \preceq z$, $z \not\preceq x$.

Definition 1. *We call the poset $\mathcal{PC} = (\mathcal{P}, \preceq)$* pattern context, *if there exists exactly one minimal element, $\mathcal{PC}$ is locally finite and* graded, *i.e. there exists a* size *function $|\ |: \mathcal{P} \to \mathbb{Z}$, such that $|p| = |p'| + 1$, if p covers p'. The elements of $\mathcal{P}$ are called* patterns *and $\mathcal{P}$ is called the* pattern space[1] *or* pattern set.

Without loss of generality we assume that the size of the minimal pattern is 0 and it is called the *empty pattern*.

In the *frequent pattern mining problem* we are given the set of input data $\mathcal{T}$, the pattern context $\mathcal{PC} = (\mathcal{P}, \preceq)$, the anti-monotonic function $supp_{\mathcal{T}} : \mathcal{P} \to \mathbb{N}$ and $min\_supp \in \mathbb{N}$. We have to find the set $F = \{p \in \mathcal{P} : supp_{\mathcal{T}}(p) \geq$

*Research is partially supported by the Hungarian National Science Foundation (Grant No. OTKA TS-044733, T42706, T42481).

[1] Many researchers improperly refer to the pattern space as the pattern lattice. It may come from the fact that patterns were first examined as sets of items [2], when $(\mathcal{P}, \preceq)$ actually formed a lattice. However this property does not hold for many other types of patterns. It is easy to prove that if the type of pattern is sequence, boolean formula or graph, then the least upper bound is not unique.

$min\_supp\}$ and the support of the patterns in F. Elements of F are called *frequent* patterns, $supp_{\mathcal{T}}$ is the *support function* and min_supp is referred to as *support threshold*.

There are many types of patterns: itemsets [2], item sequences, sequences of itemsets [4], episodes [16], boolean formulas [15], rooted labeled ordered/unordered trees [24], labeled induced subgraphs [13], labeled subgraphs [14]. In *frequent itemset mining* [2] the pattern context is $(2^{\mathcal{I}}, \subseteq)$, where $\mathcal{I}$ is a given set, $\subseteq$ is the usual subset relation, and the input data is a sequence of *transactions* ($\mathcal{T} = \langle t_1, \ldots, t_m \rangle$). The elements of $\mathcal{I}$ are called *items* and each transaction is a set of items. The support of itemset I is the number of transactions that contain I as a subset.

There exist many algorithms which efficiently solve the FIM problem. Most of them are APRIORI and FP-growth based where efficiency comes from the sophisticated use of the trie data structure. The goal of our work is to scrutinize the use of trie theoretically and experimentally. Our claims are supported by hundreds of experiments based on many different databases with various support thresholds. We believe that a result of our work was to clarify important technical details of APRIORI, and to take some steps towards finding the best implementation.

3. Tries

The data structure *trie* was originally introduced by de la Briandais [9] and Fredkin [10] to store and efficiently retrieve words of a dictionary. A trie is a rooted, labeled tree. The root is defined to be at depth 0, and a node at depth d can point to nodes at depth $d + 1$. A pointer is also referred to as *edge* or *link*. If node u points to node v, then we call u the *parent* of v, and v is a *child* node of u. For the sake of efficiency – concerning insertion and deletion – a total order on the labels of edges has to be defined.

Tries are suitable for storing and retrieving not only words, but any finite sets (or sequences). In FIM algorithms tries (also called a lexicographic tree) are used to quickly determine the support of itemsets whose size is greater than 2. In the FIM setting a link is labeled by a frequent item, and a node represents an itemset, which is the set of the items in the path from the root to the leaf. The label of a node stores the counter of the itemset that the node represents.

Figure 1 presents a trie (without the counters) that stores the itemsets {A}, {C}, {E}, {F}, {A,C}, {A,E}, {A,F}, {E,F}, {A,E,F}. Building a trie is straightforward; we omit the details.

Tries can be implemented in many ways. In *compact representation* the edges of a node are stored in a vector. Each element of a vector is a pair; the first element stores the label of the edge, the second stores the address of the node, which the edge points to. This solution is very similar to the widespread "doubly chained" representation, where

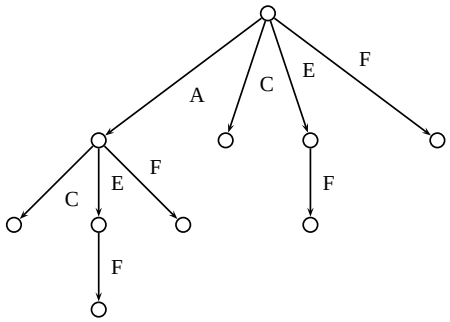


Figure 1. Example: a trie

edges of a node are stored in a linked list.

In the *non compact representation* (also called *tabular implementation* by Fredkin) only the pointers are stored in a vector with a length equal to that of the alphabet (frequent items in our case). An element at index i belongs to the edge whose label is the i^{th} item. If there is no edge with such a label, then the element is NIL. This solution has the advantage of finding an edge with a given label in $O(1)$ time, instead of $O(\log n)$ required by a binary search – which is the case in compact representation. Unfortunately for nodes with few edges this representation requires some more memory than compact representation. On the contrary, if a node has many edges (exact formula can be given based on the memory need of pointers and labels), then the non-compact representation needs less memory since labels are not stored explicitly. According to the memory need the two approaches can be combined [21], [23], [6]. If there are many nodes with single edges, then further memory can be saved by using patricia tries. Throughout this paper and in the implementations compact representation of tries is used.

Tries are used in FIM algorithms in two ways. In APRIORI based algorithms the tries store candidates (itemsets whose support has to be determined), and in APRIORI and FP-growth based algorithms the input sequence (more precisely a projection of the input) is stored in a trie.

4. Techniques for fast management

In this section we take trie issues one-by-one, describe the problem and present previous claims or naive expectations. Some issues apply to APRIORI and FP-growth based algorithms, but some apply only to the first algorithm family.

4.1. Storing the transactions

Let us call the itemset that is obtained by removing infrequent items from t the *filtered transaction* of t. All frequent itemsets can be determined even if only filtered transactions are available. To reduce IO cost and speed up the algorithm, the filtered transactions can be stored in main memory instead of on disk. It is useless to store the same filtered transactions multiple times. Instead store them once and employ counters which store the multiplicities. This way memory is saved and run-time can be significantly improved.

This fact is used in FP-growth and can be used in APRIORI as well. In FP-growth the filtered transactions are stored in an FP-tree, which is a trie with cross-links and a header table. Size of the FP-tree that stores filtered transactions is declared to be "substantially smaller than the size of database". This is said to come from the fact that a trie stores the same prefixes only once.

In the case of APRIORI, collecting filtered transactions has a significant influence on run-time. This is due to the fact that finding candidates that occur in a given transaction is a slow operation and the number of these procedure calls is considerably reduced. If a filtered transaction occurs n times, then the expensive procedure will be called just once (with counter increment n) instead of n times (with counter increment 1). A trie can be used to store the filtered transaction, and is actually used in the today's fastest APRIORI implementation made by Christian Borgelt [6].

Is trie really the best solution for collecting filtered transactions for APRIORI, or there exists a better solution? See the experimental results and explanation for the surprising answer.

4.2. Order of items

For quick insertion and to quickly decide if an itemset is stored in a trie, it is useful to store edges ordered according to their labels (i.e. items in our case) and apply a binary search. In [20] it was first noted that the order of the items affects the shape of the trie. The next figure shows an example of two tries, that store the same itemsets (ABC, ABD, ACE) but use different orders ($A > B > C > D > E$ and its reverse).

For the sake of reducing the memory need and traverse time, it would be useful to use the ordering that results in the minimal trie. Comer and Sethi proved in [8] that the minimal trie problem is NP-complete. On the other hand, a simple heuristic (which was employed in FP-growth) performs very well in practice; use the descending order according to the frequencies. This is inspired by the fact that tries store same prefixes only once, and there is a higher chance of itemsets having the same prefixes if frequent items have small indices.

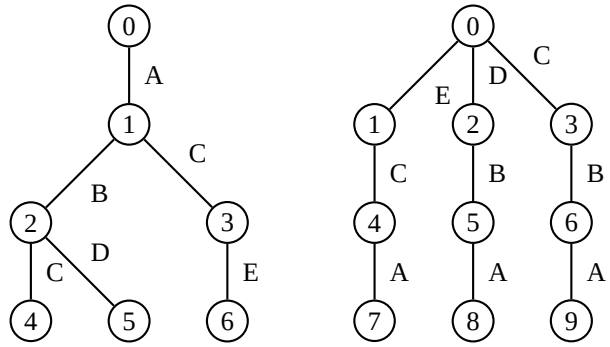


Figure 2. Example: Tries with different orders

The heuristic does not always result in the minimal trie, which is proven by the following example. Let us store itemsets AX, AY, BXK, BXL, BM, BN in a trie. A trie that uses descending order according to frequencies ($B \prec X \prec A \prec K \prec L \prec M \prec N$) is depicted on the left side of Figure 3. On the right side we can see a trie that uses an other order (items X and A are reversed) and has fewer nodes.

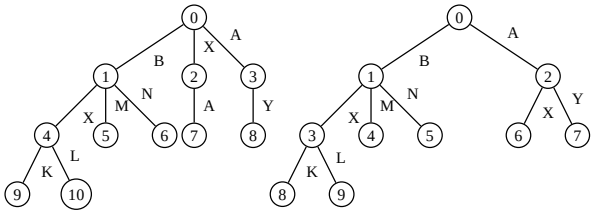


Figure 3. Example: descending order does not result the smallest trie

Descending order may not result in the smallest trie, when the trie has subtrees in which the order of items, according to the frequency of the subtree, does not correspond to the order of items according to the frequency in the whole trie. This seldom occurs in real life databases (a kind of homogeneity exists), which is the reason for the success of the heuristic.

Can this heuristic be applied in APRIORI as well? Fewer nodes require less memory, also fewer nodes need to be visited during the support count (fewer recursive steps), which would suggest faster operation. However previous observations [1] [6] claim the opposite. Here we conduct comprehensive experiments and try to find reasons for the contradiction.

4.3. Routing strategies at the nodes

Routing strategy in a trie refers to the method used at an inner node to select the edge to follow. To find out if a single itemset I is contained in a trie, the best routing strategy is to perform a binary search: at depth $d-1$ we have to find the edge whose label is the same as the d^{th} item of I. The best routing strategy is not so straightforward, if we have to find all the ℓ-itemset subsets of a given itemset, that are contained in a given trie. This is the main step of support count in APRIORI and this is the step that primarily determines the run-time of the algorithm. In this section we examine the possible solutions and propose a novel solution that is expected to be more efficient.

In support count methods we have to find all the leaves that represent ℓ-itemset candidates that are contained in a given transaction t. Let us assume that arrive at a node at depth d by following the j^{th} item of the transaction. We move forward on links that have labels $i \in t$ with an index greater than j, but less than $|t|-k+d+1$, because we need at least $k-d-1$ items to reach a leaf that represents a candidate. Or simply, given a part of the transaction (t'), we have to find the edges that correspond to an item in t'. Items in the transactions and items of the edges are ordered. The number of edges of the node is denoted by n. Two elementary solutions may be applicable:

simultaneous traversal: two pointers are maintained; one is initialized to the first element of t', and the other is initialized to the item of the first edge. The pointer that points to the smaller item is increased. If the pointed items are the same, then a match is found, and both pointers are increased. Worst case run-time is $O(n+|t'|)$.

binary search: This includes two basic approaches and the combination of both: for each item in t' we find the corresponding edge (if there is any), or for each edge the corresponding item of t'. Run-times of the two approaches are $O(|t'|\log n)$ and $O(n\log|t'|)$, respectively. Since the lists are ordered, it is not necessary to perform a binary search on the whole list if a match is found. For example if the first item in t' corresponds to the label of the fifth edge, then for the second element in t' we have to check labels starting from the sixth edge.

From the run-times we can see that if the size of t' is small and there are many edges, then the first kind of binary search is faster and in the opposite case the second kind is better. We can combine the two solutions: if $|t'|\log n < n\log|t'|$, then we perform a binary search on the edges – otherwise the binary search is applied on t'.

bitvector based: As mentioned before, a binary search can be avoided if a non-compact representation is used. However this increases the memory need. A much better solution is to change the representation of the filtered transaction rather than the nodes of the trie. We can use a bitvector instead of an ordered list. The element at index i is 1 if item i is stored in the transaction, and the length of the bitvector is the number of frequent items. A bitvector needs more space if the size of the filtered transaction is small, which is the general case in most applications. Hence, it is useful to store the filtered transactions as lists and convert them into bitvectors if stored candidates have to be determined. The run-time of this solution is $O(n)$.

indexvector based: The problem with bitvectors is that they do not exploit the fact that at a certain depth only a part of the transaction needs to be examined. For example, if the item of the first edge is the same as the last item of the basket, then the other edges should not be examined. The bitvector-based approach does not take into consideration the positions of items in the basket. We can easily overcome this problem if the indices of the items are stored in the vector. For example transaction $\{B,D,G\}$ is stored as $[0,1,0,2,0,0,3,0,0,0]$ if the number of frequent items is 10. The routing strategy with this vector is the following. We take the items of the edges one-by-one. If item i is the actual, we check the element i of the vector. If it is 0, the item is not contained. If the element is smaller than $|t|-k+d+1$ then match is found(and the support count is processed with the next edge). Otherwise the procedure is terminated. The worst case run-time of this solution is $O(n)$.

We have presented six different routing strategies that do not change the structure of the trie. Theoretically no best strategy can be declared. However, our experiments have shown that the solution we proposed last always outperforms the others.

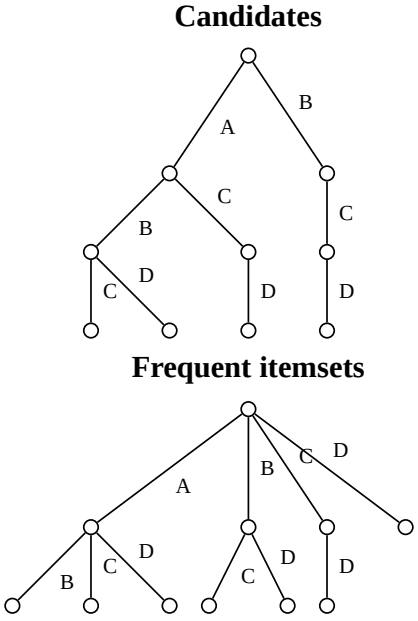
4.4. Storing the frequent itemsets

In FIM algorithms frequent itemsets that are not needed by the algorithm can be written to the disk, and memory can be freed. For example in APRIORI we need frequent items of size ℓ only for the candidate generation of $(\ell+1)$-itemsets, and later they are not used. Consequently memory need can be reduced if in the candidate generation the frequent itemsets are written out to disk and branches of the trie that do not lead to any candidate are pruned.

In most applications (for example to generate association rules) frequent itemsets are mined in order to be used. In such applications it is useful if the existence and the support

of the frequent itemsets can be quickly determined. Again, a trie is a suitable data structure for storing frequent itemsets, since frequent itemsets often share prefixes.

Regarding memory need, it is a wasteful solution to store frequent itemsets and candidates in a different trie. To illustrate this, examine the following tries.



The two tries above can easily be merged into one trie, which is shown in Figure 4.

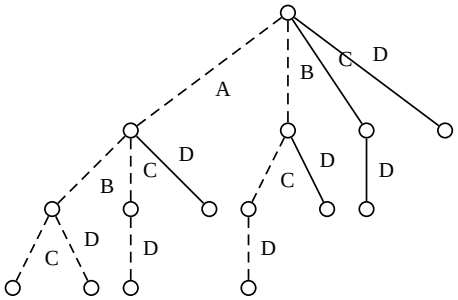


Figure 4. Example: Tries that store candidates and frequent itemsets

The memory need of the two tries (10 + 11 nodes) is more than the memory need of the merged trie (15 nodes), which stores candidates and frequent itemsets as well. The problem with a merged trie is that support counting becomes slower. This is due to the superfluous travel, i.e travel on routes that do not lead to candidates. For example, if the transaction contains items C, D, then we will follow the edges that start from the root and have labels C, D. This is obviously useless work since these edges do not lead to nodes at depth 3, where the candidates can be found.

To avoid this superfluous traveling, at every node we store the length of the longest directed path that starts from that node [5]. When searching for ℓ-itemset candidates at depth d, we move downward only if the maximum path length at the pointed node is $\ell - d + 1$. Storing maximum path lengths requires memory, but it considerably reduces the search time for large itemsets.

A better solution is to distinguish two kinds of edges. If an edge is on the way to a candidate, then it is a dashed edge and any other edges are normal edges. This solution is shown in Figure 4. Dashed and normal edges belonging to the same node are stored in different lists. During a support count only dashed edges are taken into consideration. This way many edges (the normal ones) are ignored even if their label corresponds to some element of the transaction. With this solution pointer increments are reduced (the list with dashed edges is shorter than the two lists together) and we do not need to check if the edge leads to a candidate (a comparison with an addition is spared).

4.5. Deleting unimportant transactions

Let us call a filtered transaction *unimportant* at the ℓ^{th} iteration of APRIORI, if it does not contain any $(\ell - 1)$-itemset candidates. Unimportant transactions need memory and slow down support count, since a part of the trie is visited but no leaf representing a candidate is reached. Consequently unimportant transactions should be removed, i.e. if a filtered transaction does not contain any candidate it should be removed from the memory and ignored in the later phases of APRIORI. Due to the anti-monotonic property of the support function, an ignored transaction can not contain candidates of greater sizes. Does this idea lead to faster methods? Surprisingly, experiments do not suggest that it does.

5. Experimental results

All tests were carried out on ten public "benchmark" databases, which can be downloaded from the FIM repository[2]. Seven different min_supp values were used for each database. Results would require too much space, hence only the most typical ones are shown below. All results, all programs and even the test scripts can be downloaded from `http://www.cs.bme.hu/~bodon/en/fim/test.html`.

Tests were run on a PC with a 1.8 GHz Intel P4 processor and 1 Gbytes of RAM. The operating system was Debian Linux (kernel version: 2.4.24). Run-times and memory

[2] `http://fimi.cs.helsinki.fi/data/`

usage were obtained using the `time` and `memusage` command respectively. In the tables, time is given in seconds and memory need in Mbytes. min_freq denotes the frequency threshold, i.e min_supp divided by the number of transactions.

5.1. Storing the transactions

First we compared three data structures used for storing filtered transactions. The memory need and construction time of the commonly used trie was compared to a sorted list and a red-black tree (denoted by RB-tree). RB-tree (or symmetric binary B-tree) is a self-balanced binary search tree with a useful characteristic: inserting an element needs $O(\log m)$, where m is the number of nodes (number of already inserted filtered transaction in our case).

min_ freq	sorted list	trie	RB-tree
0.05	12.4	61.1	13.8
0.02	16.2	88.5	17.1
0.0073	17.0	94.9	18.0
0.006	17.1	95.3	18.1

Database: T40I10D100K

Table 1. Memory need: storing filtered transactions

All 70 tests support the same observation: single lists need the least memory, RB-trees need a bit more, and tries consume the most memory – up to 5-6 times more than RB-trees.

The next figure shows a typical result on the construction and destruction time of the different data structures. Based

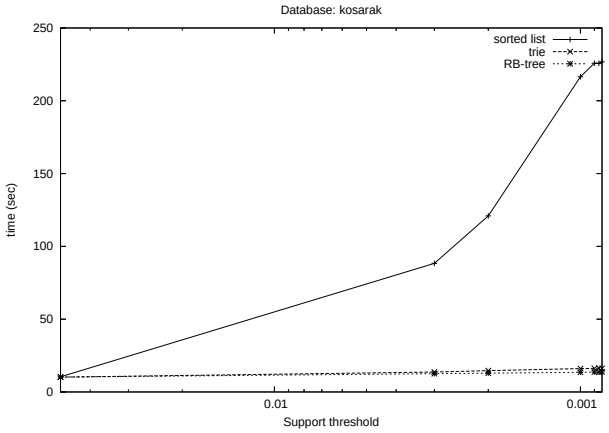


Figure 5. Construction and destruction time: storing filtered transactions

on the 70 measurements we can conclude that there is no significant difference if the database is small (i.e the number of filtered transaction is small), however – as expected – sorted lists do slow down as the database grows. RB-trees are always faster than tries, but the difference is not significant (it was always under 30%).

In support count of APRIORI, each filtered transaction has to be visited to determine the contained candidates. If transactions are stored in a sorted list, then going through the elements is a very fast operation. In the case of RB-trees and trie, this operation is slower, since a tree has to be traversed. However experiments showed that there is no significant difference when compared to building the data structure, so it does not merit serious consideration.

Experiments showed that RB-tree is the best data structure for storing filtered transactions. It needs little memory and it is the fastest with regard construction time. But why does RB-tree require less memory than trie, when trie stores the same prefixes only once and former and RB-tree stores them as many times as they appear in a filtered transaction?

The answer to this comes from the fact that a trie has many more nodes – therefore many more edges – than an RB-tree (except for one bit per node, RB-trees need the same amount of memory as simple binary trees need). In a trie each node stores a counter and a list of edges. For each edge we have to store the label and the identifier of the node the edge points to. Thus adding a node to a trie increases memory need by at least $5 \cdot 4$ bytes (if items and pointers are represented in 4 bytes). In a binary tree, like an RB-tree, the number of nodes equals to the number of filtered transactions. Each node stores a filtered transaction and its counter.

When inserting the first k-itemset filtered transaction in a trie, k nodes are created. However in an RB-tree we create only one node. Although the same prefixes are stored only once in a trie, this does not limit the memory increase as much. This is the reason that a binary tree needs 3-10 times less memory than a trie needs.

5.2. Order of items

To test the effect of ordering, we used 5 different orders: ascending and descending order by support (first item has the smallest/highest support) and three random orders. The results with the random orders were always between the results of the ascending and descending order. First the construction times and the memory needs of FP-trees (without cross links) were examined. Experiments showed that there is little difference in the construction time whichever type of ordering is used (the difference was always less than 8%). As expected, memory need is greatly affected by the ordering. Table 2 shows typical results.

Experiments with FP-trees with different orders meet our

min_freq (%)	1	0.2	0.09	0.035	0.02
ascending	42.48	58.03	61.34	63.6	65.04
descending	27.58	39.74	41.69	43.66	44.10
random 1	29.84	42.30	44.49	46.60	46.41
random 2	36.98	48.97	55.02	56.85	56.72
random 3	34.87	52.18	55.68	58.01	55.50

Database: BMS-POS

Table 2. Memory need: FP-tree with different orders

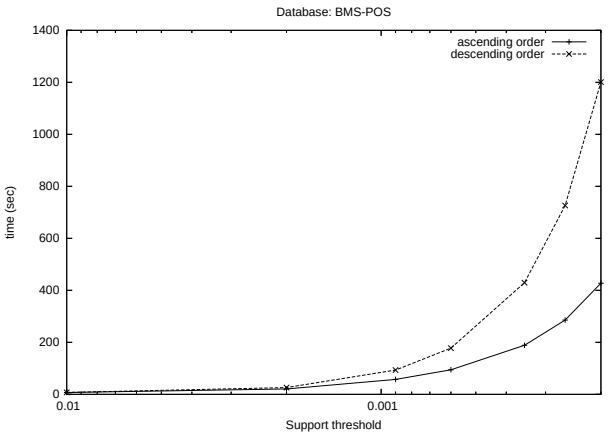


Figure 6. APRIORI with different orders

expectations: descending order leads to the smallest memory need, while ascending order leads to the highest. This agrees with the heuristic; a trie is expected to have small number of nodes if the order of the items corresponds to the descending order of supports.

Our experiments drew the attention to the fact that the memory need of FP-trees is greatly affected by the order used. The difference between ascending and descending order can be up to tenfold. In the basic FIM problem this does not cause any trouble; we can use the descending order. However in other FIM-related fields where the order of the items cannot be chosen freely, this side-effect has to be taken into consideration. Such fields are prefix anti-monotonic constraint based FIM algorithms, or FIM with multiple support threshold. For example, to handle prefix anti-monotonic constrains with FP-growth, we have to use the order determined by the constraint [19]. A naive solution for handling constraints is to add post processing to the FIM algorithm, where itemsets that do not return true on all constraints are pruned. It can be more efficient if the constraints are deeply embedded in the algorithm. In [19] it was shown that a single prefix anti-monotonic predicate can be effectively treated by FP-growth. Our experiments proved that FP-growth is very sensitive to the order of the items. Consequently, embedding a prefix anti-monotonic constraint into FP-growth does not trivially decrease resource need. Although search space can be reduced, we have seen that this may greatly increase memory need and thus traversal time.

Results on the effect of ordering in APRIORI are surprising (Figure 6). They contradict our initial claim. In almost all experiments APRIORI with the ascending order turned out to be the fastest, and the one that used descending ordering was the slowest. Highest difference (6 fold) was in the case of `retail.dat` [7].

These experiments support the previously stated, but further unexplained observation, i.e. ascending order is the best order to use in APRIORI. We have seen that a trie that uses descending order is expected to have fewer nodes than a trie that uses ascending order. However, ascending order has two advantages over descending order. First, nodes have fewer edges and hence the main step of support count (finding the corresponding edges at a given node) is faster. The second and more important factor is that a smaller number of nodes is visited during the support count. This comes from the fact that nodes near the root have edges with rare frequent items as labels. This means that in the beginning of the support count the most selective items are checked. Many transactions do not contain rare frequent items, hence the support count is terminated in the earliest phase. On the contrary, when descending order is used, many edges are visited before we get to the selective items. To illustrate this fact, let us go back to Figure 2 and consider the task of determining the candidates in transaction $\{A, B, F, G, H\}$. Nodes 0,1,2 will be visited if descending order is used, while the search will be terminated immediately at the root in the case of the ascending order.

Concerning memory need, as expected, descending order is the best solution. However its advantage is insignificant. APRIORI with ascending order never consumed more than 2% extra memory compared to APRIORI with descending order.

5.3. Routing strategies at the nodes

In the experiments of APRIORI with different routing strategies, we tested 5 competitors: (1) simultaneous traversal, (2-3) corresponding items were found by a binary search on the items of the transaction/labels of the edges, and (4-5) transactions were represented by bitvectors/vectors of indices. The results can not be characterized by a single table. Figure 7 and 8 present results with two databases. For a more comprehensive account the reader is referred to the aforementioned test web page.

Our newly proposed routing technique (i.e. indexvector based) almost always outperformed the other routing strategies. Simultaneous traversal was the runner up. The other

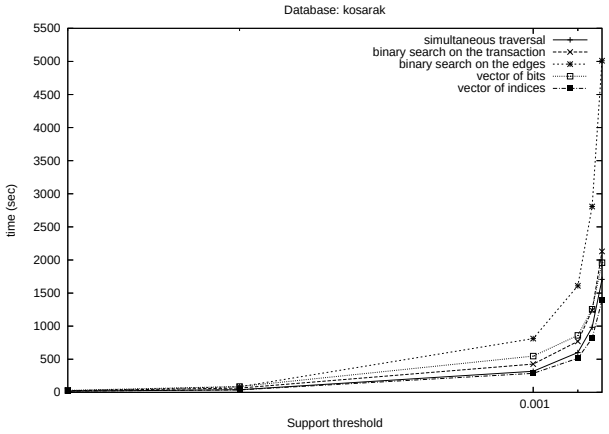


Figure 7. APRIORI with different routing strategies

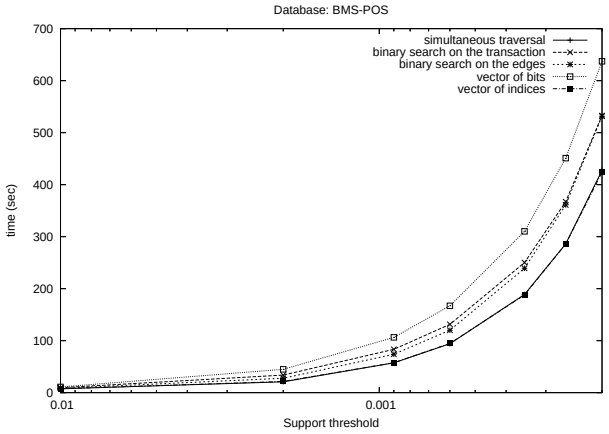


Figure 8. APRIORI with different routing strategies

three alternated in performance. The bitvector approach was most often the least effective. Routing that employed binary search on the edges sometimes lead to extremely bad result,but on one database (`retail`) it finished in first place.

Simultaneous traversal performed very well, and almost always beat the binary search approaches. Theoretical runtimes do not necessarily support this. To understand why simultaneous traversal outperformed the binary search based approaches, we have to take a closer look at them.

Simultaneous traversal is a very simple method; it compares the values of the pointers and increments one or both of them. These elementary operations are very fast. In binary search approach we increment pointers and invoke binary searches. In each binary search we make comparisons, additions and even divisions. If we take into consideration that calling a subroutine with parameters always means extra value assignments, then we can conclude the overhead of the binary search is significant and is not profitable, if the list we are searching through is short. In our case neither the number of edges of the nodes nor the number of frequent items in the transaction is large. This explains the bad performance of binary search in our case.

By using a binary vector we can avoid binary search with all of its overhead. However, experiments showed that although the bitvector based approach was better than binary search-based approaches, it could not outperform the simultaneous traversal. This is because a bitvector-based approach does not take into consideration that only a part of the transaction has to be examined. Let us see an example. Assume that the only 4-itemset candidate is $\{D, E, F, G\}$ and we have to find the candidates in transaction $\{A, B, C, D, E, F\}$. Except for the bitvector-based approach all the techniques considered will not visit any node in the trie, because there is no edge of the root whose label corresponds to any of the first $6 - 4 + 1 = 3$ items in the transaction. On the contrary, the bitvector-based approach uses the whole transaction and starts with a superfluous travel that goes down even to depth 3. A vector that stores the indices (the 5^{th} competitor) overcomes this deficiency. This seems to be the reason behind the good performance (first place most of the time).

5.4. Storing the frequent itemsets

Figure 9 shows typical run-times of three different variants of APRIORI. In the first and second frequent itemset were stored in the memory, in the third they were written to disk. To avoid superfluous traversing, maximum path values were used in the first APRIORI, and different lists of edges in the second.

Memory needs of the three implementations are found in the next table.

min_ freq (%)	maximum paths	different edgelists	written out
0.064	3.48	3.98	2.38
0.062	7.38	8.98	3.61
0.06	14.3	17.9	6.0
0.058	33.5	43.5	13.2
0.056	133.6	176.0	47.8

Database: BMS-WebView-1

Table 3. Memory need: different frequent itemset handling

As expected, we obtain the fastest APRIORI if frequent itemsets are not stored in memory but written to disk in the

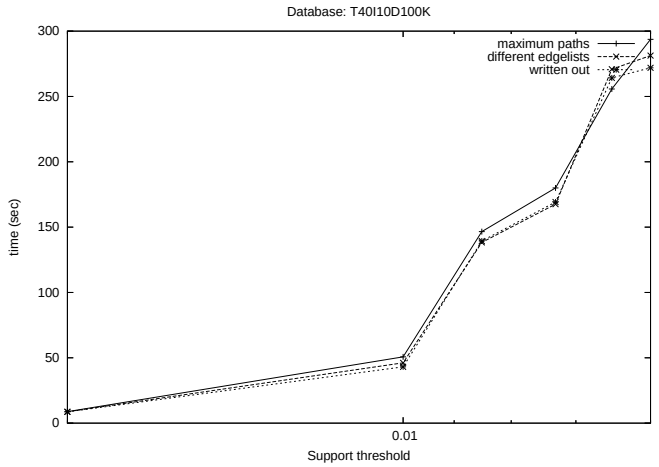


Figure 9. APRIORI with different frequent itemset handling techniques

candidate generation process. Experiments show that this APRIORI, however, is not significantly faster than APRIORI that stores maximum path lengths. This comes from the fact that APRIORI spends most of the time in determining the support of small and medium sized candidates. In such cases most edges lead to leaves, hence removing other edges does not accelerate the algorithm too much.

However, the memory need can be significantly reduced if frequent itemsets are not stored in memory. Experiments show that memory need may even decrease to the third or the quarter. Consequently if frequent itemsets are needed to determine valid association rules and memory consumption is an important factor, then it is useful to write frequent itemsets to disk and read them back when the association rule discovery phase starts.

5.5. Deleting unimportant transactions

Test results for deleting unimportant transactions contradict our expectations. Typical run-times are listed in Table 4.

min_freq (%)	90	81	75	71	69	67
non-delete	4.76	11.82	65.1	430	905	1301
delete	4.48	12.09	66.5	442	917	1339

Database: pumsb

Table 4. Run-time: deleting unimportant transactions

In six out of ten databases, deleting unimportant transaction slowed down APRIORI. In the other three databases the difference was insignificant (under 1%). The trick accelerated the algorithm only for the `retail` database.

Deleting unimportant transactions was expected to decrease run-time. However test results showed the contrary. This is attributed two extra cost factors that come into play as soon as we want to delete unimportant transactions.

First, we have to determine if a given transaction contains any candidates. This means some overhead (one assignment at each elementary step of the support count) and does not save time during APRIORI in cases where the transaction contains candidates (which is the typical case).

The second kind of extra cost comes from the fact that filtered transactions were stored in an RB-tree. For this we used `map` implemented in STL. However deleting an entry from an RB-tree is not as cheap as deleting a leaf from a simple binary tree. The red-black property has to be maintained which sometimes requires the expensive rotation operation. Notice that after determining the multiplicity of the filtered transactions we don't need to maintain any sophisticated data structures, only the filtered transactions and the multiplicity values are needed for the support count. Consequently, the second extra cost problem can be overcome in two ways. We can copy the filtered transactions and the counters of the RB-tree into a list or we may let the delete operations invalidate the red-black property of the tree. Test results showed that even with these modifications, deleting unimportant transactions does not lead to a faster APRIORI.

5.6. Overall performance gain

With our last experiment we would like to illustrate the overall performance gain of the prospective improvements. Two APRIORI implementations with different trie related options are compared. In the first ascending order, simultaneous traversal is used and filtered transactions are stored in an RB-tree. In the second implementation descending order, binary search on the edges is applied and filtered transactions are not collected.

min_freq (%)	90	83	75	71	67	65.5
original	213	2616	16315	34556	71265	stopped
new	3.5	9.3	66	158	365	706

Database: connect

Table 5. Comparing run-time of two APRIORI with different options

The results support our claim, that suitable data structure techniques lead to a remarkable improvements.

5.7. Effect of programming techniques

Most papers on FIM focus on new algorithms, new candidate generation methods, support count techniques or data structure-related issues. Less attention is paid to details, like the ones mentioned above, despite the fact that these tricks are able to speed up algorithms that are not regarded as being very fast. The fastest APRIORI implementation [6], which incorporates many sophisticated techniques, supports this claim by finishing among the best implementations (beating many newer algorithms) in the first FIM contest [11].

Besides the algorithmic and data-structure issues there is a third factor that quite possibly influences effectiveness. This factor is programming technique. FIM algorithms are computationally expensive and therefore, no algorithms that are implemented in a high level programming language (like Java, C#) are competitive with lower level implementations. The language C is widely used, flexible and effective, which is the reason why every competitive FIM implementation is implemented in C.

Unfortunately, C gives too much freedom to the implementors. Elementary operations like binary search, building different trees, list operations, memory-allocation, etc. can be done in many ways. This is a disadvantage because efficiency depends on the way these elementary operations are programmed. This may also be the reason for the performance gain of a FIM implementation. An experienced programmer is more likely to code a fast FIM algorithm than a FIM expert with less programming experience.

A tool that provides standard procedures for the elementary operations has double advantage. Efficiency of the implementation would only depend on the algorithm itself. The code would be more readable and maintainable because of the higher level of abstraction. Such a tool exists – it is the C++ and the Standard Template Library (STL). STL provides containers (general data structures), algorithms and functions that were carefully programmed by professional programmers. By using STL the code will be easier to read and less prone to error, while maintaining efficiency (STL algorithms are asymptotically optimal). Due to these advantages, STL should be used whenever possible. Actually it could be the "common language" among data mining programmers.

Besides the aforementioned advantages of STL, it also introduces some dangers. To make good use of STL's capabilities, we first need to have more advanced knowledge about them. Our experiments on STL-related issues showed that small details and small changes can lead to high variations in run-time or memory need. The goal of this paper is to draw attention to data-structure related issues, however, we also have to mention some STL-related issues that have to be taken into careful consideration. Next, we list some of the factors we ran into that have a large impact on efficiency. These are: (1) when to use sorted vector instead of an RB-tree (i.e. sorted `vector` vs. `map`), (2) when to a store pointers instead of objects in a container (double referencing vs. copy constructor), (3) memory management of the containers and the need for using the "swap trick" to avoid unnecessary memory occupation, (4) contiguous-memory containers vs. node-based containers, (5) iterators vs. using the index operator, etc. These issues are important. For example, when a vector storing pointers of objects was substituted by a vector that stores simply the objects and copy constructor overhead was avoided by reserving memory in advance, the run-time decreased significantly (for example to one third in the case of the T10I4D100K database). For more information on STL-related questions, the reader is referred to [22] [17].

6. APRIORI implementation submitted to FIMI'04

Based on our theoretical and experimental analysis, we implemented a fast APRIORI algorithm. Since we believe that readability is also very important we sacrificed an insignificant amount of efficiency if it led to a simpler code. Our final implementation uses a red-black tree to store filtered transactions, item order is ascending according to their support, simultaneous traversal is used as a routing strategy, nodes representing frequent itemsets, but playing no role in support count, are pruned and unimportant filtered transactions are not removed. Moreover, a single vector and an array is used to quickly find the support of one and two itemset candidates [18] [5]. The implementation (version 2.4.1 at the time of writing) is fully documented and can be freely downloaded for *research* purposes at `http://www.cs.bme.hu/~bodon/en/apriori`.

A version that uses a simplified output format and configuration options was submitted to the FIMI'04 contest.

7. Conclusion

In this paper we analyzed speed-up techniques of trie-based algorithms. Our main target was the algorithm APRIORI, however, some trie-related issues also apply to other algorithms like FP-growth. We also presented new techniques that result in a faster APRIORI.

Experiments proved that these data-structure issues greatly affect the run-time and memory consumption of the algorithms. A carefully chosen combination of these techniques can lead to a 2 to 1000 fold decrease in run-time, without significantly increasing memory consumption.

Acknowledgment

The author would like to thank Balázs Rácz for his valuable STL and programming related comments and for insightful discussions.

References

[1] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, 2001.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 207–216, 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *The International Conference on Very Large Databases*, pages 487–499, 1994.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In P. S. Yu and A. L. P. Chen, editors, *Proceedings of the 11th International Conference on Data Engineering, ICDE*, pages 3–14. IEEE Computer Society, 6–10 1995.

[5] F. Bodon. A fast apriori implementation. In B. Goethals and M. J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 19. November 2003.

[6] C. Borgelt. Efficient implementations of apriori and eclat. In B. Goethals and M. J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 19. November 2003.

[7] T. Brijs, G. Swinnen, K. Vanhoof, and G. Wets. Using association rules for product assortment decisions: A case study. In *Proceedings of the sixth International Conference on Knowledge Discovery and Data Mining*, pages 254–260, 1999.

[8] D. Comer and R. Sethi. The complexity of trie index construction. *J. ACM*, 24(3):428–440, 1977.

[9] R. de la Briandais. File searching using variable-length keys. In *Western Joint Computer Conference*, pages 295–298, March 1959.

[10] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[11] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: Introduction to fimi03. In B. Goethals and M. J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 19. November 2003.

[12] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 1–12. ACM Press, 2000.

[13] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 13–23. Springer-Verlag, 2000.

[14] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the first IEEE International Conference on Data Mining*, pages 313–320, 2001.

[15] H. Mannila and H. Toivonen. Discovering generalized episodes using minimal occurrences. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*, pages 146–151. AAAI Press, August 1996.

[16] H. Mannila, H. Toivonen, and A. Inkeri Verkamo. Discovering frequent episodes in sequences. In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95)*, pages 210–215. AAAI Press, August 1995.

[17] S. Meyers. *Effective STL: 50 specific ways to improve your use of the standard template library*. Addison-Wesley Longman Ltd., 2001.

[18] B. Özden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 412–421. IEEE Computer Society, 1998.

[19] J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent item sets with convertible constraints. In *Proceedings of the 17th International Conference on Data Engineering*, pages 433–442. IEEE Computer Society, 2001.

[20] T. Rotwitt, Jr. and P. A. D. de Maine. Storage optimization of tree structured files. In E. F. Codd and A. L. Dean, editors, *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control*, pages 207–217. ACM, November 11-12 1971.

[21] D. G. Severance. Identifier search mechanisms: A survey and generalized model. *ACM Comput. Surv.*, 6(3):175–194, 1974.

[22] B. Stroustrup. *The C++ Programming Language, Special Edition*. Addison-Wesley Verlag, Bosten, 2000.

[23] S. B. Yao. Tree structures construction using key densities. In *Proceedings of the 1975 annual conference*, pages 337–342. ACM Press, 1975.

[24] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80. ACM Press, 2002.

[25] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In D. Heckerman, H. Mannila, D. Pregibon, R. Uthurusamy, and M. Park, editors, *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 283–296. AAAI Press, 12–15 1997.