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 edges of a node are stored in a linked list.

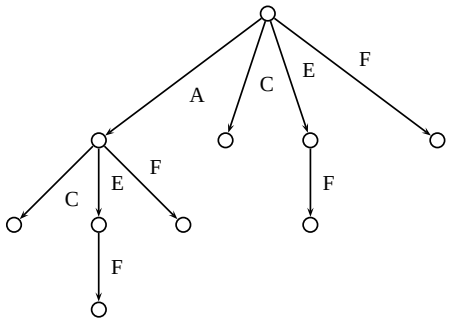


Figure 1. Example: a trie

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.

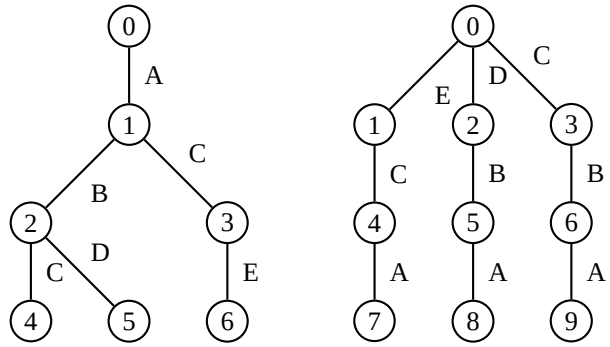


Figure 2. Example: Tries with different orders

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.

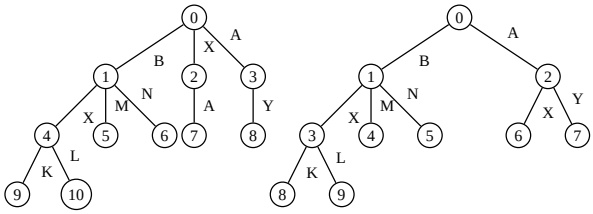


Figure 3. Example: descending order does not result the smallest trie

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 homogeneousity 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.

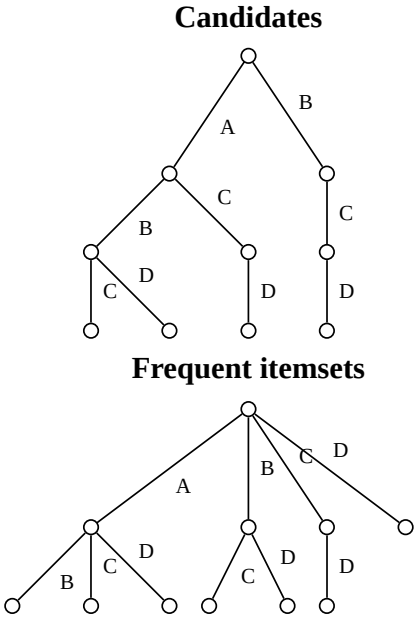
4.4. Storing the frequent itemsets

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.

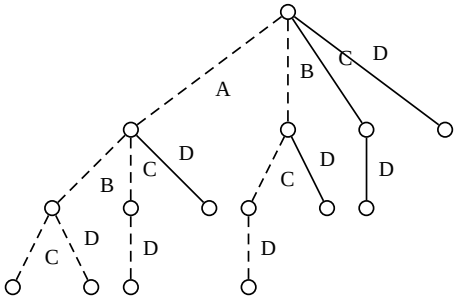


Figure 4. Example: Tries that store candidates and frequent itemsets

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

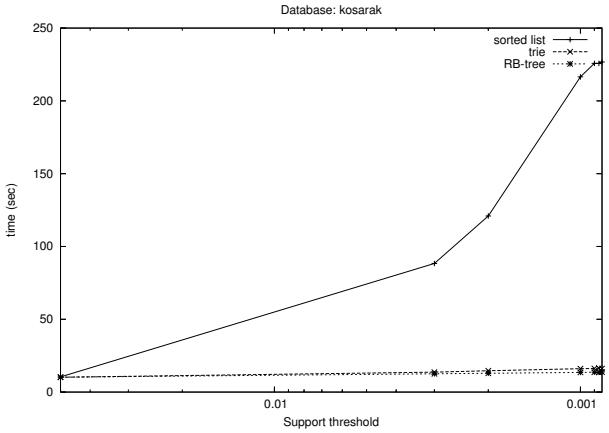


Figure 5. Construction and destruction time: storing filtered transactions

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

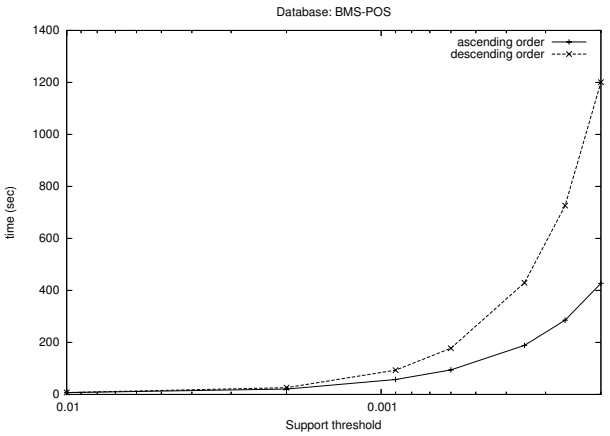


Figure 6. APRIORI with different orders

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

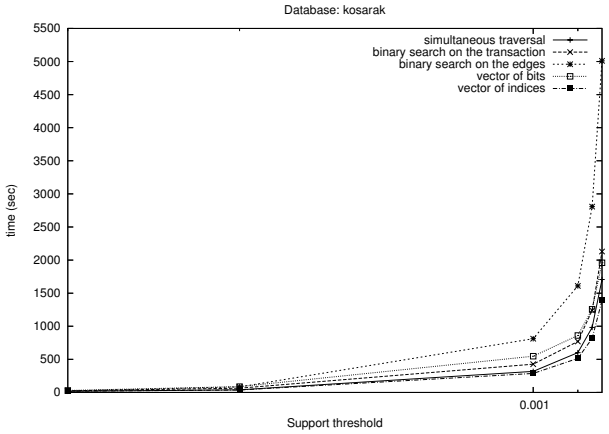


Figure 7. APRIORI with different routing strategies

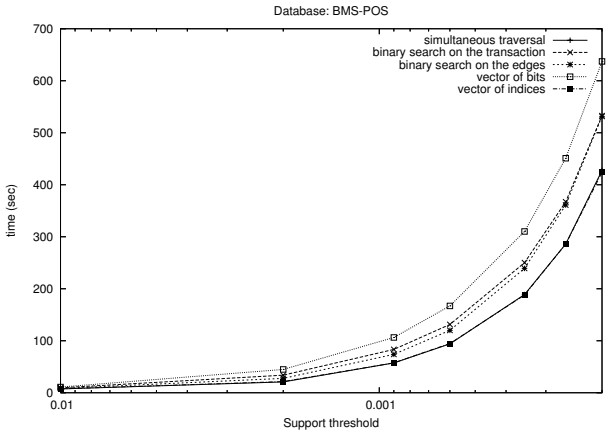


Figure 8. APRIORI with different routing strategies

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

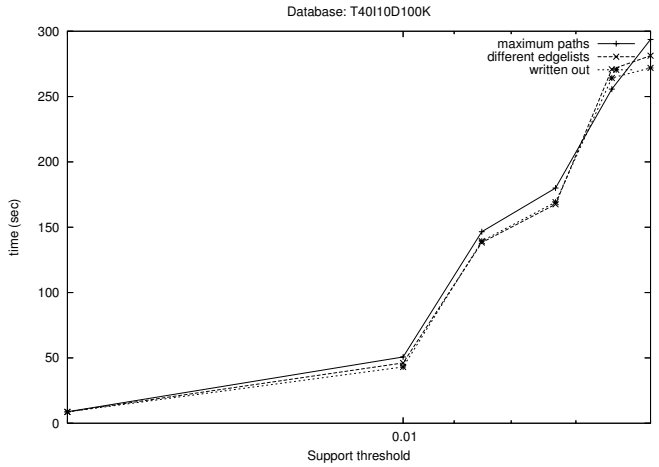


Figure 9. APRIORI with different frequent itemset handling techniques

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.

ABS: Adaptive Borders Search of frequent itemsets

Frédéric Flouvat[1], Fabien De Marchi[2], Jean-Marc Petit[1]

[1] Laboratoire LIMOS, UMR CNRS 6158
Université Blaise Pascal - Clermont-Ferrand II,
24 avenue des Landais, 63 177 Aubière cedex, France
flouvat@isima.fr, jmpetit@math.univ-bpclermont.fr

[2] Laboratoire LIRIS, FRE CNRS 2672
Université Claude Bernard Lyon 1
8, boulevard Niels Bohr, 69 622 Villeurbanne cedex France
fabien.demarchi@liris.cnrs.fr

Abstract

In this paper, we present an ongoing work to discover maximal frequent itemsets in a transactional database. We propose an algorithm called ABS for Adaptive Borders Search, which is in the spirit of algorithms based on the concept of dualization. From an abstract point of view, our contribution can be seen as an improvement of the basic APRIORI algorithm for mining maximal frequent itemsets. The key point is to decide dynamically at which iteration, if any, the dualization has to be made to avoid the enumeration of all subsets of large maximal itemsets. Once the first dualization has been done from the current negative border, APRIORI is no longer used and instead, another dualization is carried out from the positive border known so far. The process is repeated until no change occurs anymore in the positive border in construction.

Experiments have been done on FIMI datasets from which tradeoffs on adaptive behavior have been proposed to guess the best iteration for the first dualization. Far from being the best implementation wrt FIMI'03 contributions, performance evaluations of ABS exhibit better performance than IBE, the only public implementation based on the concept of dualization.

1 Introduction

In this paper, we present an ongoing work to discover maximal frequent itemsets in a transactional database. We propose to adapt an algorithm originally devised for mining inclusion dependencies in databases [8]. This algorithm is called ABS for Adaptive Borders Search, and is in the spirit of algorithms based on the concept of dualization [14, 21].

The basic idea of our proposition is to combine the strength of both levelwise algorithm [1, 18] and *Dualize and Advance* algorithm [14] in such a way that:

- "small" maximal frequent itemsets are efficiently generated with levelwise strategies.
- "large" maximal frequent itemsets may be found efficiently by dualization.

The dualization performed is quite similar to that proposed in the *Dualize and Advance* algorithm. Nevertheless, instead of starting from some subset of maximal frequent itemsets as *Dualize and Advance* algorithm does, we use infrequent itemsets to perform the dualization. As a consequence, we obtain the so-called *optimistic positive border* of maximal frequent itemsets. The set of such candidates corresponds exactly to k-uniform hypergraph clique proposed in [22]. As a consequence, our proposition contributes to clarify some related contributions [22, 17, 3, 12, 7]) since it gives an exact characterization of the optimistic positive border of maximal frequent itemsets from some subset of infrequent itemsets.

From an abstract point of view, our contribution can be seen as an improvement of the basic APRIORI algorithm for mining maximal frequent itemsets. The key point is to decide dynamically at which iteration, if any, the dualization has to be made to avoid the enumeration of all subsets of large maximal itemsets. Once the first dualization has been done from the current negative border available at that

iteration, APRIORI is no longer used and instead, another dualization is carried out from the positive border known so far. The process is repeated until no change occurs anymore in the positive border in construction.

Experiments have been done on FIMI datasets [10]. The adaptive behavior of our algorithm has been tuned from results gathered from these experiments. For the tested dataset, we were able to guess dynamically the best iteration for the first dualization, a key parameter of our algorithm.

Far from being the best implementation wrt FIMI'03 contributions [11], performance evaluations of ABS exhibit better performance than IBE [21], the only public implementation based on the concept of dualization.

2 Preliminaries

Let R be a set of symbols called *items*; a *line* is a subset of *R*, and a *binary relation* r over R is a multiset of lines. We suppose the reader is familiar with the notions of itemsets, support, and with the main aspects of frequent itemsets mining problem in a binary relation, given a threshold $minsup$ (see e.g. [1] for details). We recall the notion of borders of a set of itemsets [18]. Given F a set of itemsets over R, the *positive border* of F denoted by $\mathcal{B}d^+(F)$ is defined by $\mathcal{B}d^+(F) = max_{\subseteq}\{X \in F\}$. The negative border of F is defined by $\mathcal{B}d^-(F) = min_{\subseteq}\{Y \subseteq R \mid \forall X \in F, Y \not\subseteq X\}$. If FI is the set of all itemsets frequent in r, then $\mathcal{B}d^+(FI)$ is called the set of *maximal frequent itemsets* in r.

We will use the concepts of hypergraph and minimal transversal of a hypergraph, whose definition is pointed out here (see for example [4] for more details). Given *V* a finite set of elements. A subset *E* of *V* defines a *hypergraph* $\mathcal{H} = (V, E)$, where elements of *V* are called *vertices* of $\mathcal{H}$ and elements of *E* *edges* of $\mathcal{H}$. A *transversal* *T* of $\mathcal{H} = (V, E)$ is a subset of *V* that intersect all the elements of *E*. *T* is *minimal* if no other transversal of $\mathcal{H}$ are included in *T*. The set of all minimal transversals of $\mathcal{H}$ is noted $Tr(\mathcal{H})$.

The relationship between the notion of borders and minimal transversals of hypergraph has been exhibited in [18]. Indeed, any set of itemsets can be seen as a hypergraph; if FI is the set of frequent itemsets in a binary relation r, we have: $Tr(\overline{FI}) = \mathcal{B}d^-(FI)$, where $\overline{FI} = \{R - X \mid X \in FI\}$.

3 Method description

3.1 Starting with a levelwise algorithm

The algorithm $Apriori$ [1] was initially devoted to frequent itemset mining; Nevertheless, it has been proved to be still competitive for maximal frequent itemsets mining in many cases [11], when the size of elements to discover remain small.

Our goal is to exploit the efficiency of $Apriori$, but to automatically detect when it will fall into troubles and stop its execution. Then we propose to exploit the knowledge mined so far to initialize a different search, based on the concept of dualization between positive and negative borders; each border is updated and used to compute the corresponding dual border, until a fix point is reached.

3.2 From negative to positive border

In the sequel, let r be a binary database over a set of items R, $minsup$ a minimum support, and FI the set of frequent itemsets in r. After the levelwise part, our method is still iterative; at each iteration i, new elements of the positive and negative borders are expected to be discovered. We denote by $\mathcal{B}d_i^+$ (resp. $\mathcal{B}d_i^-$) the subset of $\mathcal{B}d^+(FI)$ (resp. $\mathcal{B}d^-(FI)$) discovered until the i^{th} iteration. In other words, $\forall i < j, \mathcal{B}d_i^+ \subseteq \mathcal{B}d_j^+$ and $\mathcal{B}d_i^- \subseteq \mathcal{B}d_j^-$. Roughly speaking, candidates for $\mathcal{B}d_i^+$ are obtained from elements of $\mathcal{B}d_i^-$, and candidates for $\mathcal{B}d_{i+1}^-$ are obtained from elements of $\mathcal{B}d_i^+$.

The following definitions and results have been proposed in [8] for inclusion dependency discovery problem in relational databases. We recall them in the context of maximal frequent itemsets mining, only the proofs are omitted.

We first define the notion of *Optimistic positive border*.

Definition 1*(Optimistic positive border) Given a set F of itemsets, the optimistic positive border of F is:* $\mathcal{F}opt(F) = max_{\subseteq}\{X \subseteq R \mid \forall Y \in F, Y \not\subseteq X\}$.

The next theorem gives a constructive characterization of $\mathcal{F}opt(F)$.

Theorem 1*[8]* $\mathcal{F}opt(F) = \overline{Tr(F)}$

Therefore, the idea is to compute the optimistic positive border for $\mathcal{B}d_i^-$ to obtain exactly the largest itemsets which do not contain any infrequent itemset discovered so far.

Proposition 1 *Let $X \in \mathcal{F}opt(\mathcal{B}d_i^-)$. If $sup(X) \geq minsup, X \in \mathcal{B}d^+(FI)$.*

Proof *Since X is maximal in the definition of $\mathcal{F}opt(\mathcal{B}d_i^-)$, each of its superset contains at least one element of $\mathcal{B}d_i^-$, and is infrequent by anti-monotonicity.* □

Then, $\mathcal{B}d_i^+$ is exactly made up of all the frequent itemsets in $\mathcal{F}opt(\mathcal{B}d_i^-)$.

3.3 From positive to negative border

In a dual way, the set $\mathcal{B}d_i^+$ is then used to compute its negative border $\mathcal{B}d^-(\mathcal{B}d_i^+)$, to finally update the negative border in construction and obtain $\mathcal{B}d_{i+1}^-$.

The next theorem gives a constructive characterization of $\mathcal{B}d^-(F)$, for any set F of frequent itemsets.

Theorem 2*[18]* $\mathcal{B}d^-(F) = Tr(\overline{F})$

Proposition 2 *Let* $X \in \mathcal{B}d^-(\mathcal{B}d_i^+)$. *If* $sup(X) < minsup, X \in \mathcal{B}d^-(FI)$.

Proof *Let X be an element of $\mathcal{B}d^-(\mathcal{B}d_i^+)$. By the definition of the negative cover of a set, each subset of X is included in an element of $\mathcal{B}d^+(FI)$ and then is frequent.* □

Then, $\mathcal{B}d_{i+1}^-$ is exactly made up of all the infrequent itemsets in $\mathcal{B}d^-(\mathcal{B}d_i^+)$.

3.4 The Algorithm ABS

Algorithm 1 computes the positive and negative borders of frequent itemsets in a given binary database. Within the framework of levelwise algorithms, ABS decides at each level whether or not the levelwise approach has to be stopped. In that case, the levelwise approach is halted, and the two borders are incrementally updated as described previously. The functions $GenPosBorder$ and $GenNegBorder$ compute respectively the optimistic positive and negative borders, using characterizations in theorems 1 and 2. The algorithm terminates when all elements of the optimistic positive border currently computed are frequent. It is worth noting that no dualization may occur at all: in this case, ABS is reduced to $APRIORI$. The proposition 3 ensures the correctness of ABS.

The behavior of the function $IsDualizationRelevant$ is described in section 3.5.

Proposition 3 *The algorithm ABS returns $\mathcal{B}d^+(FI)$ and $\mathcal{B}d^-(FI)$.*

Proof *If the test performed by $IsDualizationRelevant()$ is never true, the demonstration is obvious.*
If not, in line 15, from propositions 1 and 2, we have $\mathcal{B}d_i^+ \subseteq \mathcal{B}d^+(FI)$ and $\mathcal{B}d_{i-1}^- \subseteq \mathcal{B}d^-(FI)$
Moreover, the termination condition ensures that $\mathcal{B}d_i^+ = GenPosBorder(\mathcal{B}d_{i-1}^-)$; all elements in $\mathcal{B}d_i^+$ are frequent and all elements in $\mathcal{B}d_{i-1}^-$ are infrequent. Suppose that $\exists X \in \mathcal{B}d^-(FI) \mid X \notin \mathcal{B}d_{i-1}^-$. Then:

- *if $\exists Y \in \mathcal{B}d_{i-1}^- \mid Y \subset X$, since Y is infrequent, $X \notin \mathcal{B}d^-(FI)$ and there is a contradiction*
- *if $\nexists Y \in \mathcal{B}d_{i-1}^- \mid Y \subseteq X$, then from the definition of the optimistic positive border $\exists Z \in \mathcal{B}d_i^+ \mid X \subseteq Z$, which contradict the fact that X is infrequent.*

Thus $\mathcal{B}d_{i-1}^- = \mathcal{B}d^-(FI)$. An identical reasoning leads to $\mathcal{B}d_i^+ = \mathcal{B}d^+(FI)$.

□

Algorithm 1 ABS: Adaptive Border Search

Require: a binary database r, a integer $minsup$
Ensure: $\mathcal{B}d^+(FI)$ and $\mathcal{B}d^-(FI)$
1: $F_1 = \{A \in R \mid \text{sup}(A) \geq minsup\}$
2: $C_2 = AprioriGen(F_1)$
3: $i = 2$; $\mathcal{B}d_1^- = R - F_1$; $\mathcal{B}d_0^+ = \emptyset$
4: **while** $C_i \neq \emptyset$ **do**
5: $F_i = \{X \in C_i \mid sup(X) \geq minsup\}$
6: $\mathcal{B}d_i^- = \mathcal{B}d_{i-1}^- \cup (C_i - F_i)$
7: $\mathcal{B}d_{i-1}^+ = \mathcal{B}d_{i-2}^+ \cup \{X \in F_{i-1} \mid \forall Y \in F_i, X \not\subseteq Y\}$
8: **if** $IsDualizationRelevant(i, |\mathcal{B}d_i^-|, |F_i|, |C_i|) = TRUE$ **then**
9: $\mathcal{B}d_i^+ = \{X \in GenPosBorder(\mathcal{B}d_i^-) \mid |X| \geq minsup\}$
10: **while** $\mathcal{B}d_i^+ \neq \mathcal{B}d_{i-1}^+$ **do**
11: $\mathcal{B}d_i^- = \{X \in GenNegBorder(\mathcal{B}d_i^+) \mid |X| \leq minsup\}$
12: $\mathcal{B}d_{i+1}^+ = \{X \in GenPosBorder(\mathcal{B}d_i^-) \mid |X| \geq minsup\}$
13: $i = i + 1$
14: **end while**
15: **Return** $\mathcal{B}d_i^+$ and $\mathcal{B}d_{i-1}^-$ and **exit**
16: **end if**
17: $C_{i+1} = AprioriGen(F_i)$
18: $i = i + 1$
19: **end while**
20: $\mathcal{B}d_{i-1}^+ = \mathcal{B}d_{i-2}^+ \cup F_{i-1}$
21: **Return** $\mathcal{B}d_{i-1}^+$ and $\mathcal{B}d_{i-1}^-$

3.5 Adaptive aspects of ABS

The main adaptive aspect of ABS is conveyed by the function $IsDualizationRelevant$, line 8 of algorithm 1. As mentioned, its goal is to estimate if it is interesting to dualize the current negative border to the optimistic positive border.

We have identified four parameters specific to a given iteration of the levelwise algorithm, which can be obtained dynamically without any overhead:

- The current level i. No jump is allowed until a given integer threshold; we set the threshold equal to 4, since $Apriori$ is very efficient in practice to explore the levels 1 to 4. In our experiments, dualizing before this level incurs no improvement.
- $|\mathcal{B}d_i^-|$, the size of the current negative border. A simple remark can be made here: if this parameter is very large (more than 100000) the minimal transversals computation become prohibitive. We are not aware of existing implementations of minimal transversals

computation able to handle such input hypergraphs [1]. Moreover, such cases are likely to correspond to best scenario for $Apriori$.

- $|F_i|$, the number of frequent i-itemsets and $|\mathcal{B}d_i^-|$ have to be compared. Indeed, a small value of $|\mathcal{B}d_i^-|$ wrt $|F_i|$ is likely to give a successful dualization.
- $|F_i|$ and $|C_i|$, the number of candidates in level i, can also be compared. If $|F_i|/|C_i|$ is close to 1, we can suspect to be in a "dense" part of the search space, and thus the levelwise search should be stopped.

3.6 Practical aspects

3.6.1 Candidate generation from the current positive border

From [18], candidate generation of a levelwise algorithm for a problem representable as sets can be formulated using dualization: At the ith iteration, we have

$$C_{i+1} = Tr(\cup_{j\leq i}\overline{F_j}) - \cup_{j\leq i} C_j$$

It is shown in [18] that candidate itemsets of C_{i+1} are exactly of size $i+1$, which allows to improve candidate generation.

In the setting of this paper, we can see C_{i+1} as the set $\mathcal{B}d_{i+1}^- - \mathcal{B}d_i^-$, and thus we get:

$$C_{i+1} = Tr(\overline{\mathcal{B}d_i^+}) - \cup_{j\leq i} C_j$$

Here, the major difference with a pure levelwise approach is that $\mathcal{B}d_i^+$ may contain some elements of size greater than $i+1$.

One may question about the size of the largest elements of C_{i+1}: does there exist elements of size strictly greater than $i+1$? The answer is yes as shown in the following non trivial example.

Example 1

Let r be the binary relation over a schema $\mathbf{R} = \{A, B, C, D, E, F, G, H, I\}$ represented in Table 1. For a minsup equals to 1, the borders of frequent itemsets in r are $\mathcal{B}d^- = \{AE, BF, CG, DH, ABCDI\}$ and $\mathcal{B}d^+ = \{ABCHI, ABDGI, ABGHI, ACDFI, ACFHI, ADFGI, AFGHI, BCDEI, BCEHI, BDEGI, BEGHI, CDEFI, CEFHI, DEFGI, EFGHI, ABCD\}$.

After a levelwise pass until level two, the four NFI of size two have been discovered, i.e. $\mathcal{B}d_2^- = \{AE, BF, CG, DH\}$. Suppose the algorithm decides here to stop the pure levelwise search. Then, these sets are used to compute the optimistic positive border from level 2. It is made up of 16 itemsets of size 5, among which the only non frequent itemset is $ABCDI$. Thus, at this time, $\mathcal{B}d_2^+ = \{ABCHI, ABDGI, ABGHI, ACDFI, ACFHI, ADFGI, AFGHI, BCDEI, BCEHI, BDEGI, BEGHI, CDEFI, CEFHI, DEFGI, EFGHI\}$. We obtain $\mathcal{B}d^-(\mathcal{B}d_2^+) = \{ABCD\}$ of size 4, being understood that no elements of size 3 does exist.

A	B	C	D	E	F	G	H	I
1	1	1					1	1
1	1		1			1		1
1	1					1	1	1
1		1	1		1			1
1		1			1		1	1
1			1		1	1		1
1					1	1	1	1
	1	1	1	1				1
	1	1		1			1	1
	1		1	1		1		1
	1			1		1	1	1
		1	1	1	1			1
		1		1	1		1	1
			1	1	1	1		1
				1	1	1	1	1
1	1	1	1					

Table 1. Example database

In our first implementation, computing the set $\mathcal{B}d^-(\mathcal{B}d_i^+)$ using minimal transversals had a quite prohibitive cost on large hypergraph instances. Therefore, we made the choice to restrict $\mathcal{B}d^-(\mathcal{B}d_i^+)$ to its $(i+1)$-itemsets for *efficiency reasons*. This choice has no effect on the correctness of the algorithm, since the termination condition is always the same[2].

3.6.2 Dealing with "almost frequent" large candidate itemsets

Let us consider the case of a candidate itemset obtained after a dualization from the current negative border. Let X be this candidate. Two main cases do exist: either X is frequent, or X is infrequent. In that case, we propose to estimate a degree of error in order to "qualify the jump".

Given a new user-defined threshold δ, and a minimal support $minsup$, an *error measure*, noted

[1] Experiments conducted in [16, 2] only consider hypergraphs with not more than 32000 edges.

[2] We suspect the algorithm $PincerSearch$ [17] to be *not complete*. Indeed, the search strategy of $PincerSearch$ is very close to our proposition: if we only consider $(i+1)$-itemsets in $\mathcal{B}d^-(\mathcal{B}d_i^+)$, they correspond exactly to the candidate set C_{i+1} of $PincerSearch$. Since $PincerSearch$ stops as soon as $C_{i+1} = \emptyset$, some elements could be forgotten. From the example 1, after the level 2, C_3 is empty, and therefore the maximal set $ABCD$ seems to be never generated by $PincerSearch$.

$error(X, minsup)$, can be defined as the ratio between the minsup $minsup$ and the support of the infrequent itemset X, i.e. $error(X, minsup) = 1 - \frac{support(X)}{minsup}$.

Two sub-cases are worth considering:

- either $error(X, minsup) \leq \delta$: the "jump" was not successful but solutions should exist among the nearest subsets of X.
- or $error(X, minsup) > \delta$: In that case, the jump was over-optimistic and probably, no solution does exist among the nearest generalizations of X.

Note that this error measure is decreasing, i.e. $X \subset Y \Rightarrow error(X, minsup) \leq error(Y, minsup)$

In our current implementation, these almost frequent large itemsets are first considered as frequent to enable more pruning in subsequent passes. Afterward, they are considered at the very end of our algorithm. A pure top-down levelwise approach has been implemented to find out their subsets which can be maximal frequent itemsets.

4 Implementation details and experimental results

4.1 Implementation details

An implementation of the algorithm has been performed in C++/STL. Two existing implementations available from the FIMI repository website [10] were borrowed: the $Apriori$ code of C. Borgelt [5] and the prefix-tree implementation of B. Goethals using C++/STL [10].

To keep coherence with this implementation, we use a similar data structure for the new parts of the algorithm. The itemsets and the transactions are stored in a prefix-tree [1, 6].

Minimal Transversals computation For the minimal transversals computation, we implemented the algorithm proposed in [9] using a prefix-tree in order to handle relatively large hypergraph instances. Its incremental aspect is very interesting in our case, since the negative border is itself incremental. Note that improvements have been performed by exploiting the knowledge of previous dualizations. We do not give more details here.

4.2 Experimental results

We conducted experiments on a pentium 4.3GHz Processor, with 1Go of memory. The operating system was Redhat Linux 7.3 and we used gcc 2.96 for the compilation. We used four datasets available on the FIMI'03 repository.

We first evaluate the influence of the level from which the levelwise approach is stopped on the performances of ABS. Then, the impact of "almost frequent" large itemsets is studied for different threshold values for the error measure. Finally, we compare ABS with four maximal frequent itemsets mining algorithms: $Apriori$ and $Eclat$ [12] implemented by C.Borgelt [5], $Fpmax$ [13] based on $FP-trees$ [15] and IBE [21].

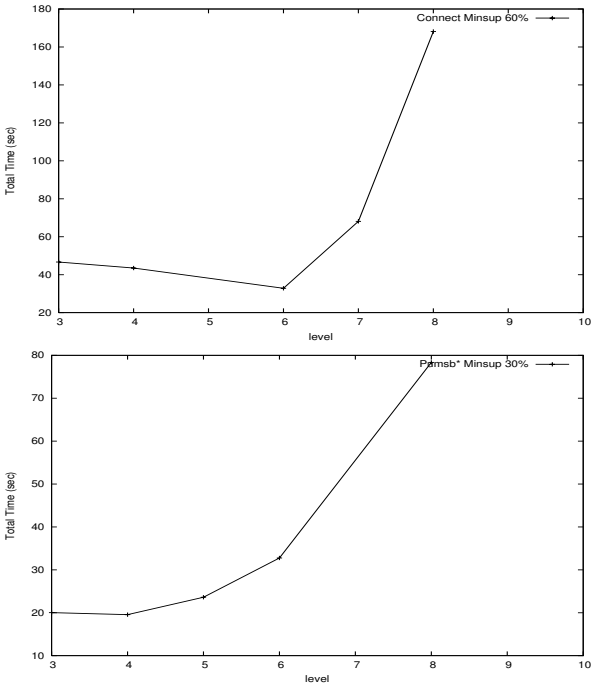


Figure 1. Forcing the first dualization at level k for connect (top) and pumsb* (bottom)

In figure 1, we forced the first dualization for different levels (from 3 to 8), on the connect dataset with a $minsup$ of 80 % and the pumsb* dataset with a $minsup$ of 30%. The results confirm the necessity to fix dynamically this parameter, and then justify an adaptive approach. Second, for all tested datasets, our function $IsDualizationRelevant$ has dynamically determined the best level to begin dualization.

The optimization based on the error measure is evaluated on figure 2. From pumsb dataset (on the top), this optimization appears to be interesting with a threshold value near 0.002. Nevertheless, on the connect dataset (bottom) no improvements is achieved. This comes from the fact that the proposed error measure is not strictly decreasing; and the equivalence classes induced by closed frequent itemsets are large. Our top down levelwise approach is prone to fail on this kind of databases .

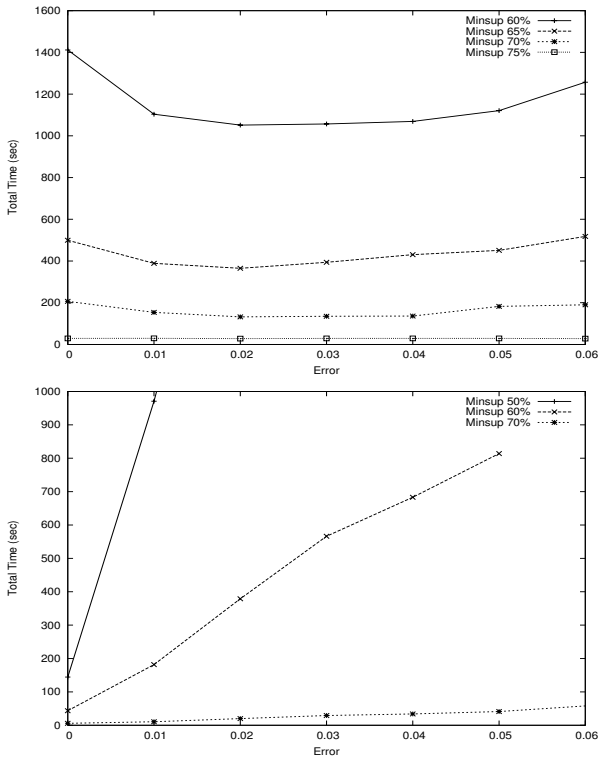


Figure 2. Exec. times for pumsb (top) and connect (down) wrt different error measure thresholds

From figures 3, 5 and 6, ABS is far to compete with best known implementations but tends to outperform IBE for most of our experimentations. Recall that IBE is the unique implementation based on the concept of dualization available from FIMI'03. We believe that this is due to the number of dualization performed by IBE, which is in the size of the positive border.

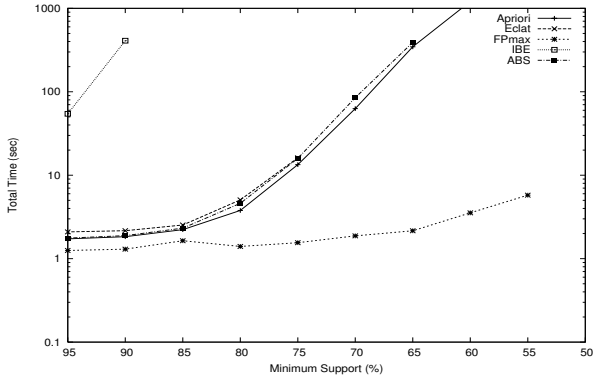


Figure 3. Execution times for database Pumsb

From figure 4, IBE exhibits better performances than ABS for low support thresholds (less than 20%). This is due to the fact that while the size of the positive border remains small (less than 5000 elements) the size of the negative border exceeds 10^6 elements, where some elements appear to have a very large size. This seems to be the worst case for ABS.

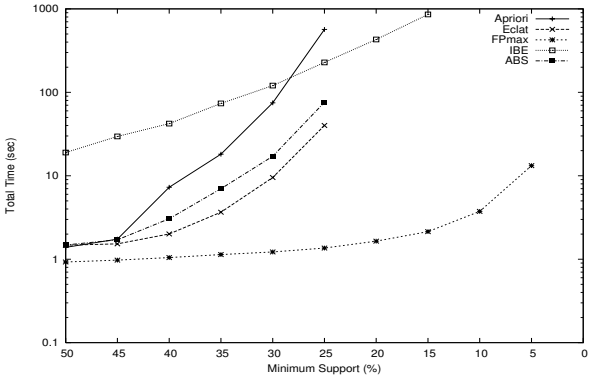


Figure 4. Execution times for database Pumsb*

From figure 5, ABS behaves like $Apriori$ as expected. Indeed, the positive border of retail is made up of "small" itemsets, and $Apriori$ turns out to be the best implementation for this kind of datasets.

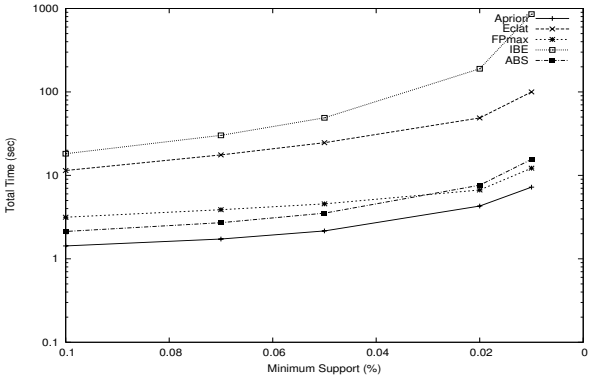


Figure 5. Execution times for database Retail

From figure 6, ABS is not as efficient as best known implementations (e.g. $fpmax$), but improves $Apriori$ by a factor of ten and beats $Eclat$ and IBE.

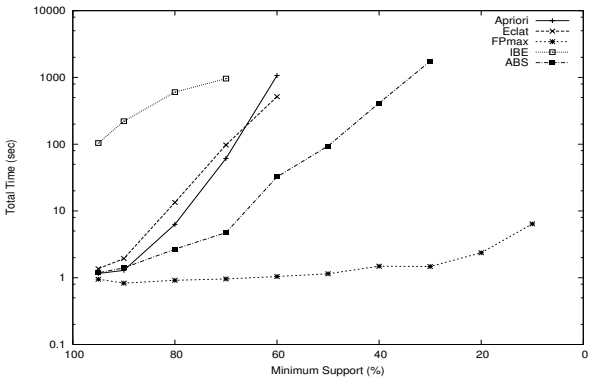


Figure 6. Execution times for database Connect

To sum up, two main reasons explain our mitigate results: 1) the cost of dualization remains high on very large hypergraph instances and 2) candidate generation and support counting seems to be not enough efficient in our current implementation.

The main parameter influencing performance of ABS turns out to be around the negative border. If for a given minsup, the negative border does not become too huge and its largest element remains "small" with respect to the largest maximal frequent itemset, ABS should have good performance.

5 Related works

Several algorithms exist for discovering maximal frequent itemsets mining in a transactional database (see FIMI'03). The goal is always to avoid an exponential search by characterizing as fast as possible largest frequent itemsets without exploring their subsets. $MaxMiner$ [3] uses a levelwise approach to explore the candidate itemsets, using the Rymon's enumeration system [20] - in which itemsets are arranged in a non redundant tree. But when a candidate X is counted over the database, the greatest candidate in the subtree of X is also counted; if it is frequent, then all the subtree can be pruned by anti-monotony of the "is frequent" property. Jumps done by $MaxMiner$ depend on the ordering of items used to build the tree and are therefore quiet different from jumps proposed in this paper. The algorithms $Mafia$ [7] and $GenMax$ [12] use the same principle as $MaxMiner$ with efficient optimizations, e.g. vertical bitmaps.

The $Pincer - Search$ Algorithm [17] uses a search strategy very close to ours. After a levelwise initialization, the principle is also to look at the largest not yet eliminated candidates. However, these large candidates are not characterized in a formal way.

In [14], the authors propose the *Dualize and Advance* algorithm. In their approach, the positive border in construction is always a subset of the positive border to be discovered. At each step, from some elements of the positive border already discovered, they generate the corresponding negative border. If one element of the negative border appears to be satisfied, they generate a specialization of it which belongs to the positive border and they re-iterate the process until each element of the negative border is indeed not satisfied. An implementation of a variant of *Dualize and Advance* has been proposed in [21] with an irredundant dualization. Their code is available from the FIMI'03 website.

Some algorithms like $Mafia$ [7] or DCI [19] can adapt themselves to mine frequent itemsets, with respect to the dataset density and some architectural characteristics (e.g. available memory). Even if these aspects improve performances, it only concerns choices for data structures; the mentioned algorithms do not really adapt their *strategy* to explore the search space.

6 Conclusion and future works

In this paper, we have proposed an ongoing effort toward the discovery of maximal frequent itemsets. Our contribution takes its roots from the algorithm ZigZag devised for inclusion dependency discovery in databases. Even if this two data mining problems fit into the same theoretical framework [18], they widely differ in practice which is not a surprise. Indeed, while ZigZag performed very well in our experiments[3], ABS does not exhibit such a good behavior for maximal frequent itemsets mining. Many reasons explain this result, for instance the availability of public datasets allowing thorough experimentations, the intrinsic properties of each problem, and may be the more important reason lies in the cost of a database access, in-memory resident data vs data stored into a DBMS.

Many improvements can be brought to our current implementation. Some are specific to our algorithm like for instance minimal transversal computation on large hypergraph instances or taking into account large equivalence classes induced by closed frequent itemsets during candidate generation from "almost frequent" itemsets. Some others belong to "the state of the art" of maximal frequent itemsets implementation : managing huge set of set, support counting... Complexity issues need also to be addressed.

To end up, we want to quote a personal note on the main objective of the FIMI workshop. We believe that frequent, closed and maximal itemsets mining are key data mining tasks since algorithms devised to solve these tasks are likely to be used in other contexts under some conditions [18]. Roughly speaking, for every problem *representable as sets*

[3]Note that dynamic parameters were quiet different, e.g. the first dualization was always performed at the second level.

with an anti-monotone predicate as for instance with functional dependency inference or simply anti-monotone predicates on itemsets other than "is frequent", the algorithms devised for FIMI should be useful to answer these tasks. Nevertheless, it seems rather optimistic to envision the application of many FIMI'03 [11] implementations to another data mining problem representable as sets. Indeed, even if the development of efficient data structures for managing huge sets of set is definitely useful, loading the database in main memory using sophisticated data structure specially devised for the anti-monotone predicate to be mined turns out to give very efficient algorithms but deserve other data mining tasks.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *International Conference on Very Large Data Bases (VLDB'94), Santiago de Chile, Chile*, pages 487–499. Morgan Kaufmann, 1994.

[2] J. Bailey, T. Manoukian, and K. Ramamohanarao. A fast algorithm for computing hypergraph transversals and its application in mining emerging patterns. In *International Conference on Data Mining (ICDM'03), Floride, USA*, pages 485–488, 2003.

[3] R. J. Bayardo. Efficiently mining long patterns from databases. In L. M. Haas and A. Tiwary, editors, *ACM SIGMOD Conference, Seattle, USA*, pages 85–93, 1998.

[4] C. Berge. *Graphs and Hypergraphs*. North Holland, Amsterdam, 1973.

[5] C. Borgelt. Efficient implementations of apriori and eclat. In *FIMI'03 Workshop on Frequent Itemset Mining Implementations*, November 2003.

[6] C. Borgelt and R. Kruse. Induction of association rules : Apriori implementation. 2002.

[7] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *International Conference on Data Engineering (ICDE'01), Heidelberg, Germany*, pages 443–452. IEEE CS, 2001.

[8] F. De Marchi and J.-M. Petit. Zigzag : a new algorithm for discovering large inclusion dependencies in relational databases. In *International Conference on Data Mining (ICDM'03), Melbourne, Florida, USA*, pages 27–34. IEEE Computer Society, 2003.

[9] J. Demetrovics and V. Thi. Some remarks on generating armstrong and inferring functional dependencies relation. *Acta Cybernetica*, 12(2):167–180, 1995.

[10] B. Goethals. Frequent itemset mining implementations repository, http://fimi.cs.helsinki.fi/, 2003.

[11] B. Goethals and M. Zaki, editors. *Workshop on Frequent Itemset Mining Implementations*. CEUR Workshop Proceedings, 2003.

[12] K. Gouda and M. J. Zaki. Efficiently mining of maximal frequent itemsets. In N. Cercone, T. Y. Lin, and X. Wu, editors, *International Conference on Data Mining (ICDM'01), San Jose, USA*. IEEE Computer Society, 2001.

[13] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI'03 Workshop on Frequent Itemset Mining Implementations*, November 2003.

[14] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. S. Sharma. Discovering all most specific sentences. *ACM Transaction on Database System*, 28(2):140–174, 2003.

[15] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD'00, Dallas, Texas, USA*, 2000.

[16] D. J. Kavvadias and E. C. Stavropoulos. Evaluation of an algorithm for the transversal hypergraph problem. In J. S. Vitter and C. D. Zaroliagis, editors, *Algorithm Engineering, International Workshop, WAE '99, London, UK*, volume 1668, 1999.

[17] D.-I. Lin and Z. M. Kedem. Pincer search: A new algorithm for discovering the maximum frequent set. In H.-J. Schek, F. Saltor, I. Ramos, and G. Alonso, editors, *Extending Database Technology (EDBT'98), Valencia, Spain*, volume 1377 of *Lecture Notes in Computer Science*, pages 105–119. Springer, 1998.

[18] H. Mannila and H. Toivonen. Levelwise Search and Borders of Theories in Knowledge Discovery. *Data Mining and Knowledge Discovery*, 1(1):241–258, 1997.

[19] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets. In *International Conference on Data Mining (ICDM'02), Maebashi City, Japan*, pages 338–345, 2002.

[20] R. Rymon. Search through systematic set enumeration. In B. Nebel, C. Rich, and W. R. Swartout, editors, *International Conference on Principles of Knowledge Representation and Reasoning (KR'92), Cambridge, USA*, pages 539–550. Morgan Kaufmann, 1992.

[21] T. Uno and K. Satoh. Detailed description of an algorithm for enumeration of maximal frequent sets with irredundant dualization. In B. Goethals and M. Zaki, editors, *ICDM 2003 Workshop on Frequent Itemset Mining Implementations (FIMI '03), Melbourne, Florida, USA*, volume 90 of *CEUR Workshop Proceedings*, 2003.

[22] M.-J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *International Conference on Knowledge Discovery and Data Mining (KDD-97), Newport Beach, California, USA*, pages 283–286. AAAI Press, 1997.

DCI_Closed: a Fast and Memory Efficient Algorithm to Mine Frequent Closed Itemsets

Claudio Lucchese
ISTI "A. Faedo"
Consiglio Nazionale delle Ricerche (CNR)
Pisa, Italy.
email: claudio.lucchese@isti.cnr.it

Salvatore Orlando
Computer Science Dept.
Università Ca' Foscari
Venezia, Italy.
email: orlando@dsi.unive.it

Raffaele Perego
ISTI "A. Faedo"
Consiglio Nazionale delle Ricerche (CNR)
Pisa, Italy.
email: raffaere.perego@isti.cnr.it

Abstract

One of the main problems raising up in the frequent closed itemsets mining problem is the duplicate detection. In this paper we propose a general technique for promptly detecting and discarding duplicate closed itemsets, without the need of keeping in the main memory the whole set of closed patterns.

Our approach can be exploited with substantial performance benefits by any algorithm that adopts a vertical representation of the dataset. We implemented our technique within a new depth-first closed itemsets mining algorithm. The experimental evaluation demonstrates that our algorithm outperforms other state of the art algorithms like CLOSET+ and FPCLOSE.

1. Introduction

Frequent itemsets mining is the most important and demanding task in many data mining applications. To describe the mining problem we introduce the following notation. Let $\mathcal{I} = \{a_1, ..., a_M\}$ be a finite set of items, and $\mathcal{D}$ a finite set of transactions (the dataset), where each transaction $t \in \mathcal{D}$ is a list of *distinct* items $t = \{i_0, i_1, ..., i_T\}$, $i_j \in \mathcal{I}$. A k-itemset is a sequence of k *distinct* items $I = \{i_0, i_1, ..., i_k\} \mid i_j \in \mathcal{I}$, sorted on the basis of some total order relation between item literals. The number of transactions in the dataset including an itemset I is defined as the *support* of I (or *supp*(I)). Mining all the frequent itemsets from $\mathcal{D}$ requires to discover all the itemsets having support higher than (or equal to) a given threshold *min_supp*.

The paper is organized as follows. In Sect. 2 we introduce closed itemsets and describe a framework for mining them. This framework is shared by all the algorithms surveyed in Sect. 3. In Sect. 4 we formalize the problem of duplicates and propose our technique. Section 5 proposes an implementation of our technique and discusses the experimental results obtained. Follow some concluding remarks.

2. Closed itemsets

The concept of closed itemset is based on the two following functions f and g:

$$f(T) = \{i \in \mathcal{I} \mid \forall t \in T, i \in t\}$$
$$g(I) = \{t \in \mathcal{D} \mid \forall i \in I, i \in t\}$$

where T and I, $T \subseteq \mathcal{D}$ and $I \subseteq \mathcal{I}$ are, respectively, subsets of all the transactions and items occurring in dataset $\mathcal{D}$. Function f returns the set of itemsets included in all the transactions in T, while function g returns the set of transactions supporting a given itemset I. Since the set of transaction $g(I)$ can be represented by a list of *transaction identifiers*, we refer to $g(I)$ as the *tid-list* of I. We can introduce the following definition:

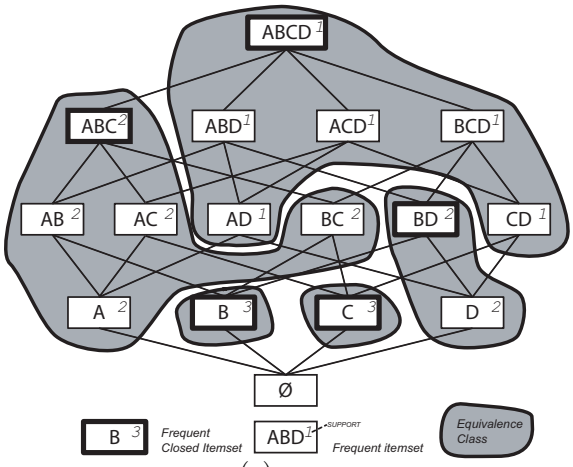
Definition 1 *An itemset I is said to be* closed *if and only if*

$$c(I) = f(g(I)) = f \circ g(I) = I$$

where the composite function $f \circ g$ is called Galois operator or closure operator.

The closure operator defines a set of equivalence classes over the lattice of frequent itemsets: two itemsets belong to the same equivalence class *iff* they have the same closure, i.e. they are supported by the same set of transactions. We can also show that an itemset I is closed if no superset of I with the same support exists. Thus, a closed itemset is also the maximal itemset of an equivalence class. Mining all these maximal elements of each equivalence class corresponds to mine all the closed itemsets.

Figure 1.(a) shows the lattice of frequent itemsets derived from the simple dataset reported in Fig. 1.(b), mined with $min\_supp = 1$. We can see that the itemsets with the same closure are grouped in the same equivalence class. Each equivalence class contains elements sharing the same supporting transactions, and closed itemsets are their maximal elements. Note that the number of closed itemsets (five) is remarkably lower than the number of frequent itemsets (fifteen).



(a)

TID	items			
1	B	D		
2	A	B	C	D
3	A	B	C	
4	C			

(b)

Figure 1. (a) Lattice of frequent itemsets with closed itemsets and equivalence classes given by the dataset (b).

The algorithms for mining frequent closed itemsets adopt a strategy based on two main steps: *Search space browsing*, and *Closure calculation*. In fact, they *browse* the search space by traversing the lattice of frequent itemsets from an equivalence class to another, and they calculate the *closure* of the frequent itemsets visited in order to determine the maximal elements (closed itemsets) of the corresponding equivalence classes. Let us analyze in some depth these two phases.

Browsing the search space. The goal of an effective browsing strategy should be to devise a spanning tree over the lattice of frequent itemsets, visiting exactly a single itemset in each equivalence class. We could in fact mine all the closed itemsets by calculating the closure of just an itemset per equivalence class. Let us call the itemsets used to compute closures during the visit *closure generators*.

Some algorithms choose the minimal elements (or *key patterns*) of each equivalence class as closure generators. Key patterns form a lattice, and this lattice can be easily traversed with a simple apriori-like algorithm. Unfortunately, an equivalence class can have more than one minimal element leading to the same closed itemset. For example, the closed itemset $\{ABCD\}$ of Fig. 1 may be mined twice, since it can be obtained as closure of the two minimal elements of its equivalence class $\{AD\}$ and $\{CD\}$.

Other algorithms use instead a different technique that we call *closure climbing*. As soon as a generator is devised, its closure is computed, and new generators are built as supersets of the closed itemset discovered. Since closed itemsets are maximal elements, this strategy always guarantees to jump from an equivalence class to another. Unfortunately, it does not guarantee that the new generators belong to equivalence classes that were not previously visited.

Regardless of the strategy adopted, some kind of duplicate check has thus to be introduced. A naive approach to check for duplicates is to search for each generated closed itemset among all the ones mined so far. Indeed, in order to avoid to perform a lot of expensive closure operations, several algorithms exploit the following lemma:

Lemma 1 *Given two itemsets X and Y, if $X \subset Y$ and $supp(X) = supp(Y)$ (i.e. $|g(X)| = |g(Y)|$), then $c(X) = c(Y)$.*

Proof. If $X \subset Y$, then $g(Y) \subseteq g(X)$. Since $|g(Y)| = |g(X)|$ then $g(Y) = g(X)$. $g(X) = g(Y) \Rightarrow f(g(X)) = f(g(Y)) \Rightarrow c(X) = c(Y)$.
□

Therefore, given a generator X, if we find an already mined closed itemsets Y that set-includes X,

and the supports of Y and X are identical, we can conclude that $c(X) = c(Y)$. Hence we can prune the generator X without actually calculating its closure. Also this duplicates checking strategy is however expensive, both in time and space. In time because we may need to search for the inclusion of each generator in a huge number of closed itemsets, in space because to perform it we need to keep all the closed itemsets in the main memory. To reduce such costs, closed sets can be stored in compact prefix tree structures, indexed by one or more levels of hashing.

Calculating Closures. To calculate the closure of an itemset X, we have to apply the Galois operator c. Applying c requires to intersect all the transactions of the dataset including X. Another way to calculate the closure is given by the following lemma:

Lemma 2 *Given an itemset X and an item i, if $g(X) \subseteq g(i) \Rightarrow i \in c(X)$.*

Proof. Since $g(X \cup i) = g(X) \cap g(i)$, $g(X) \subseteq g(i) \Rightarrow g(X \cup i) = g(X)$. Therefore, if $g(X \cup i) = g(X)$ then $f(g(X \cup i)) = f(g(X)) \Rightarrow c(X \cup i) = c(X) \Rightarrow i \in c(X)$. □

From the above lemma, we know that if $g(X) \subseteq g(i)$, then $i \in c(X)$. Therefore, by performing this inclusion check for all the items in $\mathcal{I}$ not included in X, we can *incrementally* compute $c(X)$. Note that, since the set $g(i)$ can be represented by the *tid-list* associated with i, this suggests the adoption of a vertical format for the input dataset in order to efficiently implement the inclusion check: $g(X) \subseteq g(i)$.

The closure calculation can be performed off-line or on-line. In the first case we firstly retrieve the complete set of generators, and then we calculate their closures. In the second case, as soon as a new generator is discovered, its closure is computed on-the-fly.

The algorithms that compute closures on-line are generally more efficient. This is because they can adopt the *closure climbing* strategy, according to which new generators are created recursively from closed itemsets. These generators are likely longer than key patterns, which are the minimal itemsets of the equivalence class and thus are the shorter possible generators. Obviously, the longer the generator is, the fewer checks (on further items to add) are needed to get its closure.

3. Related Works

The first algorithm proposed for mining closed itemsets was A-CLOSE [5] (N. Pasquier, et al.). A-CLOSE first browses level-wise the frequent itemsets lattice by means of an Apriori-like strategy, and mines all the minimal elements of each equivalence class. Since a k-itemset is a key pattern if and only if no one of its $(k-1)$-subsets has the same support, minimal elements are discovered with an intensive subset checking. In its second step, A-CLOSE calculates the closure of all the minimal generators previously found. Since a single equivalence class may have more than one minimal itemsets, redundant closures may be computed. A-CLOSE performance suffers from the high cost of the off-line closure calculation and the huge number of subset searches.

The authors of FP-Growth [2] (J. Han, et al.) proposed CLOSET [6] and CLOSET+ [7]. These two algorithms inherit from FP-Growth the compact FP-Tree data structure and the exploration technique based on recursive conditional projections of the FP-Tree. Frequent single items are detected after a first scan of the dataset, and with another scan the pruned transactions are inserted in the FP-Tree stored in the main memory. With a depth first browsing of the FP-Tree and recursive conditional FP-Tree projections, CLOSET mines closed itemsets by closure climbing, and growing up frequent closed itemsets with items having the same support in the conditional dataset. Duplicates are discovered with subset checking by exploiting Lemma 2. Thus, all closed sets previously discovered are kept in the main memory, and are indexed by a two level hash. CLOSET+ is similar to CLOSET, but exploits an adaptive behaviour in order to fit both sparse and dense datasets. As regards the duplicate detection technique, CLOSET+ introduces a new one for sparse datasets named *upward checking*. This technique consists in the intersection of every path of the initial FP-Tree leading to a candidate closed itemset X, if such intersection is empty then X is actually closed. The rationale for using it only in sparse dataset is that the transactions are short, a thus the intersections can be performed quickly. Note that with dense dataset, where the transactions are usually longer, closed itemsets equivalence classes are large and the number of duplicates is high, such technique is not used because of its inefficiency, and CLOSET+ steps back using the same strategy of CLOSET, i.e. storing every mined closed itemset.

FPCLOSE [1], which is a variant of CLOSET+, resulted to be the best algorithm for closed itemsets mining presented at the ICDM 2003 Frequent Itemset Mining Implementations Workshop.

CHARM [9] (M. Zaki, et al.) performs a bottom-up depth-first browsing of a prefix tree of frequent itemsets built incrementally. As soon as a frequent itemset is generated, its tid-list is compared with those of the other itemsets having the same parent. If one tid-list includes another one, the associated nodes are merged

since both the itemsets surely belong to the same equivalence class. Itemset tid-lists are stored in each node of the tree by using the diff-set technique [8]. Since different paths can however lead to the same closed itemset, also in this case a duplicates pruning strategy is implemented. CHARM adopts a technique similar to that of CLOSET, by storing in the main memory the closed itemsets indexed by single level hash.

According to our classification, A-CLOSE exploits a key pattern browsing strategy and performs off-line closure calculations, while CHARM, CLOSET+ and FP-CLOSE are different implementations of the same closure climbing strategy with incremental closure computation.

4. Removing duplicate generators of closed itemsets

In this Section we discuss a particular visit of the lattice of frequent sets used by our algorithm to identify *unique generators* of each equivalence class, and compute all the closed patterns through the minimum number of closure calculations.

In our algorithm, we use *closure climbing* to browse the search space, find generators and compute their closure. As soon as a generator is found, its closure is computed, and new generators are built as supersets of the closed itemset discovered so far. So, each generator *gen* browsed by our algorithm can be generally represented as $gen = Y \cup i$, where Y is a closed itemset, and i, $i \notin Y$ is an item in $\mathcal{I}$[1].

Looking at Figure 1.(a), we can unfortunately discover multiple generators $gen = Y \cup i$, whose closures produce an identical closed itemset. For example, we have four generators, $\{A\}$, $\{A, B\}$, $\{A, C\}$ and $\{B, C\}$, whose closure is equal to the closed itemsets $\{A, B, C\}$. Note that all these generators have the form $Y \cup i$, since they can be obtained by adding a single items to a smaller closed itemset, namely $\emptyset$, $\{B\}$ and $\{C\}$.

The technique exploited by our algorithm to detect duplicate generators exploits a *total lexicographic order* relation $\prec$ between all the itemsets of our search space[2]. Since there exist a relation $\prec$ between each pair of k-itemsets, in order to avoid duplicate closed itemsets, we do not compute the closure of the generators that do not result *order preserving* according to the definition below.

Definition 2 *A generator $X = Y \cup i$, where Y is a closed itemset and $i \notin Y$, is said to be* order preserving *one iff $i \prec (c(X) \setminus X)$.*

The following Theorem shows that, for any closed itemset Y, it is possible to find a sequence of order preserving generators in order to climb a sequence of closure itemsets and arrive at Y. The following Corollary states that this sequence is unique.

Theorem 1 *For each closed itemset $Y \neq c(\emptyset)$, there exists a sequence of n ($n \geq 1$) items $i_0 \prec i_1 \prec ... \prec i_{n-1}$ such that*

$$\{gen_0, gen_1, \ldots, gen_{n-1}\} = \{Y_0 \cup i_0, Y_1 \cup i_1, \ldots, Y_{n-1} \cup i_{n-1}\}$$

where the various gen_i are order preserving generators*, with $Y_0 = c(\emptyset)$, $Y_{j+1} = c(Y_j \cup i_j)$ $\forall j \in [0, n-1]$ and $Y = Y_n$.*

Proof. First of all, we show that given a generic generator $gen \subseteq Y$, $c(gen) \subseteq Y$. More formally, if $\exists Y'$ such that Y' is a closed itemset, and $Y' \subset Y$, and we extend Y' with an item $i \in Y \setminus Y'$ to obtain $gen = Y' \cup i \subseteq Y$, then $\forall j \in c(gen)$, $j \in Y$.

Note that $g(Y) \subseteq g(gen)$ because $gen \subseteq Y$. Moreover, if $j \in c(gen)$, then $g(c(gen)) \subseteq g(j)$. Thus, since $g(Y) \subseteq g(gen)$, then $g(Y) \subseteq g(j)$ also holds, so that $j \in c(Y)$ too. So, if $j \notin Y$ hold, Y would not be closed, and this is in contradiction with the hypothesis.

As regards the proof of the Theorem, we show it by constructing a sequence of closed itemsets and associated generators having the properties stated above.

We have that $Y_0 = c(\emptyset)$. All the items in Y_0 appear in every transaction of the dataset and therefore by definition of closure they must be included also in Y, i.e. $Y_0 \subseteq Y$.

Since $Y_0 \neq Y$ by definition, we choose $i_0 = \min_{\prec}(Y \setminus Y_0)$, i.e. i_0 is the smallest item in $\{Y \setminus Y_0\}$ with respect to the lexicographic order $\prec$, in order to create the first order preserving generator $\{Y_0 \cup i_0\}$. Afterwards we calculate $Y_1 = c(Y_0 \cup i_0) = c(gen_0)$.

Once Y_1 is found, if $Y_1 = Y$ we stop.

Otherwise we choose $i_1 = \min_{\prec}(Y \setminus Y_1)$, where $i_0 \prec i_1$ by construction, in order to build the next *order preserving* generator $gen_1 = Y_1 \cup i_1$ and we calculate $Y_2 = c(Y_1 \cup i_1) = c(gen_1)$.

Once Y_2 is found, if $Y_2 = Y$ we stop, otherwise we iterate, by choosing $i_2 = \min_{\prec}(Y \setminus Y_2)$, and so on.

Note that each generator $gen_j = \{Y_j \cup i_j\}$ is *order preserving*, because $c(\{Y_j \cup i_j\}) = Y_{j+1} \subseteq Y$ and i_j is

---

1 For each closed itemset $Y' \neq c(\emptyset)$, it is straightforward to show that there must exists at least a generator having the form $gen = Y \cup i$, where $Y, Y \subset Y'$, is a closed itemset, $i \notin Y$, and $Y' = c(gen)$.

2 This lexicographic order is induced by an order relation between single item literals, according to which each k-itemset I can be considered as a *sorted set* of k *distinct* items $\{i_0, i_1, ..., i_k\}$.

the minimum item in $\{Y \setminus Y_j\}$ by construction, i.e. $i_j \prec \{Y_{j+1} \setminus \{Y_j \cup i_j\}\}$.
□

Corollary 1 *For each closed itemset $Y \neq c(\emptyset)$, the sequence of* order preserving generators $\{gen_0, gen_1, \ldots, gen_n\} = \{Y_0 \cup i_0, Y_1 \cup i_1, \ldots, Y_n \cup i_n\}$ *as stated in Theorem 1 is unique.*

Proof. Since all the items in Y_0 appear in every transaction of the dataset, by definition of closure, they must be included also in Y, we have that $Y_0 = c(\emptyset)$.

During the construction of the sequence of generators, suppose that we choose $i_j \neq \min_{\prec}(Y \setminus Y_j)$ to construct generator gen_j. Since gen_j and all the following generators must be *order preserving*, it should be impossible to obtain Y, since we can not consider anymore the item $i = \min_{\prec}(Y \setminus Y_j) \in Y$ in any other generator or closure in order to respect the *order preserving* property.
□

Looking at Figure 1.(a), for each closed itemset we can easily identify those unique sequences of order preserving generators. For example, for the the closed itemset $Y = \{A, B, C, D\}$, we have $Y_0 = c(\emptyset) = \emptyset$, $gen_0 = \emptyset \cup A$, $Y_1 = c(gen_0) = \{A, B, C\}$, $gen_1 = \{A, B, C\} \cup D$, and, finally, $Y = c(gen_1)$. Another example regards the closed itemset $Y = \{B, D\}$, where we have $Y_0 = c(\emptyset) = \emptyset$, $gen_0 = \emptyset \cup B$, $Y_1 = c(gen_0) = B$, $gen_1 = B \cup D$, and, finally, $Y = c(gen_1)$.

In order to exploit the results of Theorem 1, we need a fast way to check whether a generator is order preserving.

Lemma 3 *Let $gen = Y \cup i$ be a generator of a closed itemset where Y is a closed itemset and $i \notin Y$, and let pre-set$(gen) = \{j \prec i \mid j \notin gen\}$. gen is* not order preserving*, iff $\exists j \in$ pre-set(gen), such that $g(gen) \subseteq g(j)$.*

Proof. If $g(gen) \subseteq g(j)$, then $j \in c(gen)$. Since, by hypothesis, $j \prec i$, it is not true that $i \prec (c(gen) \setminus gen)$ because $j \in (c(gen) \setminus gen)$.
□

The previous Lemma introduces the concept of *pre-set*(gen), where $gen = \{Y \cup i\}$ is a generator, and gives a way to check the order preserving property of gen by scanning all the $g(j)$, for all $j \in$ *pre-set*(gen).

We have thus contributed a deep study on the the problem of duplicates in mining frequent closed itemsets. By reformulating the duplicates problem as the problem of visiting the lattice of frequent itemsets, according to a total (lexicographic) order, we have moved the dependencies of the order preserving check from the set of closed itemsets already mined to the *tid-lists* associated with single items. This new technique is not resource demanding, because frequent closed itemsets need not to be stored in the main memory during the computation, and it is not time demanding, because the order preserving check is cheaper than searching the set of closed itemsets mined so far. Note that CLOSET+ needs the initial FP-tree as an additional requirement the current FP-tree in use, and morover does not use its upward checking tchnique with dense datasets.

5. The *DCI_Closed* algorithm.

The pseudo-code of the recursive procedure *DCI_Closed*() is shown in Algorithm 1. The procedure receives three parameters: a closed itemsets CLOSED_SET, and two sets of items, i.e. the PRE_SET and POST_SET.

The procedure will output all the *non-duplicate closed itemsets* that properly contain CLOSED_SET. In particular, the goal of the procedure is to deeply explore each *valid new generator* obtained from CLOSED_SET by extending it with all the element in POST_SET.

Before calling procedure *DCI_Closed*(), the dataset $\mathcal{D}$ is scanned to determine the frequent single items $\mathcal{F}_1 \subseteq \mathcal{I}$, and to build the bitwise vertical dataset $\mathcal{VD}$ containing the various *tid-lists* $g(i)$, $\forall i \in \mathcal{F}_1$. The procedure is thus called by passing as arguments CLOSED_SET = $c(\emptyset)$, PRE_SET = $\emptyset$, and POST_SET = $\mathcal{F}_1 \setminus c(\emptyset)$. Note that the itemset $c(\emptyset)$ contains, if any, the items that occur in all the transactions of the dataset $\mathcal{D}$.

The procedure builds all the possible *generators*, by extending CLOSED_SET with the various items in POST_SET (lines 2–3). The *infrequent* and *duplicate* generators (i.e., the *not order preserving* ones) are however discarded as *invalid* (lines 4-5). Note that the items $i \in POST\_SET$ used to obtain those invalid generators will no longer be considered in the following recursive calls. Only the valid generators are then extended to compute their closure (lines 6–15). It is worth noting that each generator *new_gen* ← CLOSED_SET $\cup\, i$ is strictly extended according to the order preserving property, i.e. by using all items $j \in$ POST_SET such that $i \prec j$ (line 8). Note that all the items j, $i \prec j$, which do not belong to $c(new\_gen)$ are included in the new POST_SET (line 12) and are used for the next recursive call. At the end of this process, a new closed set (CLOSED_SET$_{New}$ ← $c(new\_gen)$) is obtained (line 15). From this new closed set, new generators and corresponding closed sets can be build, by recursively calling the procedure *DCI_Closed*() (line 16). Finally, it

Algorithm 1 DCI-closed pseudocode

```
1: procedure DCI_Closed(CLOSED_SET, PRE_SET, POST_SET)
2:     for all i ∈ POST_SET do                                  ▷ Try to create a new generator
3:         new_gen ← CLOSED_SET ∪ i
4:         if supp(new_gen) ≥ min_supp then                     ▷ new_gen is frequent
5:             if is_dup(new_gen, PRE_SET) = FALSE then         ▷ Duplication check
6:                 CLOSED_SET_New ← new_gen
7:                 POST_SET_New ← ∅
8:                 for all j ∈ POST_SET, i ≺ j  do              ▷ Compute closure of new_gen
9:                     if g(new_gen) ⊆ g(j) then
10:                        CLOSED_SET_New ← CLOSED_SET_New ∪ j
11:                    else
12:                        POST_SET_New ← POST_SET_New ∪ j
13:                    end if
14:                end for
15:                Write out CLOSED_SET_New and its support
16:                DCI_Closed(CLOSED_SET_New, PRE_SET, POST_SET_New)
17:                PRE_SET ← PRE_SET ∪ i
18:            end if
19:        end if
20:    end for
21: end procedure
22:
23:
24: function is_dup(new_gen, PRE_SET)
25:    for all j ∈ PRE_SET do                                   ▷ Duplicate check
26:        if g(new_gen) ⊆ g(j) then
27:            return FALSE                                     ▷ new_gen is not order preserving!!
28:        end if
29:    end for
30:    return TRUE
31: end function
```

is worth to point out that, in order to force the lexicographic order of the visit, the two **for all**'s (line 2 and line 8) have to extract items from POST_SET while respecting this order.

Before recursively calling the procedure, it is necessary to prepare the suitable PRE_SET and POST_SET to be passed to the new recursive call of the procedure. Upon each recursive call to the procedure, the size of the new POST_SET is monotonically decreased, while the new PRE_SET's size is instead increased.

As regards the composition of the new POST_SET, assume that the closed set X =CLOSED_SET$_{new}$ passed to the procedure (line 16) has been obtained by computing the closure of a generator $new\_gen = Y \cup i$ ($c(new\_gen)$), where Y =CLOSED_SET and $i \in$ POST_SET. The POST_SET$_{new}$ to be passed to the recursive call of the procedure is built as the set of all the items that follow i in the lexicographic order and that have not been already included in X. More formally, POST_SET$_{new} = \{j \in F_1 \mid i \prec j \text{ and } j \notin X\}$. This condition allows the recursive call of the procedure to only build new generators $X \cup j$, where $i \prec j$ (according to the hypotheses of Theorem 1.

The composition of the new PRE_SET depends instead on the *valid* generators[3] that precedes $new\_gen = Y \cup i$ in the lexicographic order. If all the generators were valid, it would simply be composed of all the items j that precede i in the lexicographic order, and $j \notin X = c(new\_gen)$. In other words, the new PRE_SET would be the complement set of $X \cup$ POST_SET$_{new}$.

While the composition of POST_SET guarantees that the various generators will be produced according to the lexicographic order $\prec$, the composition of PRE_SET guarantees that duplicate generators will be pruned by function *is_dup*().

Since we have shown that for each closed itemset Y exists one and only one sequence of *order preserving* generators and since our algorithm clearly explores every possible *order preserving* generator from every

3 The ones that have passed the frequency and duplicate tests.

closed itemset, we have that the algorithm is complete and does not produce any duplicate.

5.0.1. Some optimizations exploited in the algorithm. We adopted a large amount of optimizations to reduce the cost of the bitwise intersections, needed for the duplication and closure computations (line 10 and 34). For the sake of simplicity, these optimizations are not reported in the pseudo-code shown in Algorithm 1.

DCI-CLOSED inherits the internal representation of our previous works DCI[4] and kDCI[3].The dataset is stored in the main memory using a vertical bitmap representation. With two successive scans of the dataset, a bitmap matrix $D_{M\times N}$ is stored in the main memory. The $D(i,j)$ bit is set to 1 if and only if the j-th transaction contains the i-th frequent single item. Row i of the matrix thus represent the tid-list of item i.

The columns of D are then reordered to profit of data correlation. This is possible and highly worthwhile when we mine dense datasets. As in [3][4], columns are reordered to create a submatrix E of D having all its rows identical. Every operation (e.g. intersection ones) involving rows in the submatrix E will be performed only once, thus gaining strong performance improvements.

This kind of representation fits with our framework, because the three main operations, i.e. support count, closure, and duplicates check, can be fastly performed with cheap bit-wise AND/OR operation.

Besides the DCI optimizations, specifically tailored for sparse and dense datasets, we exploited more specific techniques made possible by the depth-first visit of the lattice of itemsets.

In order to determine that the itemset X is closed, the tidlist $g(X)$ must have been compared with all the $g(j)$'s, for all items j contained in the *pre-list* (*post-list*) of X, i.e. the items that precede (follows) all items included in X according to a lexicographic order. The PRE_SET must have been accessed for *checking duplicate generators*, and the POST_SET for *computing the closure*. In particular, for all $j \in$ PRE_SET $\cup$ POST_SET, we already know that $g(X) \not\subseteq g(j)$, otherwise those items j must have been included in X.

Therefore, since $g(X)$ must have already been compared with all the $g(j)$, for all items j contained in the PRE_SET (POST_SET) of X, we may save some important information regarding each comparison between $g(j)$ and $g(X)$. Such information will be used to reduce the cost of the following use of $g(j)$, when these tidlists $g(j)$ will have to be exploited to look for further closed itemsets that include/extend X. In particular, even if, for all j, it is true that $g(X) \not\subseteq g(j)$, we may know that some large portions of the bitwise tidlists $g(X)$ are however strictly included in $g(j)$. Let $\overline{g(X)}_j$ be the portion of the bitwise tidlist $g(X)$ strictly included in the corresponding portion of $g(j)$, namely $\overline{g(j)}$. Hence, since $\overline{g(X)}_j \subseteq \overline{g(j)}$, it is straightforward to show that $\overline{g(X \cup Y)}_j \subseteq \overline{g(j)}$ continues to hold, for all itemset Y used to extend X, because $g(X \cup Y) \subseteq g(X)$ holds . So, when we extend X to obtain a new generator, we can limit the inclusion check of the various $g(j)$ to the complementary portions of tid-lists $\overline{g(j)}$, thus strongly reducing the cost of them.

5.0.2. Dealing with sparse datasets. It is possible to show that in sparse datasets the number of closed itemsets is nearly equal to the number of frequent ones, so near that they are often the same. This means that the techniques for mining closed itemsets are of no use, because almost every duplicate checking or closure calculating procedure is likely to fail.

For this reason, in case of sparse datasets, we preferred to exploit our frequent itemset mining algorithm [3] with an additional closedness test over the frequent itemset discovered. Given a new frequent itemset X, every of it subset of length $|X|-1$ with the same support as X is marked as non closed. Experiments showed that this approach is fruitful (see Fig. 2.b).

5.0.3. Space complexity. For all the algorithms requiring to keep in the main memory the whole set of closed itemsets to perform the duplicate check, the size of the output is actually a lower bound on their space complexity. Conversely, we will show that the amount of memory required by an implementation based on our duplicate discarding technique is independent of the size of the output. To some extent, its memory occupation depends on those data structures that also need to be maintained in memory by other algorithms that visit depth-first the lattice and exploit tid-list intersections adopting a vertical datasets.

The main information needed to be kept in the main memory is the tid-list of each node in the current path along the lattice, and the tid-list of every frequent single item. In this way we are able to browse the search space intersecting nodes with single item tid-lists, and to discard duplicates checking the order preserving property.

The worst case of memory occupation happens when the number of generators and frequent single items is maximal: this occurs when $c(\emptyset) = \emptyset$ and every itemset is frequent and closed. If N is the number of frequent single items, the deepest path has N nodes, and since one of this node is a single item, the total number of tid-lists to be kept in the main memory is $2N-1$. Since the length of a tid-list is equal to the number of

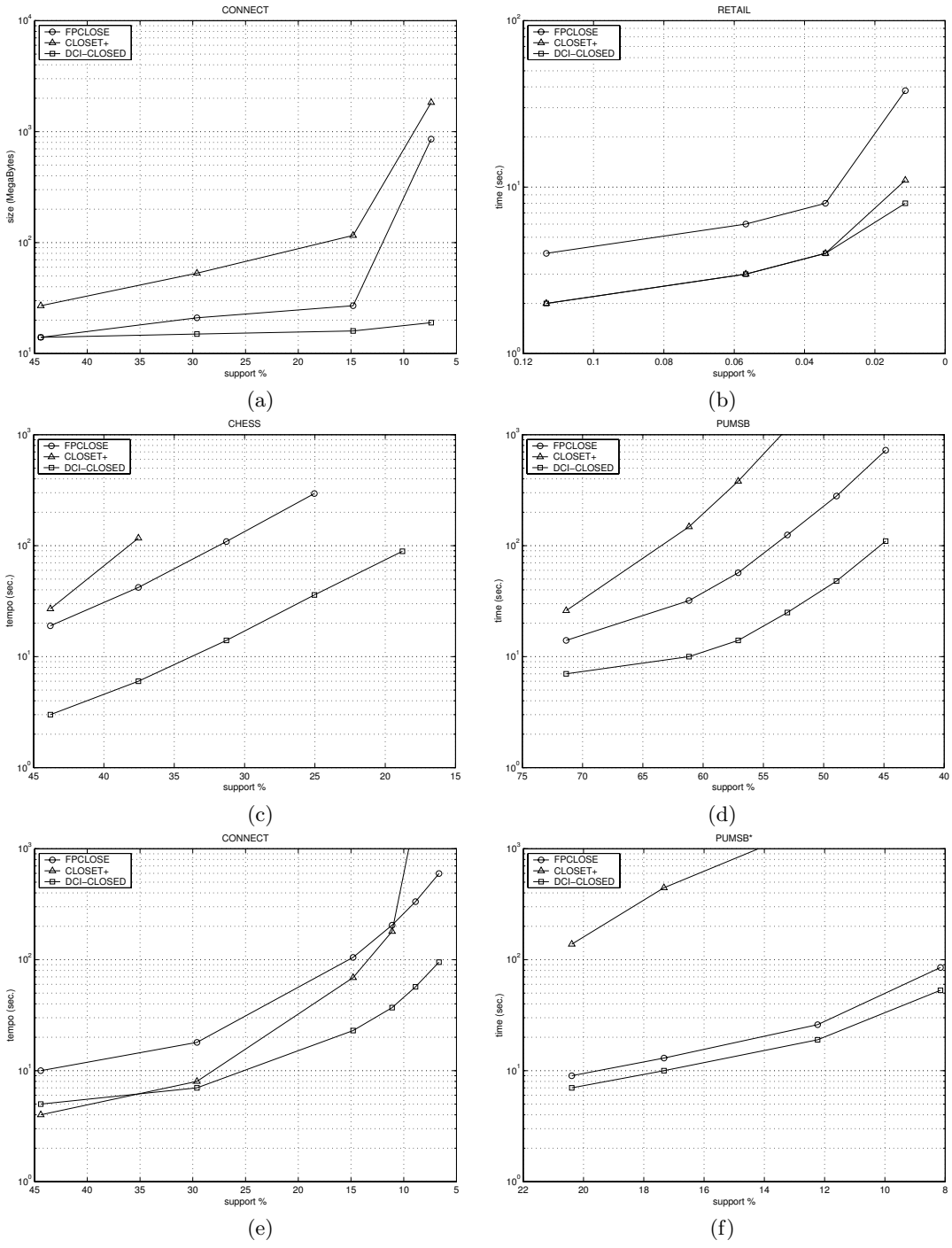


Figure 2. (a) Memory occupation on the connect dataset as a function of the minimum support threshold. (b-f) Execution times of FPCLOSE, CLOSET+, and DCI-CLOSET as a function of the minimum support threshold on various publicly available datasets.

transactions T in the dataset, the space complexity of our algorithm is

$$O\left((2N-1)\times T\right).$$

Figure 2.(a) plots memory occupation of FPCLOSE, CLOSET+ and our algorithm DCI-CLOSED when mining the `connect` dataset as a function of the support threshold. The experimental results agree with our estimates: whereas FPCLOSE and CLOSET+ memory occupation grows exponentially because of the huge number of closed itemsets generated, our implementation needs much less memory (up to two order of magnitude) because its occupation depends linearly on N.

5.1. Experimental results

We tested our implementation on a suite of publicly available dense datasets (`chess`, `connect`, `pumsb`, `pumsb*`), and compared the performances with those of two well known state of the art algorithms: FPCLOSE [1], and CLOSET+ [7]. FPCLOSE is publicly available as `http://fimi.cs.helsinki.fi/src/fimi06.html`, while the Windows binary executable of CLOSET+ was kindly provided us from the authors. Since FPCLOSE was already proved to outperform CHARM in every dataset, we did not used CHARM in our tests.

The experiments were conducted on a Windows XP PC equipped with a 2.8GHz Pentium IV and 512MB of RAM memory. The algorithms FPCLOSE and DCI-CLOSED were compiled with the gcc compiler available in the cygwin environment.

As shown in Fig. 2.(b-f), DCI-CLOSED outperforms both algorithms in all the tests conducted. CLOSET+ performs quite well on the `connect` dataset with low supports, but in any other case it is about two orders of magnitude slower. FPCLOSE is effective in `pumsb*`, where it is near to DCI-CLOSED, but it is at one order of magnitude slower in all the other tests.

6. Conclusions

In this paper we provide a deep study on the problem of mining frequent closed itemsets, formalizing a general framework fitting every mining algorithm. Use such framework we were able to analyse the problem of duplicates rising in this new mining problem.

We have proposed a technique for promptly detecting and discarding duplicates, without the need of keeping in the main memory the whole set of closed patterns, and we implemented this technique into a new algorithm which uses a vertical bitmap representation of the dataset.

The experimental evaluation demonstrated that our approach is very effective. The proposed implementation outperforms FPCLOSE and CLOSET+ in all the test conducted and requires orders of magnitude less memory.

References

[1] Gosta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, November 2003.

[2] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proc. SIGMOD '00*, pages 1–12, 2000.

[3] Claudio Lucchese, Salvatore Orlando, Paolo Palmerini, Raffaele Perego, and Fabrizio Silvestri. kdci: a multi-strategy algorithm for mining frequent sets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, November 2003.

[4] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets. In *Proc. The 2002 IEEE International Conference on Data Mining (ICDM02)*, page 338345, 2002.

[5] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. ICDT '99*, 1999.

[6] Jian Pei, Jiawei Han, and Runying Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *SIGMOD International Workshop on Data Mining and Knowledge Discovery*, May 2000.

[7] Jian Pei, Jiawei Han, and Jianyong Wang. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *SIGKDD '03*, August 2003.

[8] Mohammed J. Zaki and Karam Gouda. Fast vertical mining using diffsets. In *Technical Report 01-1, Computer Science Dept., Rensselaer Polytechnic Institute*, March 2001.

[9] Mohammed J. Zaki and Ching-Jui Hsiao. Charm: An efficient algorithm for closed itemsets mining. In *2nd SIAM International Conference on Data Mining*, April 2002.

A Space Optimization for FP-Growth

Eray Özkural and Cevdet Aykanat
Department of Computer Engineering
Bilkent University 06800 Ankara, Turkey
{erayo,aykanat}@cs.bilkent.edu.tr

Abstract

Frequency mining problem comprises the core of several data mining algorithms. Among frequent pattern discovery algorithms, FP-GROWTH *employs a unique search strategy using compact structures resulting in a high performance algorithm that requires only two database passes. We introduce an enhanced version of this algorithm called* FP-GROWTH-TINY *which can mine larger databases due to a space optimization eliminating the need for intermediate conditional pattern bases. We present the algorithms required for directly constructing a conditional FP-Tree in detail. The experiments demonstrate that our implementation has a running time performance comparable to the original algorithm while reducing memory use up to twofold.*

1. Introduction

Frequency mining is the discovery of all frequent patterns in a transaction or relational database. Frequent pattern discovery comprises the core of several data mining algorithms such as association rule mining and sequence mining [10], dominating the running time of these algorithms. The problem involves a transaction database $T = \{X|X \subseteq I\}$ that consists in a set of transactions each of which are drawn from a set I of items. The mining algorithm finds all patterns that occur with a frequency satisfying a given absolute support threshold ϵ. In practice, the number of items $|I|$ is in the order of magnitude of 10^3 and more. The number of transactions is much larger, at least 10^5. A pattern is $X \subseteq I$, a subset of I, and the set of all patterns is 2^I. The frequency function $f(T, x) = |\{x \in Y|Y \in T\}|$ computes the number of times a given item $x \in I$ occurs in the transaction set T; it is extended to sets of items $f(T, X) = |\{X \subseteq Y|Y \in T\}|$ to compute the frequency of a pattern.

Frequency mining is the discovery of all frequent patterns in a transaction set with a frequency of support threshold ϵ and more. The set of all frequent patterns is $\mathcal{F}(T, \epsilon) = \{X|X \in 2^I \wedge f(T, X) \geq \epsilon\}$. In the algorithms, the set of frequent items $F = \{x \in I \mid f(T, x) \geq \epsilon\}$ may require special consideration. A significant property of frequency mining known as downward closure states that if $X \in \mathcal{F}(T, \epsilon)$ then $\forall Y \subset X, Y \in \mathcal{F}(T, \epsilon)$ [2].

An inherent limitation of frequency mining is the amount of main memory available [8]. In this paper, we present a space optimization to FP-Growth algorithm and we demonstrate its impact on performance with experiments on synthetic and real-world datasets. In the next section, we give the background on the FP-GROWTH algorithm. Section 3 and Section 4 explain our algorithm and implementation. Section 5 presents the experiments, following that we offer our conclusions.

2. Background

2.1. Compact structures

Compact data structures have been used for efficient storage and query/update of candidate item sets in frequency mining algorithms. SEAR [12], SPEAR [12], and DIC [6] use tries (also known as prefix trees) while FP-GROWTH [10] uses FP-Tree which is an enhanced trie structure.

Using concise structures can reduce both running time and memory size requirements of an algorithm. Tries are well known structures that are widely used

for storing strings and have decent query/update performance. The aforementioned algorithms exploit this property of the data structure for better performance. Tries are also efficient in storage. A large number of strings can be stored in this dictionary type which would not otherwise fit into main memory. For frequency mining algorithms both properties are critical as our goal is to achieve efficient and scalable algorithms. In particular, the scalability of these structures is quite high [10] as they allow an algorithm to track the frequency information of the candidate patterns for very large databases. The FP-Tree structure in FP-GROWTH allows the algorithm to maintain *all* frequency information in the main memory obtained from two database passes. Using the FP-Tree structure has also resulted in novel search strategies.

A notable work on compact structures is [15] in which a binary-trie based summary structure for representing transaction sets is proposed. The trie is further compressed using Patricia tries. Although significant savings in storage and improvements in query time are reported, the effectiveness of the scheme in a frequency mining algorithm remains to be seen. In another work in FIMI 2003 workshop [3], an algorithm called PATRICIAMINE using Patricia tries has been proposed [13]. The performance of PATRICIAMINE has been shown to be consistently good in the extensive benchmark studies of FIMI workshop [3]; it was one of the fastest algorithms although it was not the most efficient. For many applications, the average case performance may be more important than performing well in a small number of cases, therefore further research on this PATRICIAMINE would be worthwhile.

In this paper, we introduce an optimized version of FP-GROWTH. A closer analysis of it is in order.

2.2. FP-Growth algorithm

The FP-GROWTH algorithm uses the frequent pattern tree (FP-Tree) structure. FP-Tree is an improved trie structure such that each itemset is stored as a string in the trie along with its frequency. At each node of the trie, *item*, *count* and *next* fields are stored. The *item*s of the path from the root of the trie to a node constitute the item set stored at the node and the *count* is the frequency of this item set. The node link *next* is a pointer to the next node with the same *item* in the FP-Tree. Field *parent* holds a pointer to the parent node, *null* for root. Additionally, we maintain a header table which stores heads of node links accessing the linked list that spans all same items. FP-Tree stores only frequent items. At the root of the trie is a *null* item, and strings are inserted in the trie by sorting item sets in a unique[1] decreasing frequency order [10].

Table 1 shows a sample transaction set and frequent items in descending frequency order. Figure 1 illustrates the FP-Tree of sample transaction set in Table 1. As shown in [10], FP-Tree carries complete information required for frequency mining and in a compact manner; the height of the tree is bounded by maximal number of frequent items in a transaction. MAKE-FP-TREE (Algorithm 1) constructs an FP-Tree from a given transaction set T and support threshold ϵ as described.

Transaction	Ordered Frequent Items
$t_1 = \{f, a, c, d, g, i, m, p\}$	$\{f, c, a, m, p\}$
$t_2 = \{a, b, c, f, l, m, o\}$	$\{f, c, a, b, m\}$
$t_3 = \{b, f, h, j, o\}$	$\{f, b\}$
$t_4 = \{b, c, k, s, p\}$	$\{c, b, p\}$
$t_5 = \{a, f, c, e, l, p, m, n\}$	$\{f, c, a, m, p\}$

Table 1. A sample Transaction Set

In Algorithm 3 we describe FP-GROWTH which has innovative features such as:

1. Novel search strategy
2. Effective use of a summary structure
3. Two database passes

FP-GROWTH turns the frequency k-length pattern mining problem into "a sequence of k-frequent 1-item set mining problems via a set of conditional pattern bases" [10]. It is proposed that with FP-GROWTH there is "no need to generate any combinations of candidate sets in the entire mining process". With an FP-Tree $Tree$ given as input the algorithm generates all frequent patterns. There are two points in the algorithm that should be explained: the single path case and conditional pattern bases. If an FP-Tree has only a single path, then an optimization is to consider all combinations of items in the path (single path case is the ba-

1 All strings must be inserted in the same order; the order of items with the same frequency must be the same.

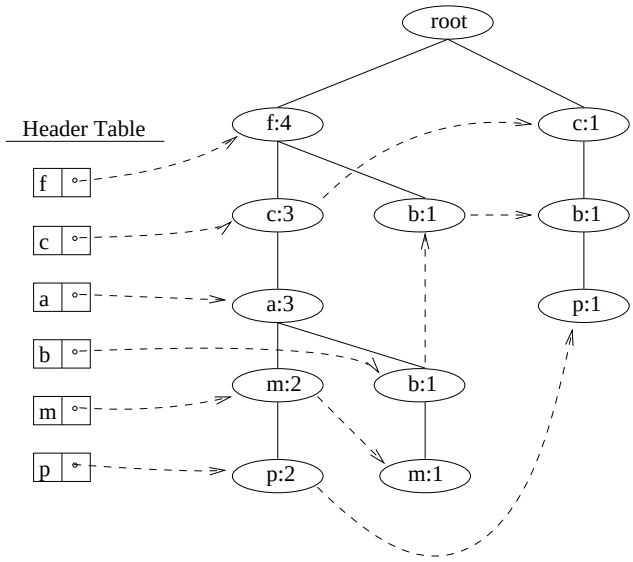


Figure 1. An FP-Tree Structure.

sis of recursion in FP-GROWTH). Otherwise, the algorithm constructs for each item a_i in the header table a *conditional pattern base* and an FP-Tree $Tree_\beta$ based on this structure for recursive frequency mining. Conditional pattern base is simply a compact representation of a derivative database in which only a_i and its prefix paths in the original $Tree$ occur. Consider path $< f:4, c:3, a:3, m:2, p:2 >$ in $Tree$. For mining patterns including m in this path, we need to consider only the prefix path of m since the nodes after m will be mined elsewhere (in this case only p). In the prefix path $< f:4, c:3, a:3 >$ any pattern including m can have frequency equal to the frequency of m, therefore we may adjust the frequencies in the prefix path as $< f:2, c:2, a:2 >$ which is called a *transformed prefix path* [10]. The set of transformed prefix paths of a_i forms a small database of patterns which cooccur with a_i and thus contains complete information required for mining patterns including a_i. Therefore, recursively mining conditional pattern bases for all a_i in $Tree$ is equivalent to mining $Tree$ (which is equivalent to ning the complete DB.). $Tree_\beta$ in FP-GROWTH is the FP-Tree of the conditional pattern base.

FP-GROWTH is indeed remarkable with its unique divide and conquer approach. Nevertheless, it may be profitable for us to view it as generating candidates despite the title of [10]. The conditional pattern base is a set of candidates among which only some of them turn out to be frequent. The main innovation however remains intact: FP-GROWTH takes advantage of a tailored data structure to solve the frequency mining problem with a divide-and-conquer method and with demonstrated efficiency and scalability. Besides, the conditional pattern base is guaranteed to be smaller than the original tree, which is a desirable property. An important distinction of this algorithm is that, when examined within the taxonomy of algorithms, it employs a unique search strategy. When the item sets tested are considered, it is seen that this algorithm is neither DFS nor BFS. The classification for FP-GROWTH in Figure 3 of [11] may be slightly misleading. As Hipp later mentions in [11], "FP-Growth does not follow the nodes of the tree . . . , but directly descends to *some part* of the itemsets in the search space". In fact, the part is so well defined that it would be unfair to classify FP-GROWTH as conducting a DFS. It does not even start with item sets of small length and proceed to longer item sets. Rather, it considers a *set of patterns at the same time* by taking advantage of the data structure. This unique search strategy makes it hard to classify FP-GROWTH in the context of traditional uninformed search algorithms.

Algorithm 1 MAKE-FP-TREE(DB, ϵ)

1: Compute F and $f(x)$ where $x \in F$
2: Sort F in frequency decreasing order as L
3: Create root of an FP-Tree T with label "null"
4: **for all** transaction $t_i \in T$ **do**
5: Sort frequent items in t_i according to L. Let sorted list be $[p|P]$ where p is the head of the list and P the rest.
6: INSERT-TRIE$([p|P])$
7: **end for**

3. An improved version of FP-Growth

During experiments with large databases, we have observed that FP-GROWTH was costly in terms of memory use. Thus, we have experimented with improvements to the original algorithm. In this section, we propose FP-GROWTH-TINY (Algorithm 4) which is an enhancement of FP-GROWTH featuring a space optimization and minor improvements. An important optimization eliminates the need for intermediate conditional pattern bases. A minor improvement comes

Algorithm 2 INSERT-TRIE($[p|P], T$)

1: **if** T has a child N such that $item[N] = item[p]$ **then**
2: $count[N] \leftarrow count[N] + 1$
3: **else**
4: Create new node N with $count = 1$, $parent$ linked to T and node-link linked to nodes with the same item via $next$
5: **end if**
6: **if** $P \neq \emptyset$ **then**
7: INSERT-TRIE(P, N)
8: **end if**

Algorithm 3 FP-GROWTH($Tree, \alpha$)

1: **if** $Tree$ contains a single path P **then**
2: **for all** combination β of the nodes in path P **do**
3: generate pattern $\beta \cup \alpha$ with support *minimum support of nodes in* β
4: **end for**
5: **else**
6: **for all** a_i in header table of $Tree$ **do**
7: generate pattern $\beta \leftarrow a_i \cup \alpha$ with $support = support[a_i]$
8: construct β's conditional pattern base and then β's conditional FP-Tree $Tree_\beta$
9: **if** $Tree_\beta \neq \emptyset$ **then**
10: FP-GROWTH($Tree_\beta, \beta$)
11: **end if**
12: **end for**
13: **end if**

from not outputting all combinations of the single path in the basis of recursion. Instead, we output a representation of this task since subsequent algorithms can take advantage of a compact representation for generating association rules and so forth.[2] Another improvement is pruning the infrequent nodes of the single path.

In the following subsection, the space optimization is discussed.

3.1. Eliminating conditional pattern base construction

The conditional tree $Tree_\beta$ can be constructed directly from $Tree$ without an intermediate conditional pattern base. The conditional pattern base in

2 For the FIMI workshop, we output all patterns separately as required. It can be argued that a meaningful mining of all frequent patterns must output them one by one.

Algorithm 4 FP-GROWTH-TINY($Tree, \alpha$)

1: **if** $Tree$ contains a single path P **then**
2: prune infrequent nodes of P
3: **if** $|P| > 0$ **then**
4: output "all patterns in 2^P and α"
5: **end if**
6: **else**
7: **for all** a_i in header table of $Tree$ **do**
8: $Tree_\beta \leftarrow$ CONS-CONDITIONAL-FP-TREE($Tree, a_i$)
9: output pattern $\beta \leftarrow a_i \cup \alpha$ with $count = f(a_i)$ $\triangleright$ $f(x)$ of $Tree$
10: **if** $Tree_\beta \neq \emptyset$ **then**
11: FP-GROWTH($Tree_\beta, \beta$)
12: **end if**
13: **end for**
14: **end if**

FP-GROWTH can be implemented as a set of *patterns*. A pattern in FP-GROWTH consists of a set of $symbols$ and an associated $count$. With a counting algorithm and retrieval/insertion of patterns directly into the FP-Tree structure, we can eliminate the need for such a pattern base. Algorithm 5 constructs a conditional FP-Tree from a given $Tree$ and a symbol s for which the transformed prefix paths are computed.

The improved procedure first counts the symbols in the conditional tree without generating an intermediate structure and constructs the set of frequent items. Then, each transformed prefix path is computed as patterns retrieved from $Tree$ and are inserted in $Tree_\beta$.

COUNT-PREFIX-PATH presented in Algorithm 6 scans the prefix paths of a given node. Since the pattern corresponding to the transformed prefix path has the count of the node, it simply adds the count to the count of all symbols in the prefix path. This step is required for construction of a conditional FP-Tree directly since an FP-Tree is based on the decreasing frequency order of F. This small algorithm allows us to compute the counts of the symbols in the conditional tree in an efficient way, and was the key observation in making the optimization possible.

Algorithm 7 retrieves a transformed prefix path for a given node excluding node itself and Algorithm 8 inserts a pattern into the FP-Tree. GET-PATTERN computes the transformed prefix path as described in [10]. INSERT-PATTERN prunes the items not present in the frequent item set F of $Tree$ (which does not have to be identical to the F of calling procedure) and sorts the pattern in decreasing frequency order to maintain FP-

Algorithm 5 CONS-CONDITIONAL-FP-TREE($Tree, s$)

1: $table \leftarrow itemtable[Tree]$
2: $list \leftarrow table[symbol]$
3: $Tree' \leftarrow$ MAKE-FP-TREE
4: $\triangleright$ Count symbols without generating an intermediate structure
5: $node \leftarrow list$
6: **while** $node \neq null$ **do**
7: COUNT-PREFIX-PATH($node, count[Tree]$)
8: $node \leftarrow next[node]$
9: **end while**
10: **for all** $sym \in range[count]$ **do**
11: **if** $count[sym] \geq \epsilon$ **then**
12: $F[Tree'] \leftarrow F[Tree'] \cup sym$
13: **end if**
14: **end for**
15: $\triangleright$ Insert conditional patterns to $Tree_\beta$
16: $node \leftarrow list$
17: **while** $node \neq null$ **do**
18: $pattern \leftarrow$ GET-PATTERN($node$)
19: INSERT-PATTERN($Tree', pattern$)
20: $node \leftarrow next[node]$
21: **end while**
22: **return** $Tree'$

Algorithm 6 COUNT-PREFIX-PATH($node, count$)

1: $prefixcount \leftarrow count[node]$
2: $node \leftarrow parent[node]$
3: **while** $parent[node] \neq null$ **do**
4: $count[symbol[node]] \leftarrow count[symbol[node]] + prefixcount$
5: $node \leftarrow parent[node]$
6: **end while**

Algorithm 7 GET-PATTERN($node$)

1: $pattern \leftarrow$ MAKE-PATTERN
2: **if** $parent[node] \neq null$ **then**
3: $count[pattern] \leftarrow count[node]$
4: $currnode \leftarrow parent[node]$
5: **while** $parent[node] \neq null$ **do**
6: $symbols[pattern] \leftarrow symbols[pattern] \cup symbol[currnode]$
7: $currnode \leftarrow parent[currnode]$
8: **end while**
9: **else**
10: $count[pattern] \leftarrow 0$
11: **end if**
12: **return** $pattern$

Tree properties and adds the obtained string to the FP-Tree structure. The addition is similar to insertion of a single string, with the difference that insertion of a pattern is equivalent to insertion of the symbol string of the pattern $count[pattern]$ times.

Algorithm 8 INSERT-PATTERN($Tree, pattern$)

1: $pattern \leftarrow pattern \cap F[Tree]$
2: Sort pattern in a predetermined frequency decreasing order
3: Add the pattern to the structure

The optimization in Algorithm 5 makes FP-GROWTH more efficient and scalable by avoiding additional iterations and cutting down storage requirements. An implementation that uses an intermediate conditional pattern base structure will scan the tree once, constructing a linked list with transformed prefix paths in it. Then, it will construct the frequent item set from the linked list, and in a second iteration insert all transformed prefix paths with a procedure similar to INSERT-PATTERN. Such an implementation would have to copy the transformed prefix paths twice, and iterate over all prefix paths three times, once in the tree, and twice in the conditional pattern list. In contrast, our optimized procedure does not execute any expensive copying operations and it needs to scan the pattern bases only twice in the tree. Besides efficiency, the elimination of extra storage requirement is significant because it allows FP-GROWTH to mine more complicated data sets with the same amount of memory.

An idea similar to our algorithm was independently explored in FP-GROWTH* by making use of information in 2-items [9]. In their implementations, Grahne and Zhu have used strategies based on 2-items to improve running time and memory usage, and they have reported favorable performance, which has also been demonstrated in the benchmarks of the FIMI '03 workshop [3].

4. Implementation notes

We have made a straightforward implementation of FP-GROWTH-TINY and licensed it under GNU GPL for public use. It has been written in C++, using GNU g++ compiler version 3.2.2.

For variable length arrays, we used `vector<T>` in standard library. For storing transactions, patterns and other structures representable as strings we have used efficient variable length arrays. We used `set<T>` to store item sets in certain places where it would be fast to do so, otherwise we have used sorted arrays to implement sets.

No low level memory or I/O optimizations were employed.

A shortcoming of the pattern growth approach is that it does not seem to be very memory efficient. We store many fields per node and the algorithm consumes a lot of memory in practice.

The algorithm has a detail which required a special code: sorting the frequent items in a transaction according to an order L, in line 2 of Algorithm 1 and line 2 of Algorithm 8. For preserving FP-Tree properties all transactions must be inserted in the very same order.[3] The items are sorted first in order of decreasing frequency and secondarily in order of indices to achieve a unique frequency decreasing order. Using this procedure, we are not obliged to maintain an L.

5. Performance study

In this section we report on our experiments demonstrating the performance of FP-GROWTH-TINY. We have measured the performance of Algorithm 3 and Algorithm 4 on a 2.4Ghz Pentium 4 Linux system with 1GB memory and a common 7200 RPM IDE hard disk. Both algorithms were run on four synthetic and five real-world databases with varying support threshold. The implementation of the original FP-GROWTH algorithm is due to Bart Goethals.[4]

We describe the data sets used for our experiments in the next two subsections. Following that, we present our performance experiments and interpret the results briefly, comparing the performance of the improved algorithm with the original one.

5.1. Synthetic data

We have used the association rule generator described in [2] for synthetic data. Synthetic databases in our evaluation have been selected from [17] and [16]. These databases have been derived from previous studies [1, 2, 14]. Table 2 explains the symbols we use for denoting the parameters of association rule generator tool. The experimental databases are depicted in Table 3. In all synthetic databases, $|I|$ is 1000, and $|\mathcal{F}_{max}|$ is 2000. The original algorithm could not process T20.I6.D1137 in memory therefore the number of transactions was decreased to 450K.

$\|T\|$	Number of transactions in transaction set
$\|t\|_{avg}$	Average size of a transaction t_i
$\|f_m\|_{avg}$	Average length of maximal pattern f_m
I	Number of items in transaction set
$\|\mathcal{F}_{max}\|$	Number of maximal frequent patterns

Table 2. Dataset parameters

Name	$\|T\|$	$\|t\|_{avg}$	$\|f_m\|_{avg}$
T10.I6.1600K	1.6×10^6	10	6
T10.I4.1024K	1.024×10^6	10	4
T15.I4.367K	3.67×10^5	15	4
T20.I6.450K	4.5×10^5	20	6

Table 3. Synthetic data sets

5.2. Real-world data

We have used five publicly available datasets in the FIMI repository. `accidents` is a traffic accident data [7]. `retail` is market basket data from an anonymous Belgian retail store [5]. The `bms-webview1`, `bms-webview2` and `bms-pos` datasets are from a benchmark study described in [4]. Some statistics of the datasets are presented in Table 4.

5.3. Memory consumption and running time

The memory consumption and running time of FP-GROWTH-TINY and FP-GROWTH are plotted for varying relative supports from 0.25% to 0.75% in Figure 2 and Figure 3 for synthetic databases and Figure 4 and Figure 5 for real-world databases except for accidents database which is a denser database that should be mined at 10% and more. The implementations were run

3 For patterns also in our implementation.

4 Goethals has made his implementation publicly available at http://www.cs.helsinki.fi/u/goethals/

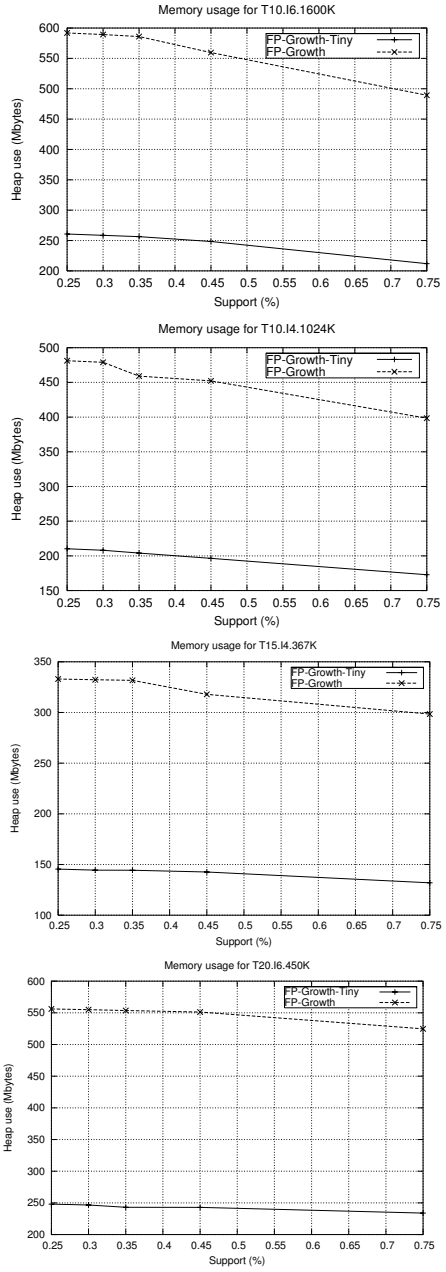


Figure 2. Memory consumption of FP-GROWTH-TINY and FP-GROWTH on synthetic databases

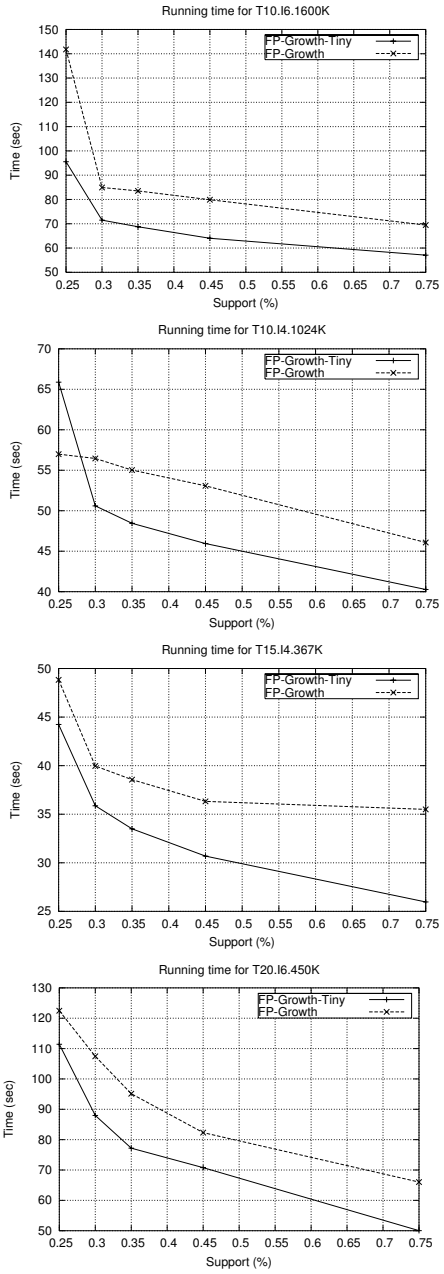
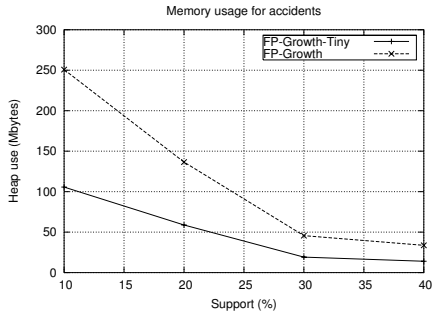
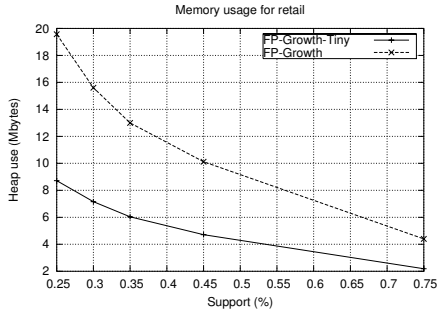
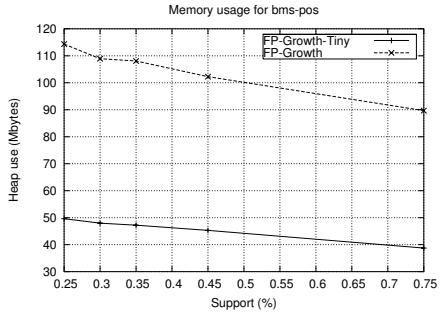
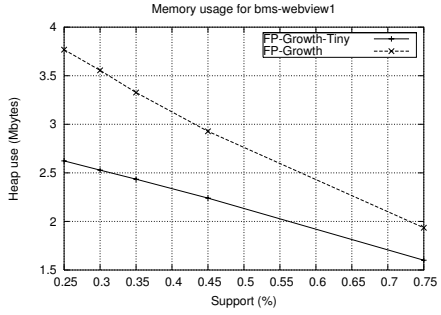


Figure 3. Running time performance of FP-GROWTH-TINY and FP-GROWTH on synthetic databases









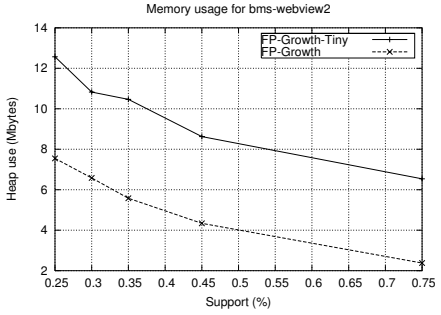
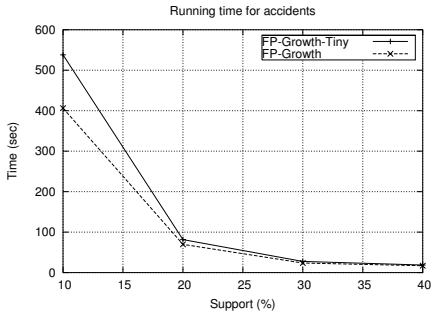
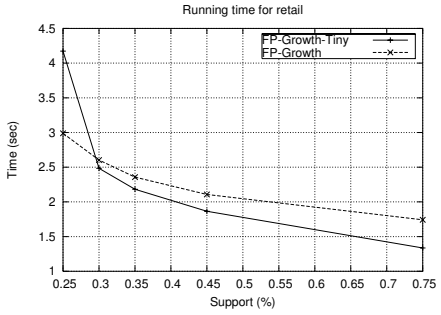
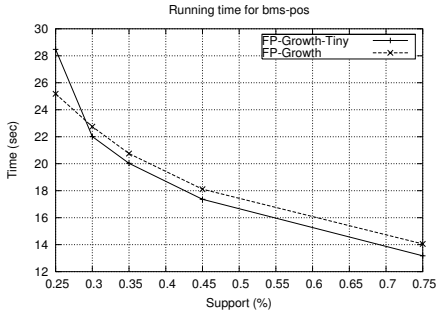
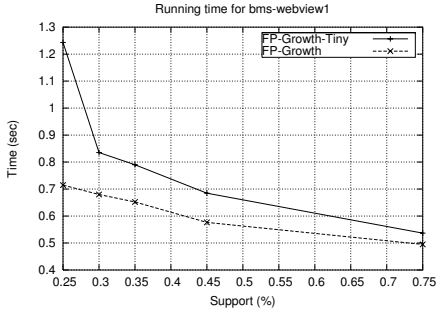


Figure 4. Memory consumption of FP-GROWTH-TINY and FP-GROWTH on real-world databases









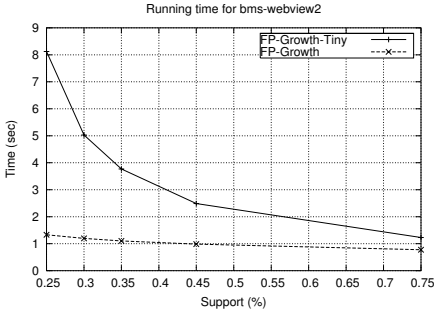


Figure 5. Running time performance of FP-GROWTH-TINY and FP-GROWTH on real-world databases

Name	$\|T\|$	$\|I\|$	$\|t\|_{avg}$
accidents	3.41×10^5	469	33.81
retail	8.82×10^4	16470	10.31
bms-pos	5.16×10^5	1657	6.53
bms-webview1	5.96×10^4	60978	2.51
bms-webview2	7.75×10^4	330286	4.62

Table 4. Real-world data sets

inside a typical KDE desktop session. The running time is measured as the wall-clock time of the system call. The memory usage is measured using the GNU glibc tool `memusage`, considering only the maximum heap size since stack use is much smaller than heap size.

The plots for synthetic datasets are similar among themselves, while we observe more variation in real-world datasets. Memory is saved in all databases, except in bms-webview2, which requires 2.74 times the memory used in FP-GROWTH; this has an adverse effect on running time as discussed below. In others, we observe that memory usage reduces down to 41.5% in accidents database with 4% support, which is 2.4 times smaller than FP-GROWTH.

Due to the optimization, our implementation can process larger databases than the vanilla version. For most problem instances, the memory consumption has been reduced more than twofold compared to the original algorithm. An advantage of our approach is that with the same amount of memory, we can process more complicated databases.[5] The experiments overall show that the conditional pattern base construction which we have eliminated has a significant space cost during the recursive construction of conditional FP-Trees.

The running time behaviors of two algorithms are quite similar on the average. Our algorithm tends to perform better and is faster in higher support thresholds, while in lower thresholds the performance gap becomes closer. FP-GROWTH-TINY runs faster except in bms-webview1, bms-webview2 and lower thresholds of T10.I4.1024K. In bms-webview1 database, FP-GROWTH-TINY runs 10-27% slower; in bms-webview2 database we observe that FP-GROWTH-TINY has slowed down by a factor of 5.56 for 0.25% support threshold, and slowdown is observed also for other support thresholds (down to 50%). In T10.I4.1024K we see 12% slowdown for 0.25% support and 2% slowdown for 0.3% support. In other problem instances FP-GROWTH-TINY, runs faster, up to 28.5% for retail dataset at 0.75% support.

In the figures, we observe a relation between memory saving and decreased running time. We had expected that improving space utilization would remarkably decrease the running time. However, we have not observed as large an improvement as we would have liked in running time. On the other hand, our trials show significant improvement in memory use contrasted to vanilla FP-GROWTH, allowing us to mine more complicated/larger datasets with the same amount of memory.

The adverse situation with bms-webview1 and bms-webview2 shows that the performance study must be extended to determine whether the undesirable behavior recurs at a large scale, since these are both sparse data sets coming from the same source. At any rate, a closer inspection of FP-GROWTH-TINY seems necessary. We anticipate that the benchmark studies at the FIMI workshop will illustrate its performance more precisely.

6. Conclusions

We have presented our version of FP-GROWTH which sports multiple improvements in Section 3. An optimization over the original algorithm eliminates a large intermediate structure required in the recursive step of the published FP-GROWTH algorithm in addition to two other minor improvements.

In Section 5, we have reported the results of our performance experiments on synthetic and real-world databases. The performance of the optimized algorithm has been compared with a publicly available FP-GROWTH implementation. We have observed more than twofold improvement in memory utilization over the vanilla algorithm. In the best case, memory size has become 2.4 times smaller, while in the worst case memory saving was not possible in a small real-world database. Typically, our implementation makes better use of memory, enabling it to mine larger and more complicated databases that cannot be processed by the original algorithm. The running time behavior of both algorithms are quite similar on the average; FP-GROWTH-TINY runs up to 28.5% percent faster, however it may run slower in a minority of instances.

5 Note that FP-GROWTH uses a compressed representation of frequency information, whose size may be thought of as related to complexity of the dataset.

References

[1] R. Agrawal and J. C. Shafer. Parallel mining of association rules. *IEEE Trans. On Knowledge And Data Engineering*, 8:962–969, 1996.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[3] M. J. Z. Bart Goethals. Fimi '03: Workshop on frequent itemset mining implementations. In *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, Melbourne, Florida, USA, 2003.

[4] Z. Z. Blue. Real world performance of association rule algorithms. In *KDD 2001*.

[5] T. Brijs, G. Swinnen, K. Vanhoof, and G. Wets. Using association rules for product assortment decisions: A case study. In *Knowledge Discovery and Data Mining*, pages 254–260, 1999.

[6] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In J. Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 255–264. ACM Press, 05 1997.

[7] K. Geurts, G. Wets, T. Brijs, and K. Vanhoof. Profiling high frequency accident locations using association rules. In *Proceedings of the* 82^{nd} *Annual Transportation Research Board*, page 18pp, Washington DC (USA), January 2003.

[8] B. Goethals. Memory issues in frequent itemset mining. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC'04)*, Nicosia, Cyprus, March 2004.

[9] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*.

[10] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In W. Chen, J. Naughton, and P. A. Bernstein, editors, *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12. ACM Press, May 2000.

[11] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Algorithms for association rule mining – a general survey and comparison. *SIGKDD Explorations*, 2(1):58–64, July 2000.

[12] A. Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, College Park, MD, 1995.

[13] A. Pietracaprina and D. Zandolin. Mining frequent itemsets using patricia tries. In *Proceedings of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*.

[14] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *The VLDB Journal*, pages 432–444, 1995.

[15] D.-Y. Yang, A. Johar, A. Grama, and W. Szpankowski. Summary structures for frequency queries on large transaction sets. In *Data Compression Conference*, pages 420–429, 2000.

[16] M. J. Zaki, S. Parthasarathy, and W. Li. A localized algorithm for parallel association mining. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 321–330, 1997.

[17] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery*, 1(4):343–373, 1997.

nonordfp: An FP-Growth Variation without Rebuilding the FP-Tree

Balázs Rácz
Computer and Automation Research Institute
of the Hungarian Academy of Sciences
H-1111 Budapest, Lágymanyosi u. 11.
bracz+fm4@math.bme.hu

Abstract

We describe a frequent itemset mining algorithm and implementation based on the well-known algorithm FP-growth. The theoretical difference is the main data structure (tree), which is more compact and which we do not need to rebuild for each conditional step. We thoroughly deal with implementation issues, data structures, memory layout, I/O and library functions we use to achieve comparable performance as the best implementations of the 1st Frequent Itemset Mining Implementations (FIMI) Workshop.

1. Introduction

Frequent Itemset Mining (FIM) is one of the first and thus most well studied problems of data mining. From the many published algorithms for this task, pattern growth approaches (FP-growth and its variations) were among the best performing ones.

This paper describes an implementation of a pattern growth algorithm. We assume the reader is familiar with the problem of frequent itemset mining[2] and pattern growth algorithms[5], like FP-growth, and hence we will omit their description here. For the reasons and goals for analyzing implementation issues of the FIM problem, see the introduction to the 1st FIMI workshop [3].

Our implementation is based on a variation of FP-tree, a similar data structure than used by FP-growth, but with a more compact representation that allows faster allocation, traversal, and optionally projection. It maintains less administrative information (the nodes do not need to store their labels (item identifiers), no header lists and children mappings are required, only counters and parent pointers), and allows more recursive steps to be carried out on the same data structure, without the need to rebuild it. There are also drawbacks of never rebuilding the tree: though projection is possible to filter conditionally infrequent items, the order of items cannot be changed to adapt to the conditional frequencies. Hence the acronym of the algorithm: nonordfp.

We describe implementational details, data structure traversal routines, memory allocation scheme, library functions and I/O acceleration, among with the algorithmic parameters of our implementation that control different traversal functions and projection. The implementation is freely available for research purposes, aimed not only for performance comparison but for further tuning of these theoretical parameters.

2. Overview of the algorithm and data structures

As a preprocession the database is scanned and the global frequencies of the items are counted. Using the minimum support infrequent items are erased and frequent items are renumbered in frequency-descending order. During a second scan of the database all transactions are preprocessed: infrequent items are erased, frequent items are translated and sorted according to the new numbering. Then the itemset is inserted into a temporary trie.

This trie is similar to the classic FP-tree: each node contains an item identifier, a counter, a parent pointer and a children map. The children map is an unordered array of pairs (child item identifier, child node index). Lookup is done with linear scan. Though this is asymptotically not an optimal structure, the number of elements in a single children map is expected to be very small, linear scan has the least overhead compared to ordered arrays with binary search, or search trees/hash maps. The implementation uses syntactics that are equivalent to the Standard Template Library (STL) interface *pair-associative container* thus it is easy to exchange this to the RB-tree based STL `map` or `hash_map`. It results in a slight performance decrease due to data structure overhead.

As a final step in the preprocessing phase this trie is copied into the data structure that the core of the algorithm will use, which we will describe later.

The core algorithm consists of a recursion. In each step the input is a condition (an itemset), a trie structure and an array of counters that describe the conditional frequencies of the trie nodes. In the body we iterate through the remaining items, calculate the conditional counters for the input condition extended with that single item, and call the recursion with the new counters and with the original or a new, projected structure, depending on the projection configuration and the percentage of empty nodes. The core recursion is shown as Algorithm 1.

Algorithm 1 Core algorithm

```
Recursion(condition, nextitem, structure, counters):
  for citem=nextitem-1 downto 0 do
    if support of citem < min_supp then
      continue at next citem
    end if
    newcounters=aggregate conditional pattern base for
    condition ∪ citem
    if projection is beneficial then
      newstructure=projection of structure to newcounters
      Recursion(condition∪citem, citem, newstructure,
      newcounters)
    else
      Recursion(condition∪citem, citem, structure, new-
      counters)
    end if
  end for
```

The recursion has four different implementations, that suit differently sized FP trees:

- Very large FP trees that contain millions of nodes are treated by *simultaneous projection*: the tree is traversed once and a projection to each item is calculated simultaneously. This phase is applied only at the first level of recursion; very large trees are expected to arise from sparse databases, like real market basket data; conditional trees projected to a single item are already small in this case.

- *Sparse aggregate* is an aggregation and projection algorithm that does not traverse those part of the tree that will not exist in the next projection. To achieve this, a linked list is built dynamically that contains the indices to non-zero counters. This is similar to the header lists of FP-trees. This aggregation algorithm is used typically near the top of the recursion, where the tree is large and many zeroes are expected. The exact choice is tunable with parameters.

- *Dense aggregate* is the default aggregation algorithm. Each node of the tree is visited exactly once and its conditional counter is added to the counter of the parent. This is the default aggregation algorithm and it is very fast due to the memory layout of the data structure, described later.

- *Single node optimization* is used near the last levels of recursion, when there is at most one node for each item left in the tree. (This is a slight generalization of the tree being a single chain.) In this case no aggregation and calculation of new counters is needed, so a specialized very simple recursive procedure starts that outputs all subsets of the paths in the tree as a frequent itemset.

The core data structure is a trie. Each node contains a counter and a pointer to the parent. As the trie is never searched, only traversed from the bottom to the top, child maps are not required. The nodes are stored in an array, node pointers are indices to this array.

Nodes that are labelled with the same item occupy a consecutive part of this array, this way we do not need to store the item identifiers in the nodes. Furthermore, we do not need the header lists, as processing all nodes of a specified item requires traversing an interval of this array. This also allows faster execution as only contiguous memory reads are executed. We only need one memory cell per frequent item to store the starting points of these intervals (the *itemstarts* array).

The parent pointers (indices) and the counters are stored in separate arrays (*parents* and *counters* rsp.) to fit the core algorithm's flexibility: if projection is not beneficial, then the recursion proceeds with the same structural information (parent pointers) but a new set of counters.

The item intervals of the trie are allocated in the array ascending, in topological order. This way the bottom-up and top-down traversal of the trie is possible with a descending rsp. ascending iteration through the array of the trie, still only using contiguous memory reads and writes. This order also allows the truncation of the tree to a particular level/item: if the structure is not rebuilt but only a set of conditional counters is calculated for an item, then the recursion can proceed with a smaller sized *newcounters* array and the original *parents* and *itemstarts* array.

The pseudocode for conditional aggregation and projection is shown as Algorithm 2 and 3. Some details are not shown, for example during the aggregation phase we calculate the expected size of the projected structure to allow decision about the projection benefits and to allocate arrays for the projected structure.

Algorithm 2 Aggregation on the compact trie data structure

cpb-aggregate(*item*, *parents*, *itemstarts*, *counters*, *newcounters*, *condfreqs*):
Input: *item* is the identifier of the item to add to the current condition; *parents* and *itemstarts* describe the current structure of the tree; *counters* and *newcounters* hold the current and new conditional counters of the nodes: *counters* is an *itemstarts*[*item*+1] sized array, *newcounters* is an *itemstarts*[*item*] sized array; *condfreqs* will hold the new conditional frequencies of the items. This is the default (dense) aggregation algorithm.

```
fill newcounters and condfreqs with zeroes
for n=itemstarts[item] to itemstarts[item+1]-1 do
  newcounters[parents[n]]=counters[n]
end for
for citem=item-1 downto 0 do
  for n=itemstarts[citem] to itemstarts[citem+1]-1 do
    newcounters[parents[n]]+=newcounters[n]
    condfreqs[citem]+=newcounters[n]
  end for
end for
```

Algorithm 3 Projection on the compact trie data structure

project(*item*, *parents*, *itemstarts*, *newcounters*, *condfreqs*, *newparents*, *newitemstarts*, *newnewcounters*):
Input: *newcounters* and *condfreqs* as computed by the aggregation algorithm; *newparents* and *newitemstarts* will hold the projected structure; *newnewcounters* will hold the values of *newcounters* reordered accordingly. The array *newcounters* is reused during the algorithm to store the old position to new position mapping.

```
newcounters[0]=0 /*node 0 is reserved for the root*/
nn=1 /*the next free node index*/
for citem=0 to item-1 do
  newitemstarts[citem]=nn
  for n=itemstarts[citem] to itemstarts[citem+1]-1 do
    if condfreqs[citem]<min_supp OR newcounters[n]==0 then
      newcounters[n]=newcounters[parents[n]] /*skip this node, the new position will be the same as the parent's*/
    else
      newnewcounters[nn]=newcounters[n]
      newcounters[n]=nn /*save the position mapping*/
      newparents[nn]=newcounters[parents[n]] /*retrieve new position of parent from the saved mapping*/
      nn++
    end if
  end for
end for
newitemstarts[item]=nn
```

3. Auxiliary routines and optimization: what counts and what doesn't

A very important observation is, that in a first and straightforward implementation of most FIM algorithms the library and auxiliary routines take 70-90% of the running time. Therefore it is essential that these tasks and routines get extra attention, especially in a FIM contest, like the FIMI workshop, where actual running times are measured and every millisecond counts.[1]

These auxiliary routines include all C/C++ library calls, memory allocation, input/output implementation, data structure management (including initialization, copy constructors, etc.). Instead of reciting some general "rule of thumbs" we describe our implementation about these issues. The most important issues are posed by those auxiliary routines that appear in the inner recursion, and thus are called proportionally to the core running time.

3.1. Input/Output

The input parsing code released for FIMI'03 is a very well written, low-level implementation, the only relevant change we made to it is that we added a buffer of several megabytes to the input file to avoid OS overhead.

The output routine was completely rewritten. The very slow `fprintf` calls are eliminated, and replaced by procedures customized for the simple format of FIMI. The most important optimization is that the output routine follows the recursive traversal of the itemset search space, and the text format of the previously outputted itemset is reused for the next output line. The library calls are eliminated or simplified as much as possible (for example outputting a zero-terminated string is approximately 20% slower than outputting a character sequence of a known length due to the additional `strlen`). This optimization is essential for the very low support test cases, where up to gigabytes of output strings are rendered.

The performance comparison of the different output routines is shown on Figure 1. Output was directed to `/dev/null`, the core running time is shown on line *no output*, where no other optimization is employed but to avoid rendering the frequent itemsets to text.

[1] This leads to an unfortunate bias: a very good low level programmer implementing a fairly good algorithm can spectacularly defeat a FIM expert implementing the best algorithm on a higher level.

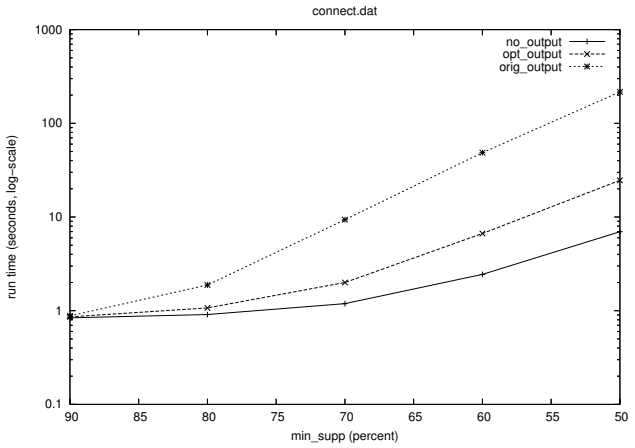


Figure 1. Performance of output routines

3.2. Memory allocation scheme

Another, similarly important point is memory allocation. In each recursive step (the number of which is equal to the total number of frequent itemsets written to output, up to tens of millions) several arrays are allocated for the conditional counters and possibly the projected structure. Calling `malloc` and `free`, or `new` and `delete` incurs a considerable overhead of these library functions due to their general memory management possibilities. Thus it is essential to reuse allocated memory.

The best approach would be to allocate these arrays for each level of the recursion in advance, but as we do not know the required size, and it can be upper bounded only by the size of the full tree, we would run out of main memory.

There are two main observations that lead to our solution to this issue. First, an allocated array can be reused for the same array in any later recursive call. Second, as the recursion proceeds into deeper levels, the required size of arrays decreases monotonically. (This is due to projections and the layout of the trie in the arrays, as discussed earlier.) Thus upon exit from a recursive step we can push the memory arrays on a stack of free blocks, this way the stack will contain decreasingly sized blocks. When entering a new level of recursion we check if the block on top of the stack is large enough. If yes, we can pop the stack and proceed. Otherwise we allocate a new memory block and proceed with an unmodified stack to the next level. Thus the monotonicity remains intact, only the free blocks are shifted one level downwards into the recursion.

In each recursive call we enter the next level several times (depending on the number of remaining items), these require differently sized arrays for the next level of recursion. It is important, that we go from the largest to the smallest when iterating through these possibilities. This way a larger memory block can be used for the smaller array, otherwise expensive reallocation would be necessary. This was already taken into consideration in Algorithm 1.

Another important factor is when and how to zero the allocated/reused memory blocks. In our first implementation all allocated memory was filled with zero before use; this resulted in up to three times more time spent in the memory fill procedure than the core recursion. This was eliminated by carefully analyzing which arrays need to be filled with zero before use. In some cases it was faster to clean up the array after use than to fill with zero before the next use (in the case when the array is sparsely filled and we have a list of non-zero elements). This way the total amount of memory zeroed was reduced (database `connect.dat`, min_supp 50%, 88 million frequent itemsets) from 54 gigabytes (!) to 915 megabytes.

This scheme is implemented as and supplemented by several block-oriented memory allocators. An important side effect of these allocators is, that the initial memory allocated by the program is high (up to 100 MB) even on very small datasets where it is not used completely. This is not a performance issue, the OS should be able to satisfy allocated but otherwise unused memory from swap space, and this parameter may be tuned if the program has to be run on computers with very limited amount of memory.

The effect of memory allocation scheme on running time is shown on Figure 2.

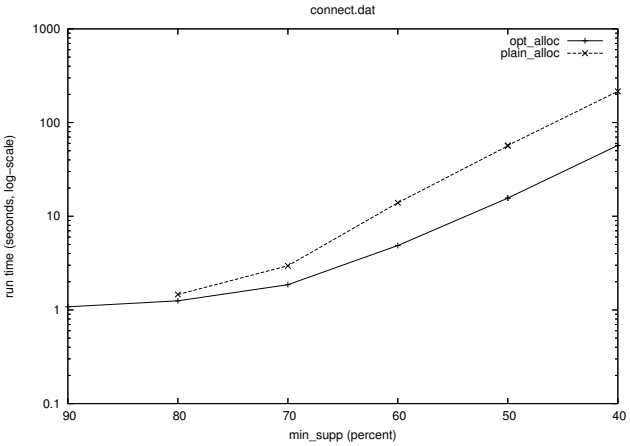


Figure 2. Performance of memory allocation schemes

3.3. Projection configuration

During the conditional pattern base aggregation phase we can easily calculate the expected size of the projected dataset, which will be the number of nodes visited with non-zero *newcounters* value.

Based on this information and the current recursion level we do projection iff the recursion level is smaller than *projlevel*, or the percentage of empty nodes exceeds *projpercent*, where *projlevel* and *projpercent* are tunable parameters of the algorithm. $projlevel = 0$ and $projpercent = 100$ means do not do projection, $projpercent = 0$ means do every projection.

We must note that "projection" here means Algorithm 3, which differs from the original concept of projected database/projected tree as premised in the introduction. Projection here does mean to compact the tree by eliminating infrequent items and nodes that have zero conditional frequency, but it does not reorder the items to continue further recursions with the conditionally most infrequent items, nor does it combine those nodes of the tree that have the same label and the same path to the root (e.g. their respective transactions differ only in conditionally infrequent items).

The effect of some projection configurations on running time is shown on Figure 3. In the line captions the first number is *projlevel*, while the second is *projpercent*. The figure shows that on many databases the projection benefits and projection costs are surprisingly well balanced, thus projection adds little enhancement to the core algorithm.

4. Performance comparison

In this section we evaluate the performance of our implementation. We use publicly available databases *accidents*, *connect*, *pumsb* and *retail* to compare the running time of our implementation to a few competitors (including the best performing ones) of the 1st Frequent Itemset Mining Implementations Workshop, *fpgrowth** [4], *patricia* [6], and *eclat_goethals*.

All of the following tests were run on a 3.0 GHz (FSB800) Intel Pentium 4 processor (hyperthreading disabled) with 2 gigabytes of dual-channel DDR400 main memory and Linux OS. Output was redirected to `/dev/null`. The running times of different implementation on the test datasets are displayed on Figure 4.

The figures show that on dense datasets the fast traversal routines take advantage, while on sparse datasets the performance is still competitive.

On sparse datasets the first level of recursion dominates the running time. To achieve better performance for these cases a specialized data structure could be employed in the simultaneous projection phase that adapts better to the skewedness of sparse datasets.

The final submitted version of the implementation (as available in the FIMI repository [1]) uses the following tuning parameters:

- *projlevel* is set to 0 (do not do any projection automatically based on the level of recursion),
- *projpercent* is set to 90% (project the tree if it will shrink to at most tenth of its size),
- *densepercent* is set to 70%, *densesize* is set to 5000 (switch from sparse to dense aggregation algorithm if there is less than 70% empty nodes, or less than 5000 nodes),
- *simultprojthres* to 1 million (do simultaneous projection on the first level of the recursion if the tree size exceeds 1 million nodes).

These values can be set in the beginning of the main program or a run-time configuration file, and should be subject to further, extensive tuning over the parameter space, which was beyond the possibilities and the time-frame available to the author. Also, on different datasets different tuning parameters may give the best performance or memory usage.

5. Conclusion and further work

We described an implementation of a pattern growth-based frequent itemset mining algorithm. We showed a compact, memory efficient representation of an FP-tree that supports the most important requirements of the core algorithm, with a memory layout that allows fast traversal.

The implementation based on this data structure and several further optimizations in the auxiliary routines performs well against some of the best competitors of the 1st Frequent Itemset Mining Implementations Workshop.

The data structure presented here can accommodate the top-down recursion approach, thereby further reducing memory need and computation time.

References

[1] FIMI repository. `http://fimi.cs.helsinki.fi`.

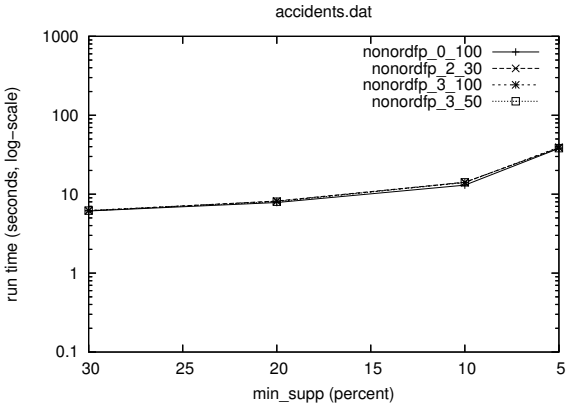
[2] B. Goethals. Survey on frequent pattern mining. Technical report, Helsinki Institute for Information Technology, 2003.

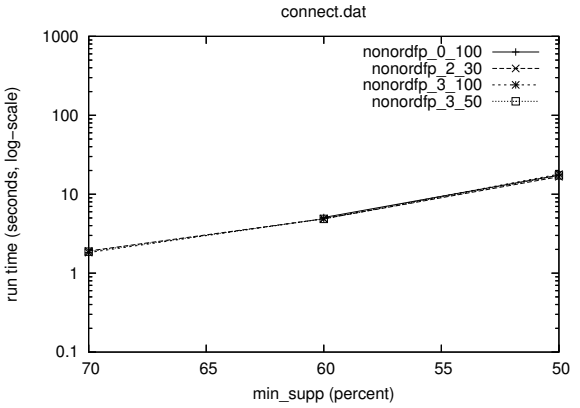
[3] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: Introduction to f imi03. In B. Goethals and M. J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 19. November 2003.

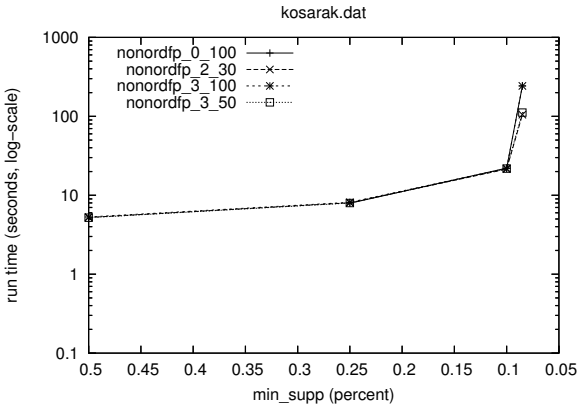
[4] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.

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

[6] A. Pietracaprina and D. Zandolin. Mining frequent itemsets using patricia tries. In B. Goethals and M. J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, Melbourne, FL, USA, November 2003.







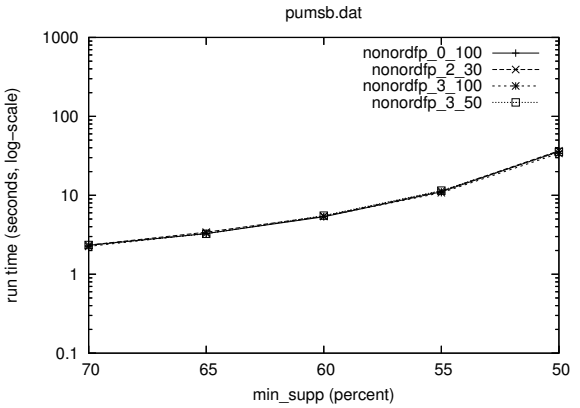
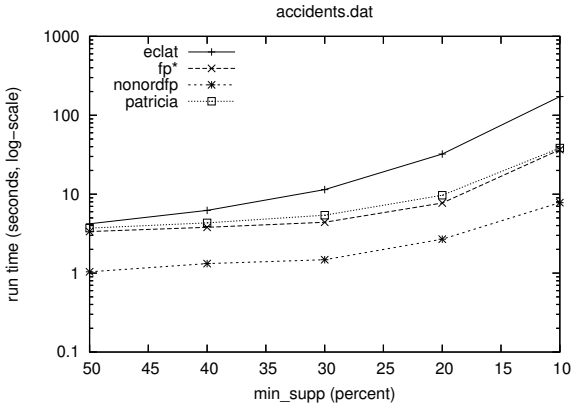
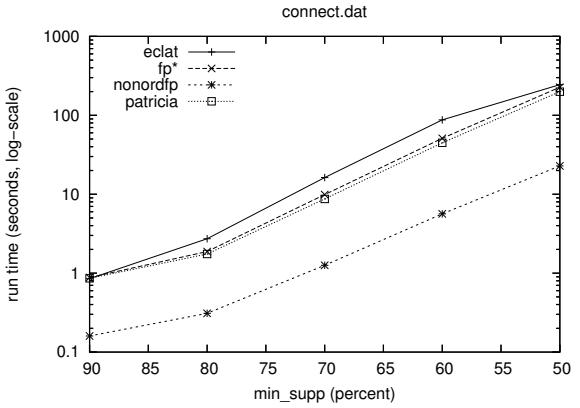
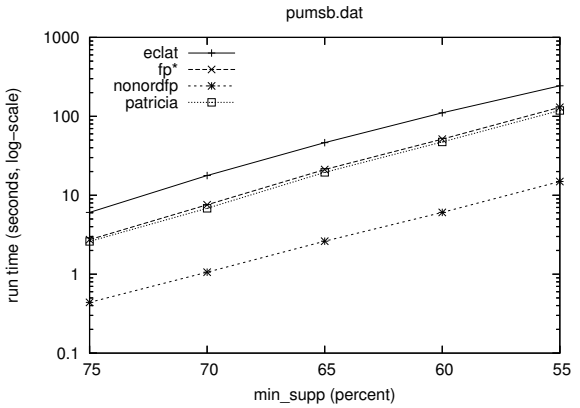


Figure 3. Performance of projection configs







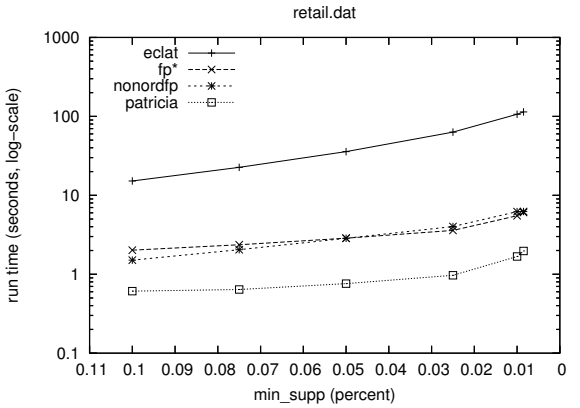


Figure 4. Performance comparison charts

Algorithmic Features of Eclat

Lars Schmidt-Thieme
Computer-based New Media Group (CGNM),
Institute for Computer Science, University of Freiburg, Germany
lst@informatik.uni-freiburg.de

Abstract

Nowadays basic algorithms such as Apriori and Eclat often are conceived as mere textbook examples without much practical applicability: in practice more sophisticated algorithms with better performance have to be used. We would like to challenge that point of view by showing that a carefully assembled implementation of Eclat outperforms the best algorithms known in the field, at least for dense datasets. For that we view Eclat as a basic algorithm and a bundle of optional algorithmic features that are taken partly from other algorithms like lcm and Apriori, partly new ones. We evaluate the performance impact of these different features and report about results of experiments that support our claim of the competitiveness of Eclat.

1. Introduction

Algorithms for mining frequent itemsets often are presented in a monolithic way and labeled with a fancy name for marketing. Careful inspection often reveals similarities with other mining algorithms that allow the transfer from a smart solution of a specific (detail) problem in one algorithm to another one. We would like to go one step further and view such mining algorithms as a basic algorithm and a bundle of algorithmic features.

Basically, there are only two large families of mining algorithms, Apriori [1] and Eclat [10] (counting fpgrowth [5] in the Eclat family what might be arguable). As the basic computation schemes of both of these algorithms are quite simple, one might get the impression, that nowadays they are good only as examples how mining algorithms work in principle for textbooks, but in practice more sophisticated algorithms have to be applied to get good performance results: for example, the four best-performing algorithms of the FIMI-03 workshop, patricia, kdci, lcm, and fpgrowth* (see [7, 6, 9, 8], for the implementations and [3] for a performance evaluation of these algorithms, respectively) do use candidate generation procedures and data structures quite different from those usually associated with the basic algorithms.

In this paper, we would like to challenge that point of view by presenting an Eclat algorithm that for dense datasets outperforms all its more sophisticated competitors.

We will start with a formal outline of Eclat algorithm in section 2. In section 3 we investigate several algorithmic features of Eclat, partly gathered from other algorithms as lcm, fpgrowth, and Apriori, partly new ones, review their usefulness in Eclat and shortly discuss their possible performance impact along with possible reasons thereof. In section 4 we present an empirical evaluation of that impact as well as a comparison with the competitor algorithms from FIMI 03 mentioned above. – We will stick to Eclat. See [2] for an excellent discussion and evaluation of different features of Apriori.

Let us fix notations for the frequent itemset mining problem in the rest of this section. Let A be a set, called **set of items** or **alphabet**. Any subset $X \in \mathcal{P}(A)$ of A is called an **itemset**. Let $\mathcal{T} \subseteq \mathcal{P}(A)$ be a multiset of itemsets, called **transaction database**, and its elements $T \in \mathcal{T}$ called **transactions**. For a given itemset $X \in \mathcal{P}(A)$, the set of transactions that contain X

$$\mathcal{T}(X) := \{T \in \mathcal{T} \mid X \subseteq T\}$$

is called **(transaction) cover of X in $\mathcal{T}$** and its cardinality

$$\sup_{\mathcal{T}}(X) := |\mathcal{T}(X)|$$

(absolute) support of X in $\mathcal{T}$. An **(all) frequent itemset mining task** is specified by a dataset $\mathcal{T}$ and a lower bound $\mathrm{minsup} \in \mathbb{N}$ on support, called **minimum support**, and asks for enumerating all itemsets with support at least minsup, called **frequent** or **(frequent) patterns**.

An itemset X is called **closed**, if

$$X = \bigcap \mathcal{T}(X)$$

i.e., if any super-itemset of X has lower support.

2. Basic Eclat Algorithm

Most frequent itemset mining algorithms as Apriori [1] and Eclat [10] use a total order on the items A of the alphabet and the itemsets $\mathcal{P}(A)$ to prevent that the same itemset, called **candidate**, is checked twice for frequency. Items orderings $\leq$ are in one-to-one-correspondence with **item codings**, i.e., bijective maps $o : A \to \{1, \ldots, n\}$ via natural ordering on $\mathbb{N}$. – For itemsets $X, Y \in \mathcal{P}(A)$ one defines their **prefix** as

$$\text{prefix}(X,Y) := \{\{x \in X \mid x \leq z\} \mid \text{maximal } z \in X \cap Y : \{x \in X \mid x \leq z\} = \{y \in Y \mid y \leq z\}\}$$

Any order on A uniquely determines a total order on $\mathcal{P}(A)$, called **lexicographic order**, by

$$X < Y :\Leftrightarrow \min(X \backslash \text{prefix}(X,Y)) < \min(Y \backslash \text{prefix}(X,Y))$$

For an itemset $X \in \mathcal{P}(A)$ an itemset $Y \in \mathcal{P}(A)$ with $X \subset Y$ and $X < Y$ is called an **extension of** X. An extension Y of X with $Y = X \cup \{y\}$ (and thus $y > \max X$) is called an **1-item-extension of** X. The extension relation organizes all itemsets in a tree, called **extension tree** or **search tree**.

Eclat starts with the empty prefix and the item-transaction incidence matrix $C_\emptyset$, shortly called **incidence matrix** in the following, and stored sparsely as list of item covers: $C_\emptyset := \{(x, \mathcal{T}(\{x\})) \mid x \in A\}$. The incidence matrix is filtered to only contain frequent items by

$$\text{freq}(C) := \{(x, \mathcal{T}_x) \mid (x, \mathcal{T}_x) \in C, |\mathcal{T}_x| \geq \text{minsup}\}.$$

that represent frequent 1-item-extensions of the prefix. For any prefix $p \in \mathcal{P}(A)$ and incidence matrix C of frequent 1-item-extensions of p one can compute the incidence matrix C_x of 1-item-extensions of $p \cup \{x\}$ by intersection rows:

$$C_x := \{(y, \mathcal{T}_x \cap \mathcal{T}_y) \mid (y, \mathcal{T}_y) \in C, y > x\}$$

where $(x, \mathcal{T}_x) \in C$ is the row representing $p \cup \{x\}$. C_x has to be filtered to get all frequent 1-item-extensions of $p \cup \{x\}$ and then this procedure is recursively iterated until the resulting incidence matrix C_x is empty, signaling that there are no further frequent 1-item-extensions of the prefix. See alg. 1 for an exact description of the Eclat algorithm.

Algorithm 1 Basic Eclat algorithm.

input: alphabet A with ordering $\leq$,
multiset $\mathcal{T} \subseteq \mathcal{P}(A)$ of sets of items,
minimum support value $\text{minsup} \in \mathbb{N}$.
output: set F of frequent itemsets and their support counts.
$F := \{(\emptyset, |\mathcal{T}|)\}$.
$C_\emptyset := \{(x, \mathcal{T}(\{x\})) \mid x \in A\}$.
$C'_\emptyset := \text{freq}(C_\emptyset) := \{(x, \mathcal{T}_x) \mid (x, \mathcal{T}_x) \in C_\emptyset, |\mathcal{T}_x| \geq \text{minsup}\}$.
$F := \{\emptyset\}$.
addFrequentSupersets$(\emptyset, C'_\emptyset)$.

function addFrequentSupersets():
input: frequent itemset $p \in \mathcal{P}(A)$ called prefix,
incidence matrix C of frequent 1-item-extensions of p.
output: add all frequent extensions of p to global variable F.
for $(x, \mathcal{T}_x) \in C$ **do**
$q := p \cup \{x\}$.
$C_q := \{(y, \mathcal{T}_x \cap \mathcal{T}_y) \mid (y, \mathcal{T}_y) \in C, y > x\}$.
$C'_q := \text{freq}(C_q) := \{(y, \mathcal{T}_y) \mid (y, \mathcal{T}_y) \in C_q, |\mathcal{T}_y| \geq \text{minsup}\}$.
if $C'_q \neq \emptyset$ **then**
addFrequentSupersets(q, C'_q).
end if
$F := F \cup \{(q, |\mathcal{T}_x|)\}$.
end for

3. Features of Eclat

The formal description of the Eclat algorithm in the last section allows us to point to several algorithmic features that this algorithm may have. These sometimes are described as implementation details, sometimes as extensions of Eclat, and sometimes as new algorithms.

3.1. Transaction Recoding

Before the first incidence matrix $C_\emptyset$ is built, it is usually beneficial 1) to remove infrequent items from the transactions, 2) to recode the items in the transaction database s.t. they are sorted in a specific order, and 3) to sort the transaction in that order. As implementations usually use the natural order on item codes, item recoding affects the order in which candidates are checked. There are several recodings used in the literature and in existing implementations of Eclat and other algorithms as Apriori (see e.g., [2]). The most common codings are coding by increasing frequency and coding by decreasing frequency. For Eclat in most cases recoding items by increasing frequency turns out to give better performance. Increasing frequency means that the length of the rows of the (initial) incidence matrix $C_\emptyset$ grows with increasing index. Let there be f_1 frequent items. As a row at index i is used $f_1 - i$ times at left side (x in the formulas above) and $i - 1$ times at right side (y in the formulas above) of the intersection operator, the order of rows is not important from the point of view of total usage in intersections. But assume the data is gray, i.e., the mining task does not contain any surprising associative patterns, where surprisingness of an itemset X is defined in

terms of lift:

$$\text{lift}(X) := \frac{\sup(X)}{|\mathcal{T}|} / \prod_{x \in X} \frac{\sup(\{x\})}{|\mathcal{T}|} /$$

$\text{lift}(X) = 1$ means that X is found in the data exactly as often as expected from the frequencies of its items, $\text{lift}(X) > 1$ or $\text{lift}(X) < 1$ means that there is an associative or dissociative effect, i.e., it is observed more often or less often than expected. Now, if $\text{lift} \approx 1$ for all or most patterns, as it is typically for benchmark datasets, then the best chances we have to identify a pattern X as infrequent before we actually have counted its support, is to check its subpattern made up from its least frequent items. And that is exactly what recoding by increasing frequency does.

3.2. Types of Incidence Structures: Covers vs. Diffsets

One of the major early improvements of Eclat algorithms has been the replacement of item covers in incidence matrices by their relative complement in its superpattern, so called **diffsets**, see [11]. Instead of keeping track of $\mathcal{T}(q)$ for a pattern q, we keep track of $\mathcal{T}(p) \setminus \mathcal{T}(q)$ for its superpattern p, i.e., $q := p \cup \{x\}$ for an item $x > \max(p)$. $\mathcal{T}(p) \setminus \mathcal{T}(q)$ are those transactions we loose if we extend p to q, i.e., its additional **defect** relative to p. From an incidence matrix C of item covers and one of the 1-item-extensions $(x, \mathcal{T}_x) \in C$ of its prefix we can derive the incidence matrix D of item defects of this extension by

$$D_x := \{(y, T_x \setminus T_y) \,|\, (y, T_y) \in C, y > x\}$$

From an incidence matrix D of item defects and one of its 1-item-extensions $(x, \mathcal{T}_x) \in D$ of its prefix we can derive the incidence matrix D_x of item defects of this extension by

$$D_x := \{(y, T_y \setminus T_x) \,|\, (y, T_y) \in D, y > x\}$$

If we expand first by x and then by y in the second step, we loose transactions that not contain y unless we have lost them before as they did not contain x.

Defects computed from covers may have at most size

$$\text{maxdef}_p := |\mathcal{T}(p)| - \text{minsup},$$

those computed recursively from other defects at most size

$$\text{maxdef}_{p \cup \{x\}} := \text{maxdef}_p - |\mathcal{T}_x|$$

1-item-extensions exceeding that maximal defect are removed by a filter step:

$$\text{freq}(D) := \{(x, \mathcal{T}_x) \,|\, (x, \mathcal{T}_x) \in C, |\mathcal{T}_x| \leq \text{maxdef}\}.$$

Computing intersections of covers or set differences for defects are computationally equivalent complex tasks. Thus, the usage of defects can improve performance only by leading to smaller incidence matrices. For dense datasets where covers overlap considerably, intersection reduces the size of the incidence matrix only slowly, while defects cut down considerably. On the other side, for sparse data using defects may deteriorate the performance. – Common items in covers also can be removed by omitting equisupport extensions (see section 3.5).

While there is an efficient transition from covers to defects as given by the formula above, the reverse transition from defects to covers seems hard to perform efficiently as all defects on the path to the root of the search tree would have to be accumulated.

Regardless which type of incidence matrix is used, it can be stored as sparse matrix (i.e., as list of lists as discussed so far) or as dense (bit)matrix (used e.g, by [2]).

A third alternative for keeping track of item-transaction incidences is not to store item covers as a set of incident transaction IDs per 1-item-extension, but to store all transactions $\mathcal{T}(p)$ that contain a given prefix p in a trie (plus some index structure, known as frequent pattern tree and first used in fp-growth; see [5]). Due to time restrictions, we will not pursue this alternative further here.

3.3. Incidence Matrix Derivation

For both incidence matrices, covers and defects, two different ways of computing the operator that derives an incidence matrix from a given incidence matrix recursively, i.e., intersection and set difference, respectively, can be chosen. The straightforward way is to implement both operators as set operators operating on the sets of transaction IDs.

Alternatively, intersection and difference of several sets $\mathcal{T}_y, y > x$ of transactions by another set $\mathcal{T}_x$ of transactions also can be computed in parallel using the original transaction database by counting in IDs of matching transactions (called occurrence deliver in [9]). To compute $\mathcal{T}'_y := \mathcal{T}_y \cap \mathcal{T}_x$ for several $y > x$ one computes

$$\forall T \in \mathcal{T}_x \forall y \in T : \mathcal{T}'_y := \mathcal{T}'_y \cup \{T\}.$$

Similar, to compute $\mathcal{T}'_y := \mathcal{T}_x \setminus \mathcal{T}_y$ for several $y > x$ one computes

$$\forall T \in \mathcal{T}_x \forall y \notin T : \mathcal{T}'_y := \mathcal{T}'_y \cup \{T\}.$$

3.4. Initial Incidence Matrix

Basic Eclat first builds the incidence matrix $C_\emptyset$ of single item covers as initial incidence matrix and then recursively derives incidence matrices C_p of covers of increasing prefixes p or D_p of defects.

Obviously, one also can start with $D_\emptyset$, the matrix of item cover complements. This seems only useful for very dense

datasets as it basically inverts the encoding of item occurrence and non-occurrence (dualization).

It seems more interesting to start already with incidence matrices for 1-item-prefixes, i.e., not to use Eclat computation schemes for the computation of frequent pairs, but count them directly from the transaction data. For Apriori this is a standard procedure. The cover incidence matrix $C_x = \{(y, \mathcal{T}_y)\}$ for an frequent item x, i.e., $\mathcal{T}_y = \mathcal{T}(\{x\}) \cap \mathcal{T}(\{y\})$, is computed as follows:

$$\forall T \in \mathcal{T} : \text{if } x \in T : \forall y \in T, y > x : \mathcal{T}_y := \mathcal{T}_y \cup \{T\}.$$

The test for $x \in T$ looks worse than it is in practice: if transactions are sorted, items x are processed in increasing order, and deleted from the transaction database after computation of C_x, then if x is contained in a transaction T it has to be its first item.

Similarly, a defect incidence matrix $D_x = \{(y, \mathcal{T}_y)\}$ for a frequent item x, i.e., $\mathcal{T}_y = \mathcal{T}(\{x\}) \setminus \mathcal{T}(\{y\})$, can be computed directly from the transaction database by

$$\forall T \in \mathcal{T} : \text{if } x \in T : \forall y \notin T, y > x : \mathcal{T}_y := \mathcal{T}_y \cup \{T\}.$$

If C_x or D_x is computed directly from the transaction database, then it has to be filtered afterwards to remove infrequent extensions. An additional pass over $\mathcal{T}$ in advance can count pair frequencies for all x, y in parallel, so that unnecessary creation of covers or defects of infrequent extensions can be avoided.

3.5. Omission of Equisupport Extensions

Whenever an extension x has the same support as its prefix p, it is contained in the closure $\bigcap \mathcal{T}(p)$ of the prefix. That means that one can add any such equisupport extension to any extension of p without changing its support; thus, one can omit to explicitly check its extensions. Equisupport extensions can be filtered out and kept in a separate list E for the active branch: whenever an itemset X is output, all its $2^{|E|}$ supersets $X' \subseteq X \cup E$ are also output.

Omission of equisupport extensions is extremely cheap to implement as it can be included in the filtering step that has to check support values anyway. For dense datasets with many equisupport extensions, the number of candidates that have to be checked and accordingly the runtime can be reduced drastically.

3.6. Interleaving Incidence Matrix Computation and Filtering

When the intersection $\mathcal{T}_x \cap T_y$ of two sets of transaction IDs is computed, we are interested in the result of this computation only if it is at least of size minsup, as otherwise it is filtered out in the next step. As the sets of transactions are sorted, intersections are computed by iterating over the lists of transaction IDs and comparing items. Once one of the tails of the lists to intersect is shorter than minsup minus the length of the intersection so far, we can stop and drop that candidate, as it never can become frequent. – For set difference of maximal length maxdef a completely analogous procedure can be used.

3.7. Omission of Final Incidence Matrix Derivation

Finally, once the incidence matrix has only two rows, the result of the next incidence matrix derivation will be an incidence matrix with a single row. As this is only checked for frequency, but its items are not used any further, we can omit to generate the list of transaction IDs and just count its length.

3.8. IO

So far we have investigated features that are specific to Eclat and the frequent itemset mining problem. Though these specific algorithmic features are what should be of primary interest, we noticed in our experiments, that often different IO mechanism dominate runtime behavior. At least three output schemes are implemented in several of the algorithms available: IO using C++ streams, IO using `printf`, and IO using handcrafted rendering of integer itemsets to a char buffer and writing that buffer to files using low-level `fwrite` (for the latter see e.g., the implementation of lcm, [9]). Handcrafted rendering of itemsets to char buffers is by far the fastest method; especially for low support values, when huge numbers of patterns are output, the runtime penalty from slower output mechanisms cannot be compensated by better mining mechanisms whatsoever.

4. Evaluation

By evaluating different features of Eclat we wanted to answer two questions:

1. What features will make Eclat run fastest? Especially, what is its marginal runtime improvement of each feature in a sophisticated Eclat implementation?

2. Is Eclat competitive compared with more complex algorithms?

To answer the question about the runtime improvement of the different features, we implemented a modular version of Eclat in C++ (basically mostly plain C) that allows the flexible inclusion or exclusion of different algorithmic features. At the time of writing the following features are implemented: the incidence structure types covers and diffsets (COV, DIFF), transaction recoding (none, decreasing,

increasing; NREC, RECDEC, RECINC), omission of equisupport extensions (NEE), interleaving incidence matrix computation and filtering (IFILT), and omission of final incidence matrix (NFIN). As initial incidence matrix alway covers of frequent 1-itemsets ($C_\emptyset$) was used.

To measure the marginal runtime improvement of a feature we configured a sophisticated Eclat algorithm with all features turned on (SOPH:= DIFF, RECINC, NEE+, IFILT+, NFIN+) and additionally for each feature an Eclat algorithm derived from SOPH by omitting this feature (SOPH-DIFF, SOPH-RECINC (decreasing encoding), SOPH-REC (no recoding at all), SOPH-NEE+, SOPH-IFILT+, SOPH-NFIN+).

We used several of the data sets and mining tasks that have been used in the FIMI-03 workshop ([4]): accidents, chess, connect, kosarak, mushroom, pumsb, pumsbstar, retail, T10I5N1KP5KC0.25D200K, T20I10N1KP5KC0.25D200K, and T30I15N1KP5KC0.25-D200K. All experiments are ran on a standard Linux box (P4/2MHz, 1.5GB RAM, SuSE 9.0). Jobs were killed if they run more than 1000 seconds and the corresponding datapoint is missing in the charts.

A sample from the results of these experiments can be seen in fig. 1 (the remaining charts can be found at http://www.informatik.uni-freiburg.de/cgnm/papers/-fimi04). One can see some common behavior across datasets and mining tasks:

- For dense mining tasks like accidents, chess, etc. SOPH is the best configuration.
- For sparse mining tasks like retail, T20I10N1KP5KC0-25D200K etc. SOPH-diff is the best configuration, i.e., using defects harms performance here – both effects are rather distinct.
- Recoding is important and shows a huge variety w.r.t. runtime: compare e.g., decreasing and no encoding for connect: the natural encoding is not much worse than decreasing encoding, but the curve for increasing encoding shows what harm the wrong encoding can do: note that the natural encoding is close to optimal only by mere chance and could be anywhere between increasing and decreasing!
- Omitting equisupport extensions also shows a clear benefit for most mining tasks, with exception for mushroom.
- Compared with other features, the impact of the features IFILT and NFIN is neglectible.

To answer the second question about competitiveness of Eclat compared with more advanced frequent pattern mining algorithms we have chosen the four best-performing algorithms from the FIMI-03 workshop: patricia, kdci, lcm, and fpgrowth* (see [7, 6, 9, 8], for the implementations and [3] for a performance evaluation of these algorithms, respectively).

Again, a sample from the results of these experiments can be seen in fig. 2 (the remaining charts also can be found at http://www.informatik.uni-freiburg.de/cgnm/-papers/fimi04). For several datasets (chess, connect, mushroom, pumsb, and – not shown – pumsbstar), Eclat-SOPH is faster than all other algorithms. For some datasets it is faster for high minimum support values, but beaten by fpgrowth* when support values get smaller (accidents, T30I15N1KP5KC0-25D200K) and for some datasets its performance is really poor (retail, T20I10N1KP5KC0-25D200K, and – not shown – kosarak and T10I5N1KP5KC0.25D200K). We can draw two conclusions from this observations: 1) at least for dense datasets, Eclat-SOPH is faster than all its competitors, 2) for sparse datasets, Eclat-SOPH is not suitable. Recalling our discussion on the potential of using defects instead of covers and on starting with frequent 2-itemsets instead of with frequent 1-itemsets, the latter conclusion is not very surprising.

5. Outlook

There are at least four more features we do not have investigated yet: using tries to store the transaction covers, the method to compute the initial incidence matrix, pruning, and memory management. Our further research will try to address questions about the impact of these features.

This update of optimization for dense datasets has to be complemented with research in performance drivers for sparse datasets. As can be seen from our results, Eclat seems not suited well for that task. Though using covers instead of defects improves performance, it still is not competitive with other algorithms in the field.

Furthermore, results for dense datasets will have to be compared with that of the next generation of mining algorithms we expect as outcome of FIMI'04 and eventually new features of these algorithms have to be integrated in Eclat. We expect both, that Eclat is clearly beaten at FIMI'04 as well that it will be not too hard to identify the relevant features and integrate them in Eclat.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94), Santiago de Chile, September 12-15*, pages 487–499. Morgan Kaufmann, 1994.

[2] C. Borgelt. Efficient implementations of apriori and eclat. In Goethals and Zaki [4].

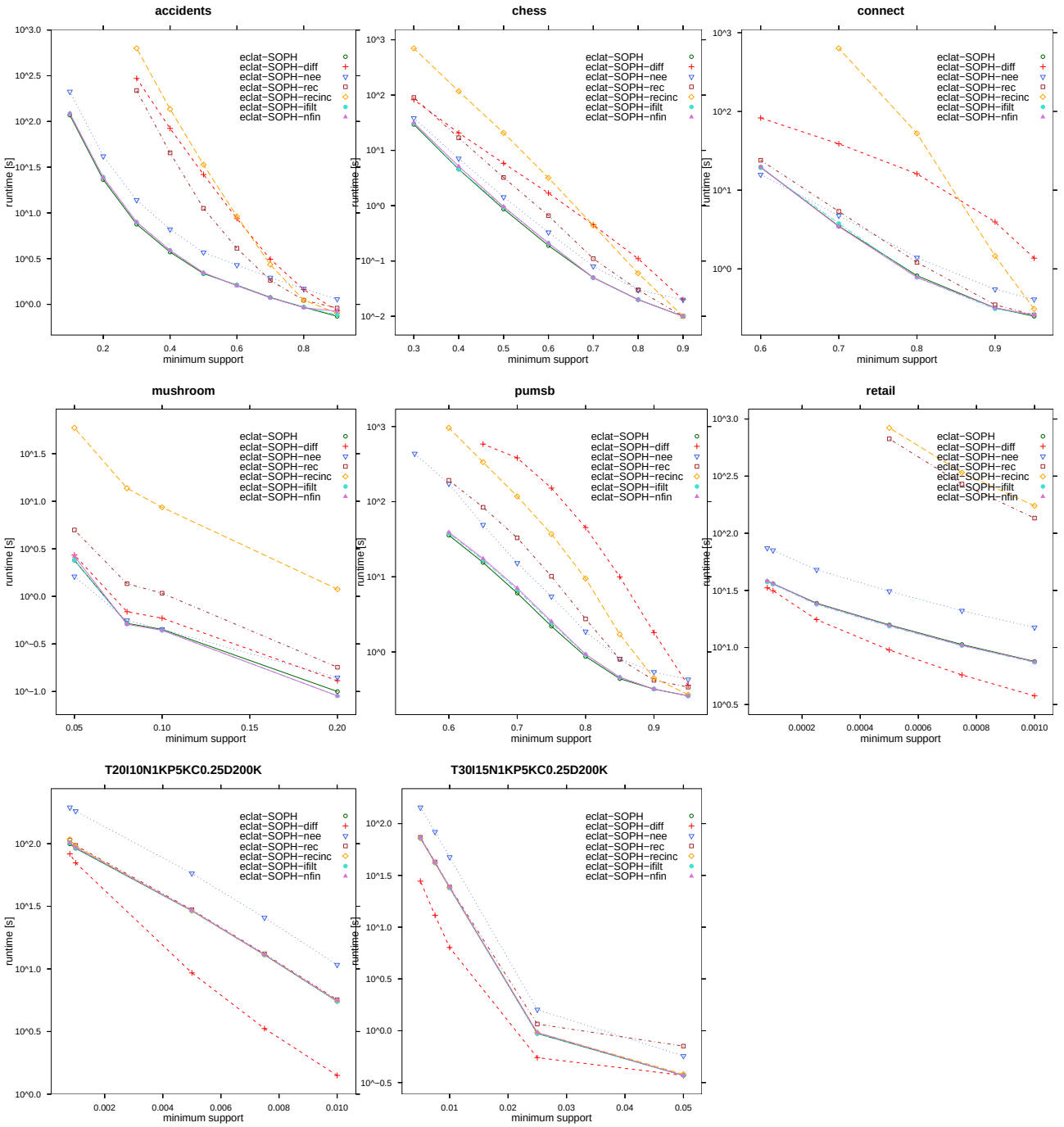


Figure 1. Evaluation of the marginal effect of different features of Eclat on runtime.

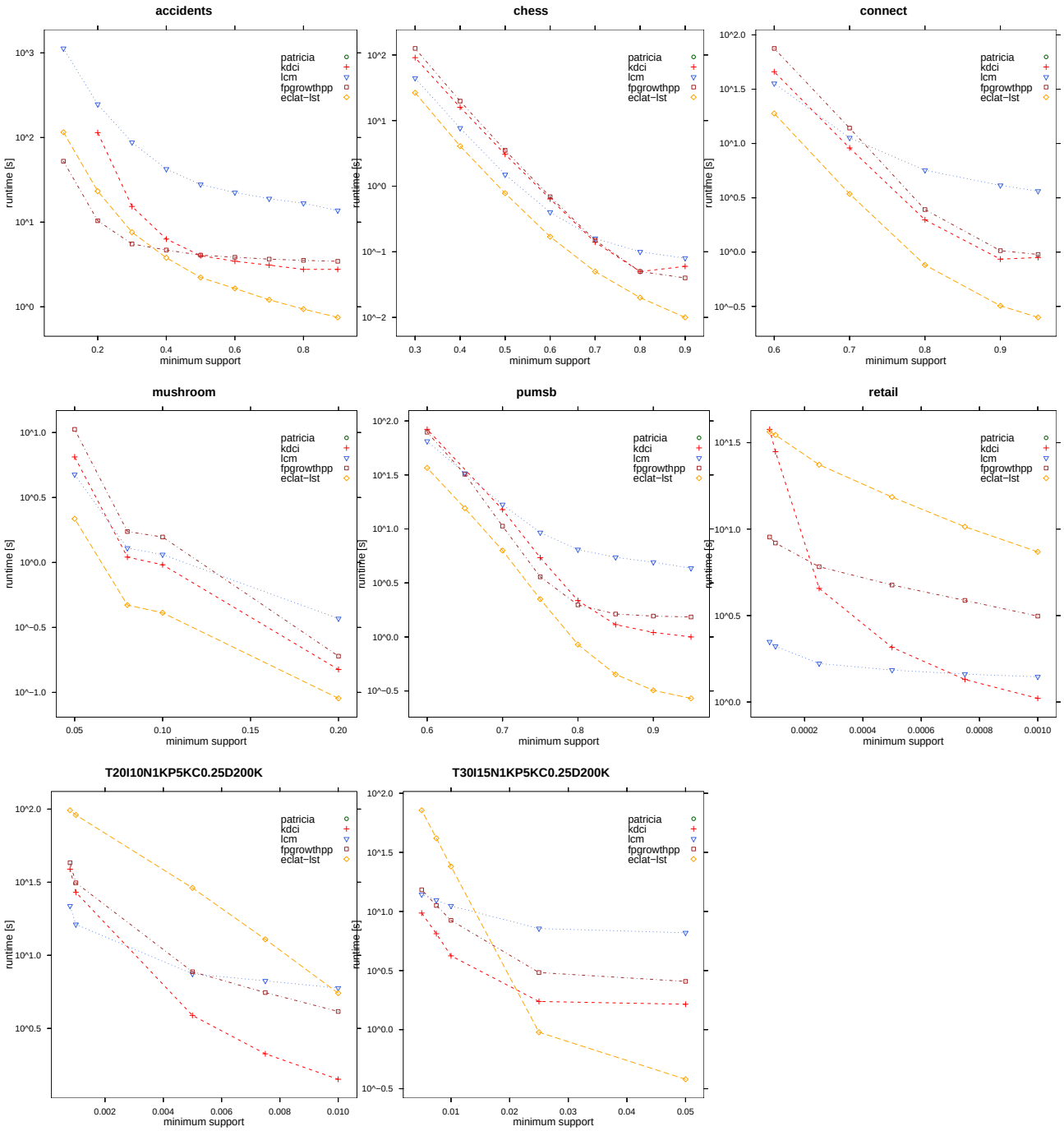


Figure 2. Evaluation of Eclat-SOPH (= eclat-lst) vs. fastest algorithms of the FIMI-03 workshop.

[3] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: Introduction to fimi03. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Melbourne, Florida, USA, November 19* [4].

[4] B. Goethals and M. J. Zaki, editors. *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Melbourne, Florida, USA, November 19, 2003*. 2003.

[5] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM Press, 2000.

[6] S. Orlando, C. Lucchese, P. Palmerini, R. Perego, and F. Silvestri. kdci: a multi-strategy algorithm for mining frequent sets. In Goethals and Zaki [4].

[7] A. Pietracaprina and D. Zandolin. Mining frequent itemsets using patricia tries. In Goethals and Zaki [4].

[8] G. sta Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In Goethals and Zaki [4].

[9] T. Uno, T. Asai, Y. Uchida, and H. Arimura. Lcm: An efficient algorithm for enumerating frequent closed item sets. In Goethals and Zaki [4].

[10] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2000.

[11] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. Technical report, RPI, 2001. Tech. Report. 01-1.

CT-PRO: A Bottom-Up Non Recursive Frequent Itemset Mining Algorithm Using Compressed FP-Tree Data Structure

Yudho Giri Sucahyo Raj P. Gopalan
Department of Computing, Curtin University of Technology
Kent St, Bentley
Western Australia 6102
{sucahyoy, raj}@cs.curtin.edu.au

Abstract

Frequent itemset mining (FIM) is an essential part of association rules mining. Its application for other data mining tasks has also been recognized. It has been an active research area and a large number of algorithms have been developed. In this paper, we propose another pattern growth algorithm which uses a more compact data structure named Compressed FP-Tree (CFP-Tree). The number of nodes in a CFP-Tree can be up to half less than in the corresponding FP-Tree. We also describe the implementation of CT-PRO which utilize the CFP-Tree for FIM. CT-PRO traverses the CFP-Tree bottom-up and generates the frequent itemsets following the pattern growth approach non-recursively. Experiments show that CT-PRO performs better than OpportuneProject, FP-Growth, and Apriori. A further experiment is conducted to determine the feasible performance range of CT-PRO and the result shows that CT-PRO has a larger performance range compared to others. CT-PRO also performs better compared to LCM and kDCI that are known as the two best algorithms in FIMI Repository 2003.

1. Introduction

Since its introduction in [1] the problem of efficiently generating frequent itemsets has been an active research area and a large number of algorithms have been developed for it; for surveys, see [2-4]. Frequent itemset mining (FIM) is an essential part of association rules mining (ARM). Since FIM is computationally expensive, the general performance of ARM is determined by it. The frequent itemset concept has also been extended for many other data mining tasks such as classification [5, 6], clustering [7], and sequential pattern discovery [8].

The data structures used play an important role in the performance of FIM algorithms. The various data structures used by FIM algorithms can be categorized as either array-based or tree-based. An example of a successful array-based algorithm for FIM is *H-Mine* [9]. It uses a data structure named *H-struct*, which is a combination of arrays and hyper-links. It was shown in [9] that *H-struct* works well for sparse datasets as *H-Mine* outperforms *FP-Growth* [10] on these datasets (note that both *H-Mine* and *FP-Growth* follows the same pattern growth method). However, the hyper-structure is not efficient on dense datasets and therefore *H-Mine* switches to *FP-Growth* for such datasets.

FP-Growth [10] shows good performance on dense datasets as it uses a compact data structure named *FP-Tree*. *FP-Tree* is a prefix tree with links between nodes containing the same item. A tree data structure is suitable for dense datasets since many transactions will share common prefixes so that the database could be compactly represented. However, for sparse datasets the tree will be bigger and bushier, and therefore its construction cost and traversal cost will be higher than array-based data structures.

The strengths of *H-Mine* and *FP-Growth* were combined in the recent pattern growth FIM algorithm named *OpportuneProject* (*OP*) [11]. *OP* is an adaptive algorithm that opportunistically chooses an array-based or a tree-based data structure depending on the sub-database characteristics.

In this paper, we describe our new data structure named *Compressed FP-Tree* (*CFP-Tree*) and also the implementation of our new FIM algorithm named *CT-PRO* that was first introduced in [12]. Here we report the compactness of *CFP-Tree* with *FP-Tree* at several support levels on the various datasets generated using the synthetic data generator [13]. The performance of *CT-PRO* is compared with *Apriori* [14, 15], *FP-Growth* [10], and *OP* [11].

Sample datasets such as real-world *BMS* datasets [3] or *UCI Machine Learning Repository* datasets [16] do not cover the full range of densities from sparse to dense. Some algorithms may work well for a certain dataset but may not be feasible when the dimensions of the database change (i.e. number of transactions, number of items, average number of items per transaction etc.). Therefore, a further study has been done in this paper, to show the feasible performance range of the algorithms. The more extensive testing of the algorithms is carried out using a set of databases with varying number of both transactions

and average number of items per transaction. For each dataset, all the algorithms are tested on supports of 10% to 90% in increments of 10%. The experimental results are reported in detail.

To show how well *CT-PRO* compares with algorithms in FIMI Repository 2003 [17], two best algorithms from the last workshop, *LCM* [18] and *kDCI* [19], are selected for comparison. The result shows that *CT-PRO* outperforms these and therefore all others.

The structure of the rest of this paper is as follows: In Section 2, we introduce the *CFP-Tree* data structure and report the results of experiments in evaluating its compactness. In Section 3, we describe the *CT-PRO* algorithm with a running example. We discuss the complexity of *CT-PRO* algorithm in Section 4. The performance of the algorithm on various datasets is compared against other algorithms in Section 5. Section 6 contains conclusions of our study.

2. *Compressed FP-Tree* Data Structure

In this section, a new tree-based data structure, named *Compressed FP-Tree* (*CFP-Tree*), is introduced. It is a variant of *CT-Tree* data structure that we introduced in [20] with the following major differences: items are sorted in descending order of their frequency (instead of ascending order, as in *CT-Tree*) and there is a link to the next node with the same item node (while links are not present in *CT-Tree*). The *CFP-Tree* is defined as follows:

Definition 1 (*Compressed FP-Tree* or *CFP-Tree*). *A Compressed FP-Tree is a prefix tree with the following properties:*

1. *It consists of an ItemTable and a tree whose root represents the index of the item with the highest frequency and a set of subtrees as the children of the root.*
2. *The ItemTable contains all frequent items sorted in descending order by their frequency. Each entry in the ItemTable consists of four fields, (1) index, (2) item-id, (3) frequency of the item, and (4) a pointer pointing to the root of the subtree of each frequent item.*
3. *If $I = \{i_1, i_2, \ldots i_k\}$ is a set of frequent items in a transaction, after being mapped to their index-id, then the transaction will be inserted into the Compressed FP-Tree starting from the root of a subtree to which i_1 in the ItemTable points.*
4. *The root of the Compressed FP-Tree is the level 0 of the tree.*
5. *Each node in the Compressed FP-Tree consists of four fields: node-id, a pointer to the next sibling, a pointer to the next node with the same id, and a count array where each entry corresponds to the number of occurrences of an itemset. If $C = \{C_0, C_1, \ldots C_k\}$ is a set of counts in the count array attached to a node and the index of the array starts from zero, then C_i is the count of a transaction with an itemset along the path from the node at level i to the node where C_i is located.* ■

The following lemma provides the worst-case space complexity of a *CFP-Tree*.

Lemma 1. Let *n* be the number of frequent items in the database for a certain support threshold. The number of nodes of the *CFP-Tree* is bounded by 2^{n-1}, which is half of the maximum for a full prefix tree.

Rationale. *If $I_F = \{i_{F1}, \ldots i_{Fn}\}$ is a set of distinct items in a CFP-Tree where i_{F1}, $i_{F2}, \ldots i_{Fn}$ are lexicographically ordered. The maximum number of nodes under subtrees i_{F1}, $i_{F2}, \ldots i_{Fn}$ is 2^{n-1}, $2^{n-2} \ldots 2^0$ respectively. Since the CFP-Tree is actually the subtree i_{F1} then the maximum number of nodes of the CFP-Tree is 2^{n-1}.* ■

Compared to *FP-Tree*, *CFP-Tree* has some important differences, as follows:

1. *FP-Tree* stores the item id in the tree while, in *CFP-Tree*, item ids are mapped to an ascending sequence of integers that is actually the array index in *ItemTable*.
2. The *FP-Tree* is compressed by removing identical subtrees of a complete *FP-Tree* and succinctly storing the information from them in the remaining nodes. All subtrees of the root of the *FP-Tree* (except the leftmost branch) are collected together at the leftmost branch to form the *CFP-Tree*..
3. Each node in the *FP-Tree* (except the root) consists of three fields: *item-id*, *count* and *node-link*. *Count* registers the number of transactions represented by the portion of the path reaching this node. *Node-link* links to the next node with the same *item-id*. Each node in the *CFP-Tree* consists of three fields: *item-id*, *count array* and *node-link*. The *count array* contains counts for item subsets in the path from the root to that node. The index of the cells in the array corresponds to the level numbers of the nodes above.
4. *FP-Tree* has a *HeaderTable* consisting of two fields: *item-id* and a pointer to the first node in the *FP-Tree* carrying the nodes with the same *item-id*. *CFP-Tree* has an *ItemTable* consisting of four fields: *index*, *item-id*, *count of the item* and *a pointer to the root of the subtree of each item*. The root of each subtree has a link to the next node with the same-item-node. Both *HeaderTable* and *ItemTable* store only frequent items.

Figure 1 shows the *FP-Tree* and the *CFP-Tree* for a sample database. In this case, the *FP-Tree* is a complete

tree for items 1-4. In this example, the number of nodes in the *FP-Tree* is twice that of the corresponding *CFP-Tree*. However, most datasets do not have such an extreme characteristic as in this example.

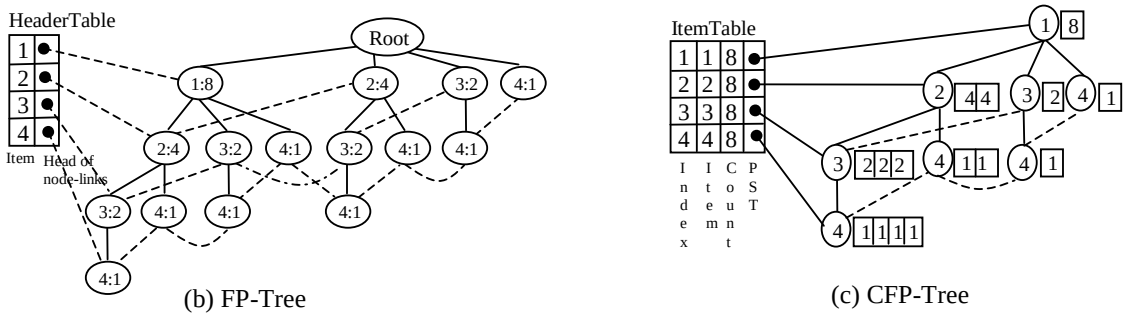
Figure 2 shows the compactness of *CFP-Tree* compared to *FP-Tree* on several synthetic datasets at various support levels (the characteristics of the datasets are explained later in Section 5.2). *CFP-Tree* has a smaller number of nodes compared to *FP-Tree* in all cases.

3. CT-PRO Algorithm

In this section, a new method that traverses the tree in a bottom-up strategy, and implemented non-recursively, is presented. The *CFP-Tree* data structure is used to compactly represent transactions in the memory. The algorithm is called *CT-PRO* and it has three steps in it: finding the frequent items, constructing the *CFP-Tree*, and mining. Algorithm 1 shows the first two steps in *CT-PRO*.

Tid	Items	Tid	Items	Tid	Items
1	1 2 3 4	6	2	11	1
2	2 4	7	1 4	12	2 3 4
3	1 3 4	8	1 2 3	13	1 2
4	3	9	3 4	14	1 2 4
5	2 3	10	4	15	1 3

(a) Sample Database



(b) FP-Tree (c) CFP-Tree

Figure 1: FP-Tree and CFP-Tree

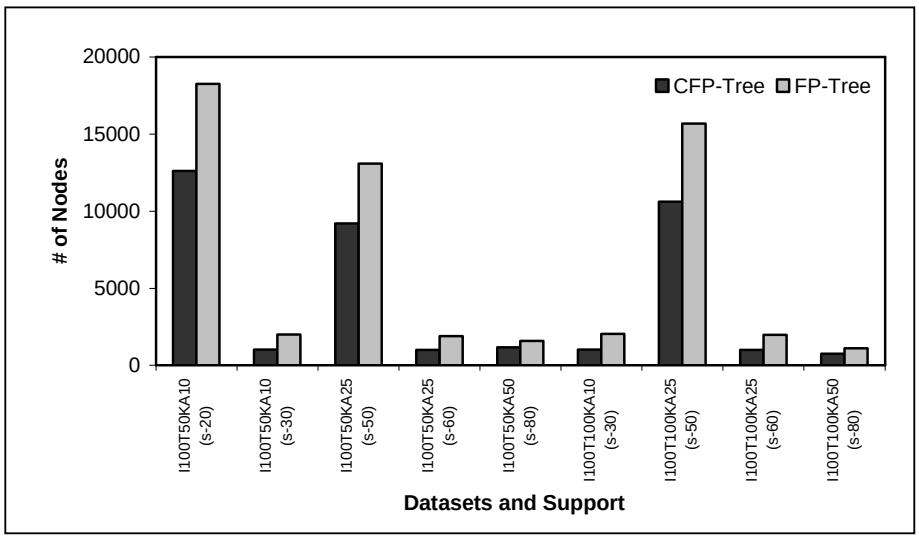


Figure 2: Compactness of CFP-Tree Compared to FP-Tree on Various Synthetic Datasets at Various Support Levels

Algorithm 1 CT-PRO Algorithm: Step 1 and Step 2

input Database *D*, Support Threshold σ
output *CFP-Tree*

```
1   begin
2     // Step 1: Identify frequent items
3     for each transaction t ∈ D
4       for each item i ∈ t
5         if i ∈ ItemTable
6           Increment count of i
7         else
8           Insert i into GlobalItemTable with count = 1
9         end if
10      end for
11    end for
12    Sort GlobalItemTable in
      frequency descending order
13    Assign an index for each frequent item in the
      GlobalItemTable
14    // Step 2: Construct CFP-Tree
15    Construct the left most branch of the tree
16    for each transaction t ∈ D
17      Initialize mappedTrans
18      for each frequent item i ∈ t
19        mappedTrans = mappedTrans ∪ GetIndex(i)
20      end for
21      Sort mappedTrans in ascending order of item ids
22      InsertToCFPTree(mappedTrans)
23    end for
24  end
25  Procedure InsertToCFPTree(mappedTrans)
26    firstItem := mappedTrans[1]
27    currNode := root of subtree pointed by
      ItemTable[firstItem]
28    for each subsequent item i ∈ mappedTrans
29      if currNode has child represent i
30        Increment count[firstItem-1] of the child node
31      else
32        Create child node and set its
          count[firstItem-1] to 1
33        Link the node to its respective node-link
34      end if
35    end for
36  end
```

Suppose the user wants to mine all frequent itemsets from the transaction database shown in Figure 3a with a support threshold of two transactions (or 40%). First, we need to identify frequent items by reading the database once (lines 3-11). The frequent items are stored in frequency descending order in the *GlobalItemTable* (line 12). In a second pass over the database, only frequent items are selected from each transaction (see Figure 3b), mapped to their index id in *GlobalItemTable* on-the-fly, sorted in ascending order of their index id (see Figure 3c) and inserted into the *CFP-Tree* (see Figure 3d). The pointer in *GlobalItemTable* also acts as the start of the links to other nodes with the same item ids (indicated by the dashed lines in Figure 3d). For illustration, at each node we also show the index of the array, the transaction represented at each index entry and its count. In the implementation of *CFP-Tree*, however, only the second column that represents the count is stored.

Tid	Items
1	3 4 5 6 7 9
2	1 3 4 5 13
3	1 2 4 5 7 11
4	1 3 4 8
5	1 3 4 10

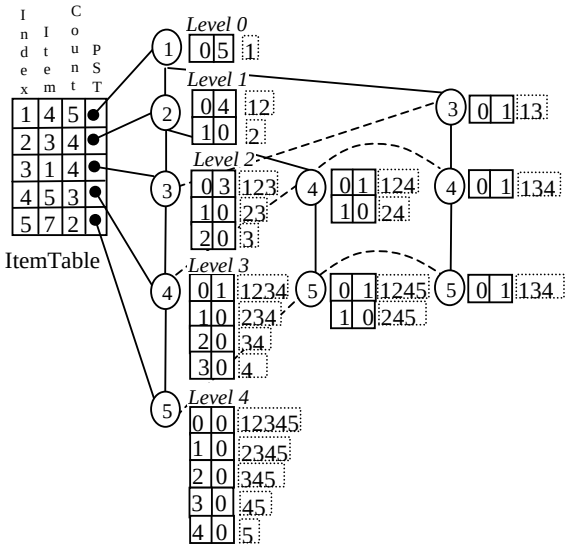
(a) Sample Database

Tid	Items
1	3 4 5 7
2	1 3 4 5
3	1 4 5 7
4	1 3 4
5	1 3 4

(b) Frequent Items

Tid	Items
1	1 2 4 5
2	1 2 3 4
3	1 3 4 5
4	1 2 3
5	1 2 3

(c) Mapped



(d) Global CFP-Tree

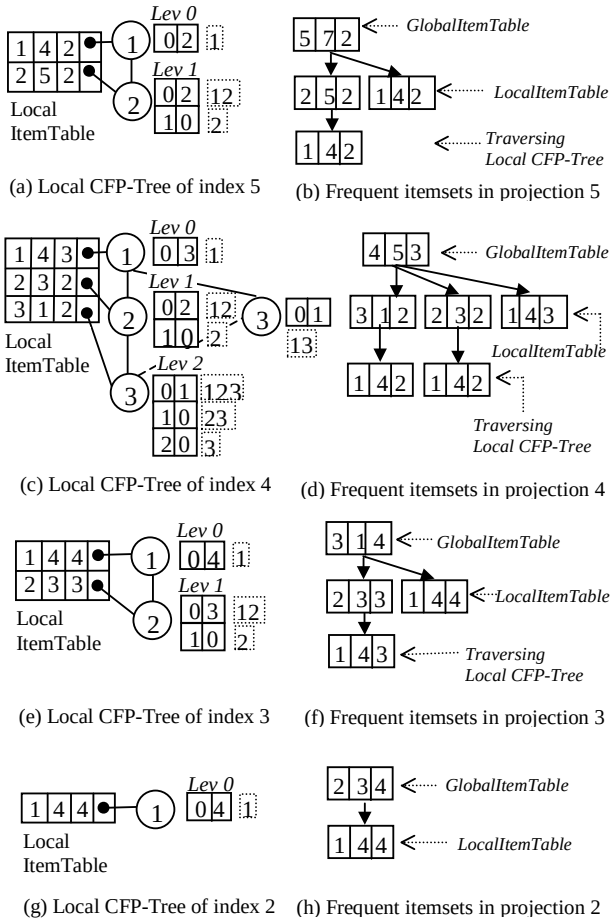
Figure 3: CFP-Tree for the Sample Dataset

The mining process in *CT-PRO* is shown in Algorithm 2 and illustrated by the following example.

Example 1. Let the *CFP-Tree*, as shown in Figure 3d, be the input for the mining step in *CT-PRO* and suppose the user wants to get all the frequent itemsets with minimum support of two transactions (or 40%).

Figure 4 shows the *LocalCFP-Tree* and *LocalFrequentPatternTree* at each step during the mining process. *CT-PRO* starts from the least frequent item (index: 5, item: 7) in the *GlobalItemTable* (line 2). Item 7 is frequent and it will be the root of the *LocalFrequentPatternTree* (line 3). Then *CT-PRO* creates a projection of all transactions ending with index 5. This projection is represented by a *LocalCFP-Tree* and only contains *locally frequent items*. Traversing the *node-link* of index 5 in the *GlobalCFP-Tree* identifies the local frequent items that occur together with it. There are three

nodes of index 5 and the path to the root for each node is traversed counting the other indexes that occur together with index 5 (lines 13-23). In all, we have 1 (2), 2 (1), 3 (1) and 4 (2) for index 5. As indexes 1,4 (item id: 4,5) are *locally frequent*, they are registered in the *LocalItemTable* and assigned new index ids (see Figure 4a). They also become the child of the *LocalFrequentPatternTree*'s root (lines 5-7). Together, the root and its children form frequent itemsets with length two.



(a) Local CFP-Tree of index 5 (b) Frequent itemsets in projection 5

(c) Local CFP-Tree of index 4 (d) Frequent itemsets in projection 4

(e) Local CFP-Tree of index 3 (f) Frequent itemsets in projection 3

(g) Local CFP-Tree of index 2 (h) Frequent itemsets in projection 2

Figure 4: Local CFP-Tree during Mining Process

After *local frequent items* for the projection have been identified, the *node-link* in the *GlobalCFP-Tree* is re-traversed and the path to the root from each node is revisited to get the *local frequent items* occurring together with index 5 in the transactions. These *local frequent items* are mapped to their index in the *LocalItemTable* on-the-fly, sorted in ascending order of their index id and inserted into the *LocalCFP-Tree* (lines 24-33). The first path of index 5 returns nothing. From the second path of index 5, a transaction 14 (1) is inserted into the *LocalCFP-Tree* and another transaction 14 (1) from the third path of index 5 also is inserted. In total, there are two

Algorithm 2 CT-PRO Algorithm: Mining Part

input *CFP-Tree*
output Frequent Itemsets *FP*

```
1   Procedure Mining
2     for each frequent item i ∈ GlobalItemTable
      from the least to the most frequent
3       Initialize LocalFrequentPatternTree
        with i as the root
4       ConstructLocalItemTable(x)
5       for each frequent item j ∈ LocalItemTable
6         Attach i as a child of x
7       end for
8       ConstructLocalCFPTree(x)
9       RecMine(x)
10      Traverse the LocalFrequentPatternTree
        to print the frequent itemsets
11    end for
12  end
13  Procedure ConstructLocalItemTable(i)
14    for each occurrence of node i in the CFP-Tree
15      for each item j in the path to the root
16          if j ∈ LocalItemTable
17            Increment count of j
18          else
19            Insert j into LocalItemTable with count = 1
20          end if
21      end for
22    end for
23  end
24  Procedure ConstructLocalCFPTree(i)
25    for each occurrence of node i in the CFP-Tree
26      Initialize mappedTrans
27      for each frequent item j ∈ LocalItemTable
        in the path to the root
28        mappedTrans = mappedTrans ∪ GetIndex(j)
29      end for
30      Sort mappedTrans in ascending order of item ids
31      InsertToCFPTree(mappedTrans)
32    end for
33  end
34  Procedure RecMine(x)
35    for each child i of x
36      Set all counts in LocalItemTable to 0
37      for each occurrence of node i in
        the LocalCFPTree
38        for each item j in the path to the root
39            Increment count of j in LocalCFPTree
40        end for
41      end for
42      for each frequent item k ∈ LocalItemTable
43        Attach k as a child of i
44      end for
45      RecMine(i)
46    end for
47  end
```

occurrences of transaction 14. Indexes 1 and 4 in the *GlobalItemTable* represent items 4 and 5 respectively. The indexes of items 4 and 5 in the *LocalItemTable* are 1 and 2 respectively and so the transaction 14 is inserted as transaction 12 in the *LocalCFP-Tree*. As the item index in the *GlobalItemTable* and *LocalItemTable* are different, the item id is always maintained for output purposes.

Longer frequent itemsets, with length greater than two, are extracted by calling the procedure *RecMine* (line 9). For simplicity, we have described this procedure (lines 34-47) using recursion but, in the program, it is implemented as a non-recursive procedure. Starting from the least frequent item in the *LocalItemTable*, (line 35), the *node-link* is traversed (lines 37-41). For each node, the path to the root in the *LocalCFP-Tree* is traversed counting the other items that are together with the current item. For example, in Figure 4a, traversing the *node-link* of node 2 will return the index 1 (2) and, since it is frequent, an entry is created and attached as the child of index 2 in the *LocalFrequentPatternTree* (lines 42-44). All frequent itemsets containing item 7 can be extracted by traversing the *LocalFrequentPatternTree* (line 10): 7 (2), 75 (2), 754 (2), 74 (2).

The process is continued to mine the next item in the *GlobalItemTable* in the *GlobalCFP-Tree* with indexes 4, 3, 2 and finally, when the mining process reaches the root of the tree of Figure 3d, it outputs 4 (5).

One major advantage of *CT-PRO* compared to *FP-Growth* is that *CT-PRO* avoids the cost of creating *conditional FP-Trees*. *FP-Growth* needs to create a *conditional FP-Tree* at each step of its recursive mining. This overhead adversely affects its performance, as the number of *conditional FP-Trees* corresponds to the number of frequent itemsets. In *CT-PRO*, for each frequent item (not frequent itemsets), only one *LocalCFP-Tree* is created and traversed non-recursively to extract all frequent itemsets beginning with the frequent item.

4. Time Complexity

In this section, the best-case and worst-case time complexity of *CT-PRO* algorithm is presented. Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of all n items, let transaction database D be $\{t_1, t_2, \ldots, t_m\}$, and let v be the total number of items in all transactions.

Lemma 2. In the best-case, the cost of generating frequent itemsets is $O(v + n)$.

Proof. *The best-case for the CT-PRO algorithm occurs when there is no frequent item. The algorithm has to read v items in all transactions and add the count of n items. The count of all n items are stored in the ItemTable and checked to determine whether there is any frequent item or not. If there is no frequent item, the process stops.* ■

Lemma 3. In the worst-case, the cost of generating frequent itemsets is

$$(v + n) + (v + 2^{n-1}) + \sum_{k=2}^{n} ((2^{(n-k)}-1)(2^{(n-k)}) + 2^{(k-2)}) = O(2^{2n}).$$

Proof. *The worst-case happens when all n items are frequent and all combinations of them are present in m transactions. CT-PRO has three steps: finding frequent items, constructing the CFP-Tree, and mining. The cost of finding frequent items has been provided by Lemma 2. The worst case for the GlobalCFP-Tree corresponds to a situation where all the possible paths exist. In constructing the GlobalCFP-Tree, all the transactions in the database are read (the cost is v) and inserted into the tree (the total number of nodes is* 2^{n-1}*). For the mining process, for each frequent item* f_k *where* $2 \le k \le n$, $2^{(k-2)}$ *nodes in the GlobalCFP-Tree are visited to construct a LocalCFP-Tree. The LocalCFP-Tree has* $(2^{(n-k)}-1)$ *paths that correspond to, at most,* $2^{(n-k)}$ *candidate itemsets. So the worst case mining cost is:*

$$\sum_{k=2}^{n} ((2^{(n-k)}-1)(2^{(n-k)}) + 2^{(k-2)}).$$

Therefore, the total worst-case cost of CT-PRO is

$$(v+n)+(v+2^{n-1})+\sum_{k=2}^{n} ((2^{(n-k)}-1)(2^{(n-k)}) + 2^{(k-2)}) = O(2^{2n})$$ ■

5. Experimental Evaluation

This section contains three sub-sections. In Section 5.1, we compare *CT-PRO* against other well-known algorithms including with *Apriori* [14, 15], *FP-Growth* [10] and recently proposed *OpportuneProject* (OP) [11] on the various datasets available at FIMI Repository 2003 [17]. In Section 5.2, we report the result of more comprehensive testing to determine the feasible performance range of the algorithms. Finally, in Section 5.3, we compare *CT-PRO* with the two best algorithms in the FIMI Repository 2003 [17], *LCM* [18] and *kDCI* [19].

5.1. Comparison with Apriori, FP-Growth and OpportuneProject

Six real datasets are used in this experiment including two dense datasets: *Chess* and *Connect4*; two less dense datasets: *Mushroom* and *Pumsb**; and two sparse datasets: *BMS-WebView-1* and *BMS-WebView-2*. The first four datasets are originally taken from UCI ML Repository [16] and the last two datasets are donated by Blue Martini Software [3]. All the datasets are also available at FIMI Repository 2003 [17].

We used the implementation of *Apriori* created by Christian Borgelt [21] by enabling the use of the prefix

tree data structure. As for *FP-Growth*, we used the executable code available from its authors [10]. However, for comparing the number of nodes of *FP-Tree* to our proposed data structure, we modified the source code of *FP-Growth* provided by Bart Goethals in [22]. For *OpportuneProject* (*OP*), we used the executable code available from its author, Junqiang Liu [11].

All the algorithm were implemented and compiled using MS Visual C++ 6.0. All the experiments (except comparisons with algorithms in the FIMI Repository 2003 website [17]) were performed on a Pentium III PC 866 MHz with 512 MB RAM and 110 GB Hard Disk running on MS Windows 2000. All the reported runtime used in our charts is the total execution time, the period between input and output. It also includes the time of constructing all the data structures used in all programs.

Figure 5 shows the results of the experiment on various datasets. All the charts use a logarithmic scale for run time along the *y*-axis on the left of the chart. We did not plot the results in the chart if the runtime was more than 10,000 seconds. For a comprehensive evaluation of the algorithm's performance, rather than showing where our algorithm performed best at some of the support levels, all the algorithms were extensively tested on various datasets with a support level of 10% to 90% for dense datasets (e.g. *Connect4*, *Chess*, *Pumsb**, *Mushroom*), a support level of 0.1% to 1% for the sparse dataset *BMS-WebView-1*, and a support level of 0.01% to 0.1% for the sparse dataset *BMS-WebView-2*. As the average number of items increases and/or the support level decreases, at some point, every algorithm 'hits the wall' (i.e. takes too long to complete).

CT-PRO outperforms others at *all* support thresholds on the *Connect4*, *Chess*, *Mushroom* and *Pumsb** datasets. On the sparse dataset *BMS-WebView-1*, *CT-PRO* is a runner-up, after *OP*, with only small performance differences (0.4 seconds to 0.49 seconds at a support level of 0.1% and 5.18 seconds to 7.69 seconds at 0.06%). Below the support level of 0.06%, none of the algorithms could mine the *BMS-WebView-1* dataset. On the sparse dataset *BMS-WebView-2*, a remarkable result is obtained. *Apriori*, which is known as a traditional FIM algorithm, outperforms *FP-Growth* at all support levels. *CT-PRO* is the fastest from a support threshold of 1% down to 0.4% and becomes the runner-up, after *OP*, at a support level of 0.3% down to 0.1% with small performance differences.

From these results, we can claim that, on dense datasets, *CT-PRO* generally outperforms others. On sparse datasets, the high cost of the tree construction reduces *CT-PRO* to runner-up. However, as the gap is very small, we can say that *CT-PRO* also works well for sparse datasets.

5.2. Determining the Feasible Performance Range

As mentioned earlier, sample datasets such as real-word *BMS* datasets [3] and the *UCI Machine Learning Repository* [16], which also are available at the FIMI Repository 2003 [17], have their own static characteristics and thus do not cover the full range of densities. An algorithm that works well for one dataset may not have the same degree of performance on other datasets with different dimensions. Dimensions, here, could be the number of transactions, number of items, average number of items per transaction, denseness or sparseness, etc. In this section, a more comprehensive evaluation of the performance of various algorithms is presented.

We generated ten datasets using the synthetic data generator [13]. The first five datasets contained 100 items with 50,000 transactions, and an average number of items per transaction of 10, 25, 50, 75, and 100. The second five datasets contained 100 items with 100,000 transactions, also with an average number of items per transaction of 10, 25, 50, 75, and 100. *CT-PRO*, *Apriori*, *FP-Growth* and *OP* were tested extensively on these datasets at a support level of 10% to 90%, in increments of 10%.

Figure 6 shows the performance comparisons of the algorithms on various datasets. The dataset name shows its characteristics. For example, *I100T100KA10* means there are 100 items, and 100,000 transactions with an average of 10 items per transaction. The experimental results show that the performance characteristics on databases of 50,000 to 100,000 transactions are quite similar. However, the runtime increases with the number of transactions.

The *Apriori* algorithm is very feasible for sparse datasets (with an average number of items in each transaction of 10 and 25). Its performance is good, as it consistently performs better than *FP-Growth* at all support levels. Although *Apriori* is slower than *CT-PRO* and *OP* using the two sparse datasets, its runtime is still acceptable to the user. (It needs only 60 seconds to mine the *I100T50KA25* dataset at the support level of 10%). However, on the datasets with an average number of items per transaction of 50, 75, and 100, *Apriori* performs worst and it can only mine down to a support level of 30%, 50%, and 70% respectively. These results confirm that, for dense datasets, if the support levels used are low, *Apriori* is infeasible.

FP-Growth performs worst at all support levels on the datasets with a low average number of items per transaction (i.e. 10 and 25). The fact that *FP-Growth* does not outperform *Apriori* on these two datasets shows that *Apriori* is more feasible than *FP-Growth* for sparse datasets. However, *FP-Growth* performs significantly better than *Apriori* for the larger average number of items in transactions.

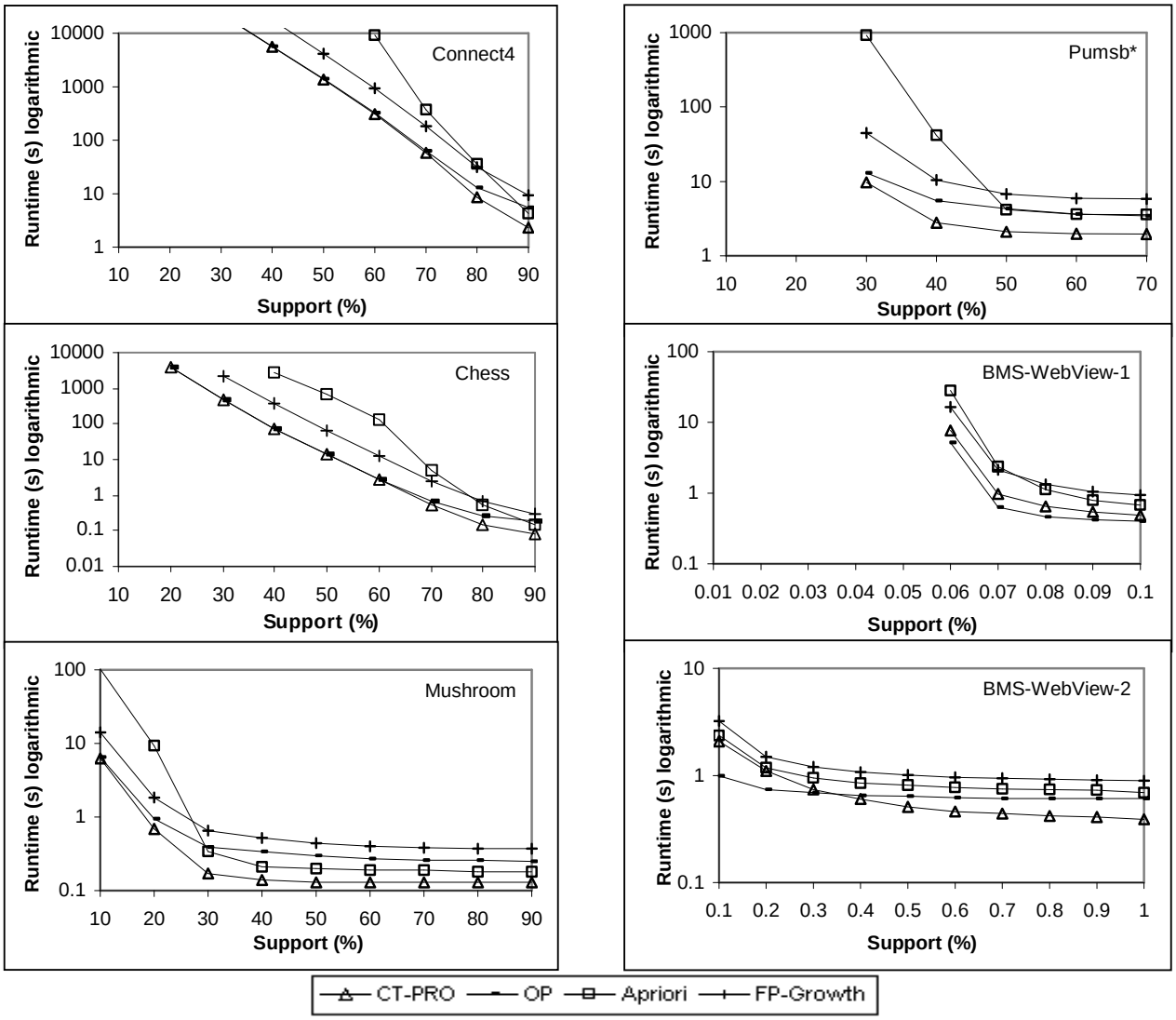


Figure 5: Performance Evaluation of *CT-PRO* Against Others on Various Datasets

Both *CT-PRO* and *OP* have larger feasible performance ranges compared to the other algorithms. *OP* does not perform well on the sparse datasets *I100T50KA10* and *I100T100KA10*. Its performance was even worse than *Apriori* on this dataset. However, it performs better than *Apriori* and *FP-Growth* on other datasets. On the datasets with an average number of items per transaction of 50, 75, and 100, *FP-Growth*, *CT-PRO* and *OP* can mine down to a support level of 20%, 40%, and 50% respectively.

CT-PRO can be considered the best among all other algorithms as it generally performs the best at most support levels. However, as the support level gets lower, its performance is similar to *OP*. Only at a support level of 10%, *OP* occasionally runs slightly faster than *CT-PRO* (e.g. at a support level of 10% on the *I100T50KA25* dataset).

5.3. Comparison with Best Algorithms in the FIMI Repository 2003

For comparison with the best algorithms in the FIMI Repository 2003 [17], we ported our algorithm *CT-PRO* to a Linux operating system and compared it with their two best algorithms: *LCM* [18] and *kDCI* [19]. We performed the experiments on a PC AMD Athlon™ XP 2000+ 1.6 GHz, 1 GB RAM, 2 GB Swap with 40GB Hard Disk running Fedora Core 1. All programs were compiled using g++ compiler.

Figure 7 shows the performance comparisons of algorithms that were submitted to FIMI 2003 on *Chess* and *Connect4* datasets. The figures are taken from [17]. On *Chess* dataset, *kDCI* is the best at a support level of 90% to a support level of 70%. Below that, *LCM* outperforms others. On *Connect4* dataset, at a support

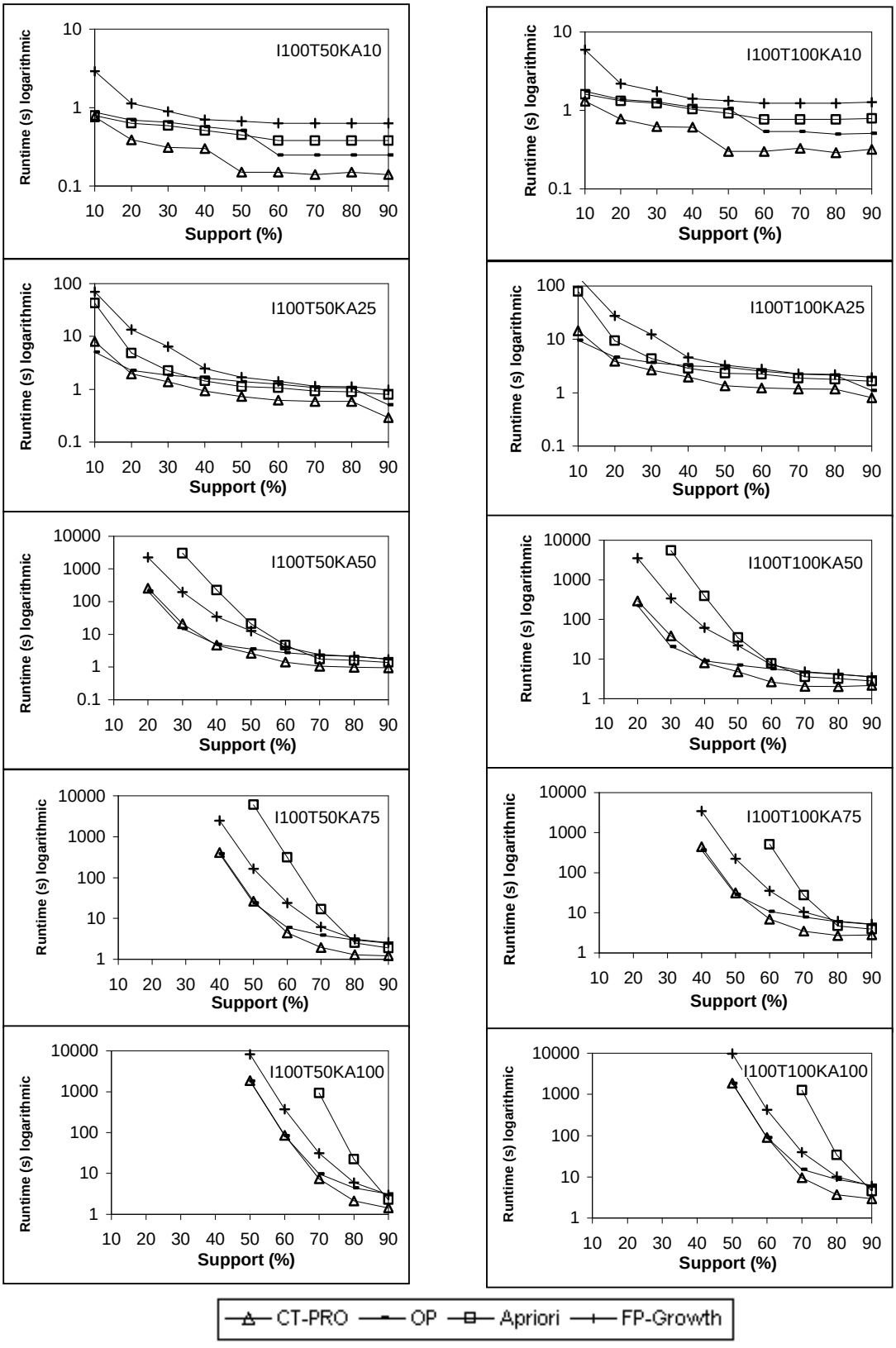


Figure 6: Performance Evaluation on Various Synthetic Datasets

level of 95% to a support level of 70%, *kDCI* is the best. Below that, *LCM* outperforms others. We can conclude that for higher support levels, *kDCI* is the best, but for lower support levels, *LCM* is the best. These best two algorithms are compared with *CT-PRO*.

The *kDCI* algorithm [19] is a multiple heuristics hybrid algorithm that able to adapt its behaviour during the execution. It is an extension of the *DCI* (Direct Count and Intersect) algorithm [23] by adding its adaptability to the dataset specific features. *kDCI* is also a resource aware algorithm which can decides mining strategy based on the hardware characteristics of the computing platform used. Moreover, *kDCI* also used counting inference strategy which originally proposed in [24].

The *LCM* (*Linear time Closed itemset Miner*) algorithm [18] uses the parent-child relationship defined on frequent itemsets. The search tree technique is adapted from the algorithms for generating maximal bipartite cliques [25, 26] based on reverse search [27, 28]. In enumerating all frequent itemsets, *LCM* uses hybrid techniques involving occurrence deliver or diffsets [29] according to the density of the database.

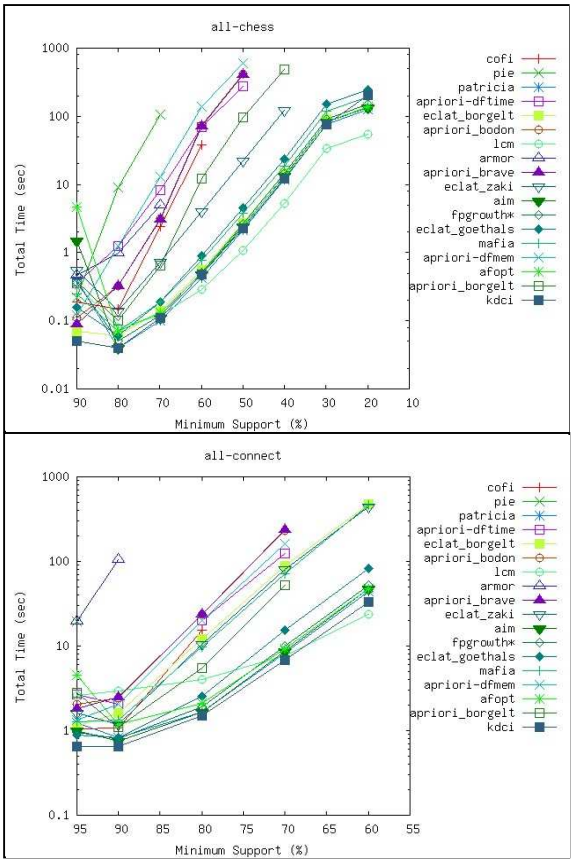


Figure 7: Performance Comparisons of Algorithms available in the FIMI 2003 Repository on *Chess* and *Connect4* Datasets [17]

Figure 8 shows the performance comparisons on *Chess* and *Connect4* datasets. From these results, *CT-PRO* always outperforms others at high support levels. For lower support levels, the performances of these three algorithms are similar. Since *LCM* and *kDCI* are the best algorithms in FIMI Repository 2003 on *Chess* and *Connect4* datasets, we can conclude that *CT-PRO* outperforms *all* other algorithms available in FIMI Repository 2003 [17] on *Chess* and *Connect4* datasets.

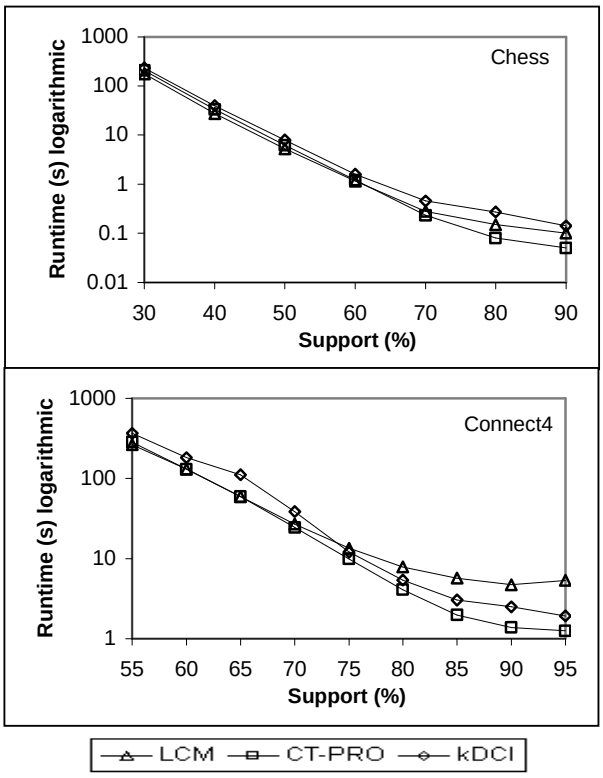


Figure 8: Performance Comparisons of *CT-PRO, LCM* and *kDCI* on *Chess* and *Connect4* Datasets

6. Conclusions

In this paper, we have described a new tree-based data structure named *CFP-Tree* that is more compact than *FP-Tree* used in *FP*-Growth algorithm. Depending on the database characteristics, the number of nodes in an *FP-Tree* could be up to twice as many as in the corresponding *CFP*-Tree for a given database. *CFP-Tree* is used in our new algorithm named *CT-PRO* for mining all frequent itemsets. *CT-PRO divides* the *CFP-Tree* into several projections represented also by *CFP-Trees*. Then *CT-PRO conquers* the *CFP-Tree* for mining all frequent itemsets in each projection.

CT-PRO was explained in detail using a running example and the best-case and worst-case time complexity of the algorithm also was presented. Performance

comparisons of *CT-PRO* against other well-known algorithms, including *Apriori* [14, 15], *FP-Growth* [10] and *OpportuneProject* (*OP*) [11] also were reported. The results show that *CT-PRO* outperforms other algorithms at all support levels on dense datasets and also works well on sparse datasets.

Extensive experiments to measure the feasible performance range of the algorithms are also presented in this paper. A synthetic data generator is used to generate several datasets with varying number of both transactions and average number of items per transaction. Then the best available algorithms including *CT-PRO*, *Apriori*, *FP-Growth* and *OP* are tested on those datasets. The result shows that *CT-PRO* generally outperforms others.

In addition, to relate our research to the last workshop on frequent itemset mining implementations [17], we selected two best algorithms (*LCM* and *kDCI*) from FIMI Repository 2003 and compared their performance with *CT-PRO*. It was shown that *CT-PRO* performed better than the others.

7. Acknowledgement

We are very grateful to Jian Pei for providing the executable code of *FP-Growth*, Bart Goethals for providing his *FP-Growth* program, Christian Borgelt for the *Apriori* program, and Junqiang Liu for the *OpportuneProject* program.

8. References

[1] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases", Proceedings of ACM SIGMOD International Conference on Management of Data, Washington DC, 1993, pp. 207-216.

[2] J. Hipp, U. Guntzer, and G. Nakhaeizadeh, "Algorithms for Association Rule Mining - A General Survey and Comparison", *SIGKDD Explorations*, vol. 2, pp. 58-64, July 2000.

[3] Z. Zheng, R. Kohavi, and L. Mason, "Real World Performance of Association Rule Algorithms", Proceedings of the 7th International Conference on Knowledge Discovery and Data Mining (KDD), New York, 2001.

[4] B. Goethals, "Efficient Frequent Pattern Mining", PhD Thesis, University of Limburg, Belgium, 2002.

[5] B. Liu, W. Hsu, and Y. Ma, "Integrating Classification and Association Rule Mining", Proceedings of ACM SIGKDD, New York, NY, 1998.

[6] Y. G. Sucahyo and R. P. Gopalan, "Building a More Accurate Classifier Based on Strong Frequent Patterns", Proceedings of the 17th Australian Joint Conference on Artificial Intelligence, Cairns, Australia, 2004.

[7] K. Wang, X. Chu, and B. Liu, "Clustering Transactions Using Large Items", Proceedings of ACM CIKM, USA, 1999.

[8] R. Agrawal and R. Srikant, "Mining Sequential Patterns", Proceedings of the 11th International Conference on Data Engineering (ICDE), Taipei, Taiwan, 1995.

[9] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, "H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases", Proceedings of the IEEE International Conference on Data Mining (ICDM), San Jose, California, 2001.

[10] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation", Proceedings of the ACM SIGMOD International Conference on Management of Data, Dallas, TX, 2000.

[11] J. Liu, Y. Pan, K. Wang, and J. Han, "Mining Frequent Item Sets by Opportunistic Projection", Proceedings of ACM SIGKDD, Edmonton, Alberta, Canada, 2002.

[12] R. P. Gopalan and Y. G. Sucahyo, "High Performance Frequent Pattern Extraction using Compressed FP-Trees", Proceedings of SIAM International Workshop on High Performance and Distributed Mining (HPDM), Orlando, USA, 2004.

[13] IBM, "Synthetic Data Generation Code for Associations and Sequential Patterns", Intelligent Information Systems, IBM Almaden Research Center, 2002, http://www.almaden.ibm.com/software/quest/Resources/index.shtml.

[14] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules", Proceedings of the 20th International Conference on Very Large Data Bases, Santiago, Chile, 1994.

[15] C. Borgelt, "Efficient Implementations of Apriori and Eclat", Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI), Melbourne, Florida, 2003.

[16] C. L. Blake and C. J. Merz, "UCI repository of machine learning databases", Irvine, CA: University of California, Department of Information and Computer Science, 1998.

[17] FIMI, "FIMI Repository", 2003, http://fimi.cs.helsinki.fi.

[18] T. Uno, T. Asai, Y. Uchida, and H. Arimura, "LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets", Proceedings of the IEEE ICDM Workshop of Frequent Itemset Mining Implementations (FIMI), Melbourne, Florida, 2003.

[19] C. Lucchese, S. Orlando, P. Palmerini, R. Perego, and F. Silvestri, "kDCI: a Multi-Strategy Algorithm for Mining Frequent Sets", Proceedings of the IEEE ICDM Workshop of Frequent Itemset Mining Implementations (FIMI), Melbourne, Florida, 2003.

[20] Y. G. Sucahyo and R. P. Gopalan, "CT-ITL: Efficient Frequent Item Set Mining Using a Compressed Prefix Tree with Pattern Growth", Proceedings of the 14th Australasian Database Conference, Adelaide, Australia, 2003.

[21] C. Borgelt and R. Kruse, "Induction of Association Rules: Apriori Implementation", Proceedings of the 15th Conference on Computational Statistics, Berlin, Germany, 2002.

[22] B. Goethals, "Home page of Bart Goethals", 2003, http://www.cs.helsinki.fi/u/goethals.

[23] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri, "Adaptive and Resource-Aware Mining of Frequent

Sets", Proceedings of the IEEE International Conference on Data Mining (ICDM), Maebashi City, Japan, 2002.

[24] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal, "Mining Frequent Patterns with Counting Inference", *ACM SIGKDD Explorations*, vol. 2, pp. 66-75, December 2000.

[25] T. Uno, "A Practical Fast Algorithm for Enumerating Cliques in Huge Bipartite Graphs and Its Implementation", Proceedings of the 89th Special Interest Group of Algorithms, Information Processing Society, Japan, 2003.

[26] T. Uno, "Fast Algorithms for Enumerating Cliques in Huge Graphs", Research Group of Computation, IEICE, Kyoto University 2003.

[27] D. Avis and K. Fukuda, "Reverse Search for Enumeration", *Discrete Applied Mathematics*, vol. 65, pp. 21-46, 1996.

[28] T. Uno, "A New Approach for Speeding Up Enumeration Algorithms", Proceedings of ISAAC'98, 1998.

[29] M. J. Zaki and C. Hsiao, "CHARM: An Efficient Algorithm for Closed Itemset Mining", Proceedings of SIAM International Conference on Data Mining (SDM), Arlington, VA, 2002.

LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets

Takeaki Uno[1], Masashi Kiyomi[1], Hiroki Arimura[2]
[1] National Institute of Informatics 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430, Japan
e-mail: uno@nii.jp, masashi@grad.nii.ac.jp
[2] Information Science and Technology, Hokkaido University
Kita 14-jo Nishi 9-chome, 060-0814 Sapporo, JAPAN, e-mail: arim@ist.hokudai.ac.jp

Abstract: For a transaction database, a frequent itemset is an itemset included in at least a specified number of transactions. A frequent itemset P is maximal if P is included in no other frequent itemset, and closed if P is included in no other itemset included in the exactly same transactions as P. The problems of finding these frequent itemsets are fundamental in data mining, and from the applications, fast implementations for solving the problems are needed. In this paper, we propose efficient algorithms LCM (Linear time Closed itemset Miner), LCMfreq and LCMmax for these problems. We show the efficiency of our algorithms by computational experiments compared with existing algorithms.

1 Introduction

Frequent item set mining is one of the fundamental problems in data mining and has many applications such as association rule mining, inductive databases, and query expansion. From these applications, fast implementations of frequent itemset mining problems are needed. In this paper, we propose the second versions of LCM, LCMfreq and LCMmax, for enumerating closed, all and maximal frequent itemsets. LCM is an abbreviation of *Linear time Closed item set Miner*.

In FIMI03[7], we proposed the first version of LCM, which is for enumerating frequent closed itemsets. LCM uses *prefix preserving closure extension* (ppc extension in short), which is an extension from a closed itemset to another closed itemset. The extension induces a search tree on the set of frequent closed itemsets, thereby we can completely enumerate closed itemsets without duplications. Generating a ppc extension needs no previously obtained closed itemset. Hence, the memory use of LCM does not depend on the number of frequent closed itemsets, even if there are many frequent closed itemsets.

The time complexity of LCM is theoretically bounded by a linear function in the number of frequent closed itemsets, while the existing algorithms are not. We further developed algorithms for the frequency counting, *occurrence deliver* and *hybrid of diffsets*. They reduce the practical computation time efficiently. Moreover, the framework of LCM is simple. Generating ppc extensions needs no sophisticated data structure such as binary trees. LCM is implemented with only arrays. Therefore, LCM is fast, and outperforms than other algorithms for some sparse datasets.

However, LCM does not have any routine for reducing the database, while many existing algorithms have. Thus, the performance of LCM is not good for dense datasets with large minimum supports, which involve many unnecessary items and transactions. At FIMI03, we also proposed modifications of LCM, LCMfreq and LCMmax, for enumerating all frequent itemsets and maximal frequent itemsets. Although they are fast for some instances, if LCM is not fast for an instance, they are also not fast for the instance. Existing maximal frequent itemset mining algorithms have efficient pruning methods to reduce the number of iterations, while LCMmax does not have. It is also a reason of the slowness of LCMmax.

This paper proposes the second version of LCM algorithms. We added database reduction to LCM, so that problems of dense datasets can be solved in short time. The second version of LCMmax includes a pruning method, thus the computation time is reduced when the number of maximal frequent itemsets is small. We further developed new algorithms for checking the maximality of a frequent itemset and

for taking the closure of an itemset. We compare the performance of LCM algorithms and other algorithms submitted to FIMI03 by computational experiments. In many instances, LCM algorithms perform above other algorithms.

The organization of the paper is as follows. Section 2 introduces preliminaries. The main algorithms and practical techniques of LCM algorithms are described in Section 3. Section 4 shows the results of computational experiments, and Section 5 concludes the paper.

2 Preliminaries

Let $\mathcal{I} = \{1, ..., n\}$ be the set of *items*. A *transaction database* on $\mathcal{I}$ is a set $\mathcal{T} = \{t_1, \ldots, t_m\}$ such that each t_i is included in $\mathcal{I}$. Each t_i is called a *transaction*. We denote by $||\mathcal{T}||$ the sum of sizes of all transactions in $\mathcal{T}$, that is, the size of database $\mathcal{T}$. A set $P \subseteq \mathcal{I}$ is called an *itemset*.

For itemset P, a transaction including P is called an *occurrence* of P. The *denotation* of P, denoted by $\mathcal{T}(P)$ is the set of the occurrences of P. $|\mathcal{T}(P)|$ is called the *frequency* of P, and denoted by $frq(P)$. For given constant θ, called a *minimum support*, itemset P is *frequent* if $frq(P) \geq \theta$. If a frequent itemset P is included in no other frequent itemset, P is called *maximal*. For any itemsets P and Q, $\mathcal{T}(P \cup Q) = \mathcal{T}(P) \cap \mathcal{T}(Q)$ holds, and if $P \subseteq Q$ then $\mathcal{T}(Q) \subseteq \mathcal{T}(P)$. An itemset P is called *closed* if no other itemset Q satisfies $\mathcal{T}(P) = \mathcal{T}(Q), P \subseteq Q$.

Given set $\mathcal{S} \subseteq \mathcal{T}$ of transactions, let $\mathcal{I}(\mathcal{S})$ be the set of items common to all transactions in $\mathcal{S}$, i.e., $\mathcal{I}(\mathcal{S}) = \bigcap_{T \in \mathcal{S}} T$. Then, we define the *closure* of itemset P in $\mathcal{T}$, denoted by $clo(P)$, by $\mathcal{I}(\mathcal{T}(P))(= \bigcap_{t \in \mathcal{T}(P)} t)$. For every pair of itemsets P and Q, the following properties hold[13, 14].

(1) If $P \subseteq Q$, then $clo(P) \subseteq clo(Q)$.
(2) If $\mathcal{T}(P) = \mathcal{T}(Q)$, then $clo(P) = clo(Q)$.
(3) $clo(clo(P)) = clo(P)$.
(4) $clo(P)$ is the unique smallest closed itemset including P.
(5) A itemset P is a closed itemset if and only if $clo(P) = P$.

For itemset P and item $i \in P$, let $P(i) = P \cap \{1, \ldots, i\}$ be the subset of P consisting only of elements no greater than i, called the *i-prefix* of P. An itemset Q is a *closure extension* of an itemset P if $Q = clo(P \cup \{i\})$ holds for some $i \notin P$. If Q is a closure extension of P, then $Q \supset P$, and $frq(Q) < frq(P)$. We call the item with the maximum index in P the *tail* of P, and denote by $tail(P)$.

3 Algorithms for Efficient Enumeration

In this section, we explain the techniques used in the second versions of LCM algorithms. We explain them one-by-one with comparing to the techniques used by the other algorithms, in the following subsections. The new techniques used in the second version are:

3.2. new database reduction (reduce the frequency counting cost)
3.6. database reduction for fast checking closedness
3.8. database reduction for fast checking maximality
3.7. new pruning algorithm for backtracking-based maximal frequent itemset mining.

The techniques also used in the first versions are:

3.4. occurrence deliver (compute frequency in linear time)
3.5. ppc extension (generates closed itemsets with neither memory nor duplication)
3.3. hypercube decomposition (fast enumeration by grouping frequent itemsets by equivalence class).

The techniques used in the existing algorithms and the first version have citations to the previous papers.

3.1 Enumerating Frequent Itemsets

Any itemset included in a frequent itemset is itself frequent. Thereby, the property "frequent" is monotone. From this, we can construct any frequent itemset from the empty set by adding items one-by-one without passing through any infrequent itemset. Roughly speaking, the existing algorithms are classified into two groups, and algorithms in both groups use this property.

The first group is so called *apriori* or *level-by-level* algorithms [1, 2]. Let $\mathcal{D}_k$ be the set of frequent itemsets of size k. Apriori algorithms start with $\mathcal{D}_0$, that is $\{\emptyset\}$, and compute $\mathcal{D}_k$ from $\mathcal{D}_{k-1}$ in the increasing order of k from $k = 1$. Any itemset in $\mathcal{D}_k$ is obtained from an itemset of $\mathcal{D}_{k-1}$ by adding an item. Apriori algorithms add every item to each itemset of $\mathcal{D}_{k-1}$, and choose frequent itemsets among them. If $\mathcal{D}_k = \emptyset$ holds for some k, then $\mathcal{D}_{k'} = \emptyset$ holds for any $k' > k$. Thus, apriori algorithms stop at such k. This is the scheme of apriori algorithms.

The other group is so called *backtracking* algorithms[3, 18, 19]. Backtracking algorithm is based on recursive calls. An iteration of a backtracking algorithm inputs a frequent itemset P, and generates itemsets by adding every items to P. Then, for each itemset being frequent among them, the iteration generates recursive calls with respect to it. To avoid duplications, an iteration of backtracking algorithms adds items with indices larger than the tail of P. We describe the framework of backtracking algorithms as follows.

ALGORITHM BackTracking (P:current solution)
1. **Output** P
2. **For each** $e \in \mathcal{I}, e > tail(P)$ **do**
3. **If** $P \cup \{e\}$ is frequent **then**
call BackTracking ($P \cup \{e\}$)

An execution of backtracking algorithms gives a tree structure such that the vertices of the tree are iterations, and edges connect two iterations if one of the iteration calls the other. If an iteration I recursively calls another iteration I', then we say that I is the parent of I', and I' is a child of I. For an iteration, the itemset received from the parent is called the *current solution*.

Apriori algorithms use much memory for storing $\mathcal{D}_k$ in memory, while backtracking algorithms use less memory since they keep only the current solution. Backtracking algorithms need no computation for maintaining previously obtained itemsets, so the computation time of backtracking algorithms is generally short. However, apriori algorithms have advantages for the frequency counting.

LCM algorithms are based on backtracking algorithms, and use an efficient techniques for the frequency counting, which are occurrence deliver and anytime database reduction described below. Hence, LCM algorithms compute the frequency efficiently without keeping previously obtained itemsets in memory.

3.2 Maintaining Databases

In the existing studies, *database reduction* is said to be important to reduce the computation time. It is to reduce the input database as the following rules:

1. remove each item included in less than θ transactions
2. remove each item included in all transactions
3. merge the identical transactions into one.

Database reduction performs well when the minimum support is large, and many existing algorithms use it. LCM algorithms also use database reduction.

In the existing studies, the input databases are often stored and maintained by using FP-tree (frequent pattern tree), which is a version of prefix tree (trie) [9]. By using FP-tree, we can search specified transactions from the datasets efficiently. FP-tree compresses the common prefix, so we can decrease the memory use. In addition, FP-tree can detect the identical transactions, thus we can merge them into one. This merge accelerates the frequency counting. From these reasons, FP-trees are used in many algorithms and implementations.

Although FP-tree has many good advantages, we do not use it in the implementation of LCM, but use simple arrays. The main reason is that LCM does not have to search transactions in the database. The main operation of LCM is tracing the transactions in the the denotation of the current solution. Thus, we do not need to use sophisticated data structures for searching.

The other reason is the computation time for the initialization. If we use a standard binary tree for implementing FP-tree, the initialization of the input database takes $O(||\mathcal{T}|| + |\mathcal{T}| \log |\mathcal{T}|)$ time. for constructing FP-tree in memory. Compared to this, LCM detects the identical transactions and stores the database in memory within linear time of the database size. This is because that LCM uses radix sort for this task, which sorts the transactions in a lexicographic order in linear time. In general, the datasets of data mining problems have many transactions, and each transaction has few items. Thus, $||\mathcal{T}||$ is usually smaller than $|\mathcal{T}| \log |\mathcal{T}|$, and LCM has an advantage. The constant factors of the computation time of binary tree operations are relatively larger than that of array operations. LCM also has an advantage at this point. Again, we recall that LCM never search the transactions, so each operation required by LCM can be done in constant time.

FP-tree has an advantage in reducing the memory use. This memory reduction can also reduce the computation time of the frequency counting. To check the efficiency of the reduction, we checked the reduction ratio by FP-tree for some datasets examined in FIMI03. The result is shown in Table 1. Each cell shows the ratio of the number of items needed to be stored by arrays and FP-tree. Usually, the input database is reduced in each iteration, hence we sum up the numbers over all iterations to compute the ratio. In the results of our experiments,

the ratio was not greater 3 in many instances. If $|\mathcal{I}|$ is small, $|\mathcal{T}|$ is large, and the dataset has a randomness, such as accidents, the ratio was up to 6. Generally, a binary tree uses memory three times as much as an array. Thus, the performance of FP-tree seems to be not quite good rather than an array in both memory use and computation time, for many datasets.

3.3 Hypercube Decomposition

LCM finds a number of frequent itemsets at once for reducing the computation time[18]. Since the itemsets obtained at once compose a hypercube in the itemset lattice, we call the technique *hypercube decomposition*. For a frequent itemset P, let $H(P)$ be the set of items e satisfying $e > tail(P)$ and $\mathcal{T}(P) = \mathcal{T}(P \cup \{e\})$. Then, for any $Q \subseteq H(P)$, $\mathcal{T}(P \cup Q) = \mathcal{T}(P)$ holds, and $P \cup Q$ is frequent. LCMfreq uses this property. For two itemsets P and $P \cup Q$, we say that P' is between P and $P \cup Q$ if $P \subseteq P' \subseteq P \cup Q$. In the iteration with respect to P, we output all P' between P and $P \cup H(P)$. This saves about $2^{|H(P)|}$ times of the frequency counting.

To avoid duplications, we do not generate recursive calls with respect to items included in $H(P)$. Instead of generating these recursive calls, we output frequent itemsets including items of $H(P)$ in recursive calls with respect to items not included in $H(P)$. When the algorithm generates a recursive call with respect to $e \notin H(P)$, we pass $H(P)$ to it. In the recursive call, we output all itemsets between $P \cup \{e\}$ and $P \cup \{e\} \cup H(P) \cup H(P \cup \{e\})$. Since any itemset Q satisfies $\mathcal{T}(P \cup Q \cup H(P)) = \mathcal{T}(P \cup Q)$, the itemsets output in the recursive calls are frequent. We describe hypercube decomposition as follows.

ALGORITHM HypercubeDecomposition
(P:current solution, S:itemset)
$S' := S \cup H(P)$
Output all itemsets including P
and included in $P \cup S'$
For each item $e \in \mathcal{I} \setminus (P \cup S')$, $e > tail(P)$ **do**
If $P \cup \{e\}$ is frequent **then**
call HypercubeDecomposition $(P \cup \{e\}, S')$
End for

3.4 Frequency Counting

Generally, the most heavy part of the frequent itemset mining is the frequency counting, which is to count the number of transactions including a newly generated itemset. To reduce the computation time, existing algorithms uses *down project*. For an itemset P, down project computes its denotation $\mathcal{T}(P)$ by using two subsets P_1 and P_2 of P. If $P = P_1 \cup P_2$, then $\mathcal{T}(P) = \mathcal{T}(P_1) \cap \mathcal{T}(P_2)$. Under the condition that the items of P_1 and P_2 are sorted by their indices, the intersection can be computed in $O(|\mathcal{T}(P_1)| + |\mathcal{T}(P_2)|)$ time. Down project uses this property, and computes the denotations quickly. Moreover, if $|\mathcal{T}(P_1)| < \theta$ or $|\mathcal{T}(P_2)| < \theta$ holds, we can see that P never be frequent. It also helps to reduce the computation time.

Apriori-type algorithms accelerates the frequency counting by finding a good pair P_1 and P_2 of subsets of P, such that $|\mathcal{T}(P_1)| + |\mathcal{T}(P_2)|$ is small, or either P_1 or P_2 is infrequent. Backtracking algorithm adds an item e to the current solution P in each iteration, and compute its denotation. By using $\mathcal{T}(\{e\})$, the computation time for the frequency counting is reduced to $O(\sum_{e>tail(P)}(|\mathcal{T}(P)| + |\mathcal{T}(\{e\})|))$.

The bitmap method[5] is a technique for speeding up the computation of taking the intersection in down project. It uses a bitmap image (the characteristic vector) of the denotations. To take the intersection, we have to take $O(|\mathcal{T}|)$ time with bitmap. However, a 32bit CPU can take the intersection of 32bits at once, thus roughly speaking the computation time is reduced to 1/32. This method has a disadvantage for sparse datasets, and is not orthogonal to anytime database reduction described in the below. From the results of the experiments in FIMI 03, bitmap method seems to be not good for sparse large datasets.

LCM algorithms use another method for the frequency counting, called *occurrence deliver*[18, 19]. Occurrence deliver computes the denotations of $P \cup \{e\}$ for $e = tail(P) + 1, ..., |\mathcal{I}|$ at once by tracing transactions in $\mathcal{T}(P)$. It use a bucket for each e to be added, and set them to empty set at the beginning. Then, for each transaction $t \in \mathcal{T}(P)$, occurrence deliver inserts t to the bucket of e for each $e \in t, e > tail(P)$. After these insertions, the bucket of e is equal to $\mathcal{T}(P \cup \{e\})$. For each transaction t, occurrence deliver takes $O(|t \cap \{tail(P)+1, ..., |\mathcal{I}|\}|)$ time. Thus, the computation time is $O(\sum_{T \in \mathcal{T}(P)} |T \cap \{tail(P)+1, ..., |\mathcal{I}|\}|) = O(|\mathcal{T}(P)| + \sum_{e>tail(P)} |\mathcal{T}(P \cup \{e\})|)$. This time complexity is smaller than down project. We describe the pseudo code of occurrence deliver in the following.

ALGORITHM OccurrenceDeliver
(T:database, P:itemset)
1. Set $Bucket[e] := \emptyset$ for each item $e > tail(P)$

dataset and minimum support	chess 40%	accidents 30%	BMS-WebView2 0.05%	T40I10D100K 0.1%
reduction factor by FP-tree	2.27	6.01	1.9	1.57
reduction factor by Hypercube decomposition	6.25	1	1.21	1
reduction factor by apriori (best)	1.11	1.34	1.35	2.85

Table 1: Efficiency test of FP-tree, hypercube decomposition, and apriori: the reduction factor of FP-tree is (sum of # of elements in reduced database by LCM) / (sum of # of elements in reduced database by FP-tree), over all iterations, the reduction ratio of hypercube decomposition is the average number of output frequent itemsets in an iteration, and the reduction ratio of apriori is (sum of $\sum_{e>tail(P)} |\mathcal{T}(P \cup \{e\})|$) / (sum of $\sum_{e>F(P)} |\mathcal{T}(P \cup \{e\})|$), over all iterations.

2. **For each** transaction $t \in \mathcal{T}(P)$ **do**
3. **For each** item $e \in t$, $e > tail(P)$ **do**
4. Insert t to $Bucket[e]$
5. **End for**
6. **End for**
7. **Output** $Bucket[e]$ for all $e > tail(P)$

Let $F(P)$ be the set of items e such that $e > tail(P)$ and $P \cup \{e\}$ is frequent. Apriori algorithms have possibility to find out in short time that $P \cup \{e\}$ is infrequent, thus, in the best case, their computation time can be reduced to $O(\sum_{e \in F(P)} |\mathcal{T}(P \cup \{e\})|)$. If $\sum_{e>tail(P), e \notin F(P)} |\mathcal{T}(P \cup \{e\})|$ is large, occurrence deliver will be slow.

To decrease $\sum_{e>tail(P), e \notin F(P)} |\mathcal{T}(P \cup \{e\})|$, LCM algorithms sort indices of items e in the increasing order of $|\mathcal{T}(\{e\})|$. As we can see in Table 1, this sort reduces $\sum_{e>tail(P), e \notin F(P)} |\mathcal{T}(P \cup \{e\})|$ to 1/4 of $\sum_{e>tail(P)} |\mathcal{T}(P \cup \{e\})|$ in many cases. Since apriori algorithms take much time to maintain previously obtained itemsets, the possibility of speeding up by apriori algorithms is not so large.

LCM algorithms further speeds up the frequency counting by iteratively reducing the database. Suppose that an iteration I of a backtracking algorithm receives a frequent itemset P from its parent. Then, in any descendant iteration of I, no item of indices smaller than $tail(P)$ is added. Hence, any such item can be removed from the database while the execution of the descendant iterations. Similarly, the transactions not including P never include the current solution of any descendant iteration, thus such transactions can be removed while the execution of the descendant iterations. Indeed, infrequent items can be removed, and the identical transactions can be merged.

According to this, LCM algorithms recursively reduce the database while the execution of recursive calls. Before the recursive call, LCM algorithms generate a reduced database according to the above discussion, and pass it to the recursive call. We call this technique *anytime database reduction*.

Anytime database reduction reduces the computation time of the iterations located at the lower levels of the recursion tree. In the recursion tree, many iterations are on the lower levels and few iterations are on the upper levels. Thus, anytime database reduction is expected to be efficient. In our experiments, anytime database reduction works quite well. The following table shows the efficiency of anytime database reduction. We sum up over all iterations the sizes of the database received from the parent, in both cases with anytime database reduction and without anytime database reduction. Each cell shows the sum. The reduction ratio is large especially if the dataset is dense and the minimum support is large.

3.5 Prefix Preserving Closure Extension

Many existing algorithms for mining closed itemsets are based on frequent itemset mining. That is, the algorithms enumerate frequent itemsets, and output those being closed. This approach is efficient when the number of frequent itemsets and the number of frequent closed itemsets differ not so much. However, if the difference between them is large, the algorithms generate many non-closed frequent itemsets,

dataset and minimum support	connect 50%	pumsb 60%	BMS-WebView2 0.1%	T40I10D100K 0.03%
Database reduction	188319235	2125460007	2280260	1704927639
Anytime database reduction	538931	7777187	521576	77371534
Reduction factor	349.4	273.2	4.3	22.0

Table 2: Accumulated number of transactions in database in all iterations

thus they will be not efficient. Many pruning methods have been developed for speeding up, however they are not complete. Thus, the computation time is not bounded by a linear function in the number of frequent closed itemsets. There is a possibility of over linear increase of computation time in the number of output.

LCM uses *prefix preserving closure extension* (ppc-extension in short) for generating closed itemsets[18, 19]. For a closed itemset P, we define the closure tail $clo\_tail(P)$ by the item i of the minimum index satisfying $clo(P(i)) = P$. $clo\_tail(P)$ is always included in P. We say that P' is a ppc extension of P if $P' = clo(P \cup \{e\})$ and $P'(e-1) = P(e-1)$ hold for an item $e > clo\_tail(P)$. Let P_0 be the itemset satisfying $\mathcal{T}(P') = \mathcal{T}$. Any closed itemset $P' \neq P_0$ is a ppc extension of another closed itemset P, and such P is unique for P'. Moreover, the frequency of P is strictly larger than P', hence ppc extension induces a rooted tree on frequent closed itemsets. LCM starts from P_0, and finds all frequent closed itemsets in a depth first manner by recursively generating ppc extensions. The proof of ppc extension algorithms are described in [18, 19].

By ppc extension, the time complexity is bounded by a linear function in the number of frequent closed itemsets. Hence, the computation time of LCM never be super linear in the number of frequent closed itemsets.

3.6 Closure Operation

To enumerate closed itemsets, we have to check whether the current solution P is a closed itemset or not. In the existing studies, there are two methods for this task. The first method is to store in memory previously obtained itemsets which are currently maximal among itemsets having the identical denotation. In this method, we find frequent itemsets one-by-one, and store them in memory with removing itemsets included in another itemset having the identical denotation. After finding all frequent itemsets, only closed itemsets remain in memory. We call this *storage method*. The second method is to generate the closure of P. By adding to P all items e such that $frq(P) = frq(P \cup \{e\})$, we can construct the closure of P. We call the second *closure operation*.

LCM uses closure operations for generating ppc extensions. Similar to the frequency counting, we use database reduction for closure operation. Suppose that the current solution is P, the reduced database is composed of transactions $S_1, ..., S_h$, and each S_l is obtained from transactions $T_1^l, ..., T_k^l$ of the original database. For each S_l, we define the *interior intersection* $In(S_l)$ by $\bigcap_{T \in \{T_1^l, ..., T_k^l\}} T$. Here the closure of P is equal to $\bigcap_{S \in \{S_1, ..., S_h\}} In(S)$. Thus, by using interior intersections, we can efficiently construct the closure of P.

When we merge transactions to reduce the database, interior intersections can be updated efficiently, by taking the intersection of their interior intersections. In the same way as the frequency counting, we can remove infrequent items from the interior intersections for more reduction. The computation time for the closure operation in LCM depends on the size of database, but not on the number of previously obtained itemsets. Thus, storage method has advantages if the number of frequent closed itemsets is small. However, for the instances with a lot of frequent closed itemsets, which take long time to be solved, LCM has an advantage.

3.7 Enumerating Maximal Frequent Itemsets

Many existing algorithms for maximal frequent itemset enumeration are based on the enumeration of frequent itemsets. In breadth-first manner or depth-first manner, they enumerate frequent itemsets and output maximal itemsets among them. To reduce the computation time, the algorithms prune the unnecessary itemsets and recursive calls.

Similar to these algorithms, LCMmax enumerates closed itemsets by backtracking, and outputs maximal itemsets among them. It uses a pruning to cut off unnecessary branches of the recursion. The pruning is based on a re-ordering of the indices of items, in each iteration. We explain the re-ordering in the

following.

Let us consider a backtracking algorithm for enumerating frequent itemsets. Let P be the current solution of an iteration of the algorithm. Suppose that P' is a maximal frequent itemset including P. LCMmax puts new indices to items with indices larger than $tail(P)$ so that any item in P' has an index larger than any item not in P'. Note that this re-ordering of indices has no effect to the correctness of the algorithm.

Let $e > tail(P)$ be an item in P', and consider the recursive call with respect to $P \cup \{e\}$. Any frequent itemset $\hat{P}$ found in the recursive call is included in P', since every item having an index larger than e is included in P', and the recursive call adds to P items only of indices larger than e. From this, we can see that by the re-ordering of indices, recursive calls with respect to items in $P' \cap H$ generates no maximal frequent itemset other than P'.

According to this, an iteration of LCMmax chooses an item $e^* \in H$, and generates a recursive call with respect to $P \cup \{e^*\}$ to obtain a maximal frequent itemset P'. Then, re-orders the indices of items other than e^* as the above, and generates recursive calls with respect to each $e > tail(P)$ not included in $P' \cup \{e^*\}$. In this way, we save the computation time for finding P', and by finding a large itemset, increase the efficiency of this approach. In the following, we describe LCMmax.

ALGORITHM LCMmax (P:itemset, H:items to be added)
1. $H' :=$ the set of items e in H s.t. $P \cup \{e\}$ is frequent
2. **If** $H' = \emptyset$ **then**
3. **If** $P \cup \{e\}$ is infrequent for any e **then** **output** P ; **return**
4. **End if**
5. **End if**
6. Choose an item $e^* \in H'$; $H' := H' \setminus \{e^*\}$
7. LCMmax ($P \cup \{e\}$, H')
8. $P' :=$ frequent itemset of the maximum size found in the recursive call in 7
9. **For each** item $e \in H \setminus P'$ **do**
10. $H' := H' \setminus \{e\}$
11. LCMmax ($P \cup \{e\}$, H')
12. **End for**

3.8 Checking Maximality

When LCMmax finds a frequent itemset P, it checks the current solution is maximal or not. We call this operation *maximality check*. Maximality check is a heavy task, thus many existing algorithms avoid it. They store in memory maximal itemsets among previously obtained frequent itemsets, and update them when they find a new itemset. When the algorithms terminate and obtain all frequent itemsets, only maximal frequent itemsets remain in memory. We call this *storage method*. If the number of maximal frequent itemsets is small, storage method is efficient. However, if the number is large, storage method needs much memory. When a frequent itemset is newly found, storage method checks whether the itemset is included in some itemsets in the memory or not. If the number of frequent itemsets is large, the operation takes long time.

To avoid the disadvantage of storage method, LCMmax operates maximality check. LCMmax checks the maximality by finding an item e such that $P \cup \{e\}$ is frequent. If and only if such e exists, P is not maximal. To operate this efficiently, we reduce the database. Let us consider an iteration of LCMmax with respect to a frequent itemset P. LCM algorithms reduce the database by anytime database reduction for the frequency counting. Suppose that the reduced database is composed of transactions $S_1, ..., S_h$, and each S_l is obtained by merging transactions $T_1^l, ..., T_k^l$ of the original database. Let H be the set of items to be added in the iteration. Suppose that we remove all items e from H such that $P \cup \{e\}$ is infrequent. Then, for any l, $T_1^l \cap H = T_2^l \cap H = , ..., = T_k^l \cap H$ holds. For an item e and a transaction S_l, we define the weight $w(e, S_l)$ by the number of transactions in $T_1^l, ..., T_k^l$ including e. Here the frequency of $P \cup \{e\}$ is $\sum_{S \in \{S_1, ..., S_h\}} w(e, S)$. Thus, by using the weights, we can efficiently check the maximality, in linear time of the size of the reduced database.

When we merge transactions to reduce the database, the weights can be updated easily. For each item e, we take the sum of $w(e, S)$ over all transactions S to be merged. In the same way as frequency counting, we can remove infrequent items from the database for maximality checking, for more reduction.

The computation time for maximality check in LCMmax depends on the size of database, but not on the number of previously obtained itemsets. Thus, storage method has advantages if the number of maximal frequent itemsets is small, but for the instances with a lot of maximal frequent itemsets, which take long time to be solved, LCMmax has an advantage.

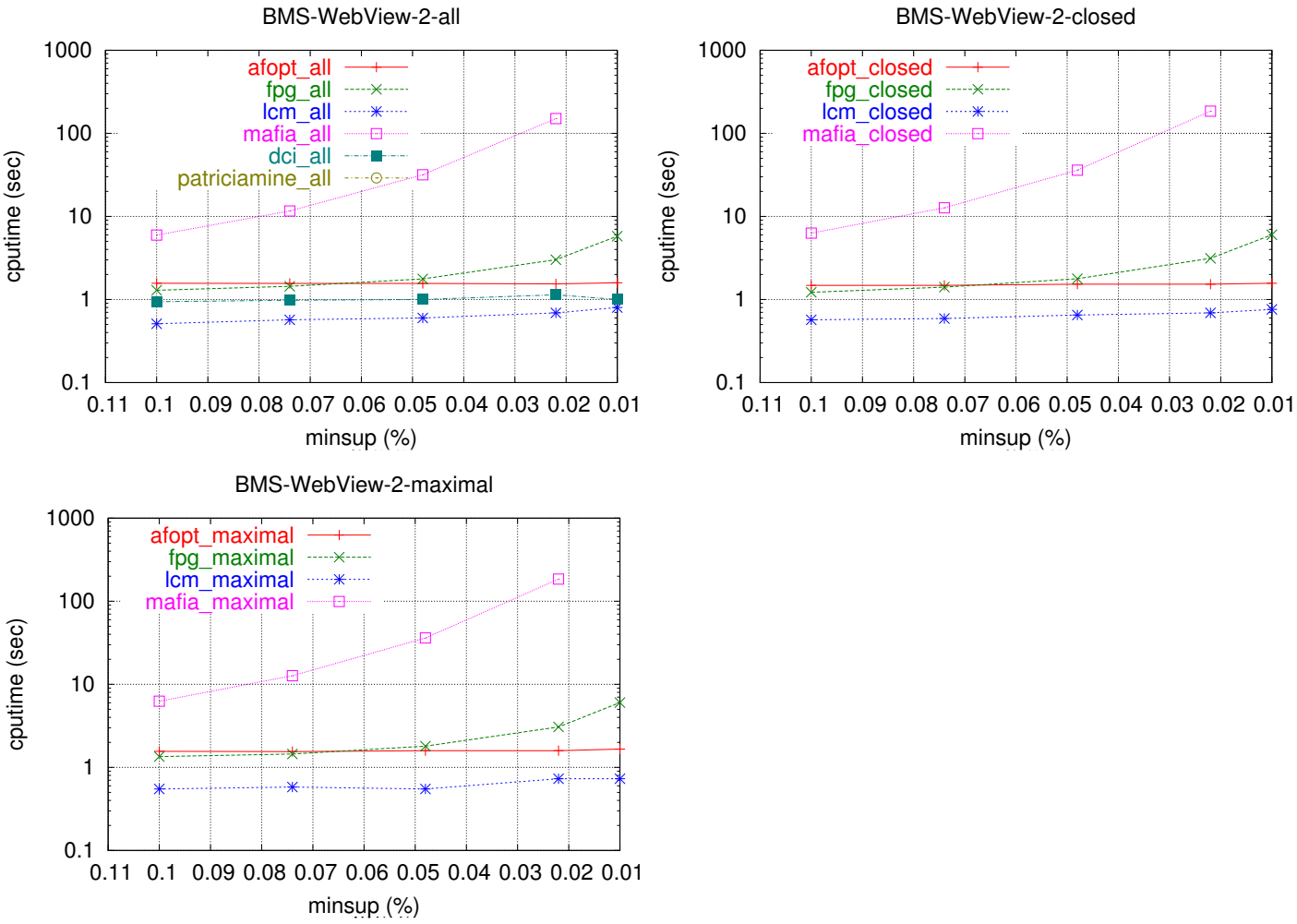


Figure 1: Results 1

4 Computational Experiments

In this section, we show the results of our computational experiments. We implemented our three algorithms LCM, LCMfreq, and LCMmax. They are coded by ANSI C, and complied by gcc. The experiments were executed on a notebook PC, with AMD athron XP 1600+ of 224MB memory. The performance of LCM algorithms are compared with the algorithms which marked good score on FIMI 03: fpgrowth[8], afopt[11], MAFIA[5, 6], kDCI[12], and PATRICIAMINE[16]. We note that kDCI and PATRICIAMINE are only for all frequent itemset mining. To reduce the time for experiments, we stop the execution when an algorithm takes more than 10 minute. The following figures show the results. We do not plot if the computation time is over 10 minutes, or abnormal terminations. The results are displayed in Figure 1 and 2. In each graph, the horizontal axis is the size of minimum supports, and the virtical axis is the CPU time written in a log scale.

From the performances of implementations, the instances were classified into three groups, in which the results are similar. Due to the space limitation, we show one instance as a representative for each group.

The first group is composed of BMS-WebView1, BMS-WebView2, BMS-POS, T10I4D100K, kosarak, and retail. These datasets have many items and transactions but are sparse. We call these datasets *sparse datasets*. We chosen BMS-WebView2 as the representative.

The second group is composed of datasets taken

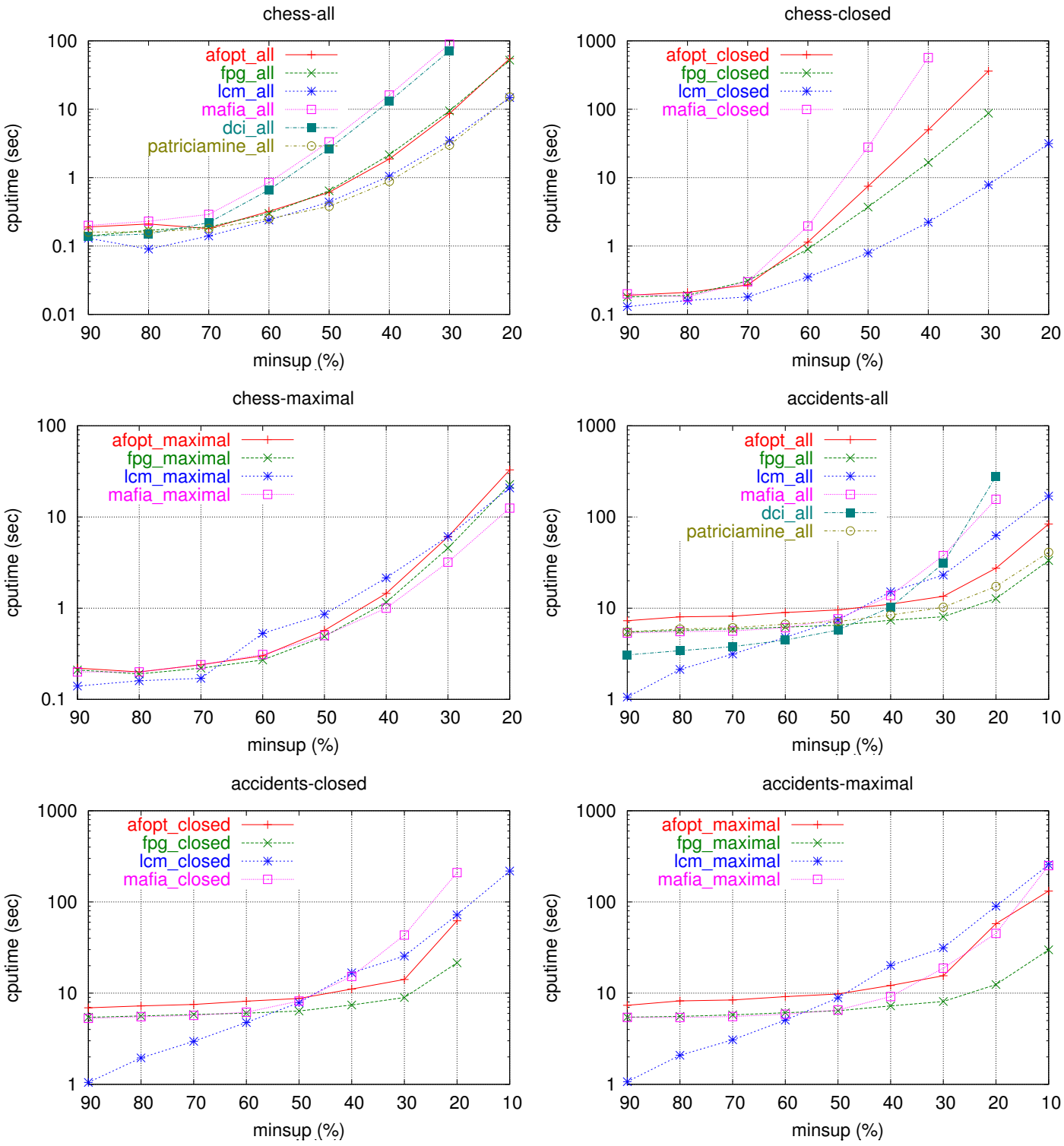


Figure 2: Results 2

from UCI-Machine Learning Repository[1], connect, chess, mushrooms, pumsb, and pumsb-star. These datasets have many transactions but few items. We call these datasets *middle density datasets*. As a representative, we show the result of chess.

The third group is accidents. It is different from any other dataset. It has huge number of transactions, but few items. Transactions includes many items, so the dataset is very dense. We call this dataset *very dense dataset*.

In almost instances and minimum supports, LCM algorithms perform well. When the minimum support is large, LCM algorithms are the fastest for all instances, because of the fast initialization. For all instances with any minimum support, LCM outperforms other closed itemset mining algorithms. This shows the efficiency of ppc extension.

For sparse datasets, LCM algorithms are the fastest, for any minimum support. The efficiency of FP-tree is not large, and occurrence deliver works efficiently. The performances of afopt and fp-growth are quite similar for these problems. They are the second bests, and 2 to 10 times slower than LCM algorithms. For enumerating frequent closed itemsets, they take much time when the number of closed itemsets is large. Although PATRICIAMINE is fast as much as fp-groth and afopt, it abnormally terminated for some instances. kDCI is slow when the number of frequent itemsets is large. MAFIA was the slowest for these instances, for any minimum support.

For middle density datasets, LCM is the fastest for all instances on closed itemset mining. On all and maximal frequent itemset mining, LCMfreq and LCMmax are the fastest for large minimum supports, for any dataset. For small minimum supports, for half instances LCMfreq and LCMmax are the fastest. For the other instances, the results are case by case: each algorithm won in some cases.

For accidents, LCM algorithms are the fastest when the minimum support is large. For small supports, LCM(closed) is the fastest, however LCMfreq and LCMmax are slower than fp-growth For this dataset, the efficiency of FP-tree is large, and the compression ratio is up to 6. Bitmap is also efficient from the density. Hence, the computation time for the frequency counting is short in the execution of existing implementations. However, by ppc extension, LCM has an advantage for closed itemset mining. hence LCM(closed) is the fastest.

[1] `http://www.ics.uci.edu/ mlearn/MLRepository.html`

5 Conclusion

In this paper, we proposed a fast implementation of LCM for enumerating frequent closed itemsets, which is based on prefix preserving closure extension. We further gave implementations LCMfreq and LCMmax for enumerating all frequent itemsets and maximal frequent itemsets by modifying LCM. We show by computational experiments that our implements of LCM, LCMfreq and LCMmax perform above the other algorithms for many datasets, especially for sparse datasets. There is a possibility of speeding up LCM algorithms by developing more efficient maximality checking algorithms, or developing a hybrid of array and FP-tree like data structures.

Acknowledgment

This research is supported by joint-research funds of National Institute of Informatics.

References

[1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," *In Proceedings of VLDB '94,* pp. 487–499, 1994.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen and A. I. Verkamo, "Fast Discovery of Association Rules," *In Advances in Knowledge Discovery and Data Mining,* MIT Press, pp. 307–328, 1996.

[3] R. J. Bayardo Jr., "Efficiently Mining Long Patterns from Databases", In *Proc. SIGMOD'98*, pp. 85–93, 1998.

[4] E. Boros, V. Gurvich, L. Khachiyan, and K. Makino, "On the Complexity of Generating Maximal Frequent and Minimal Infrequent Sets," *STACS 2002*, pp. 133-141, 2002.

[5] D. Burdick, M. Calimlim, J. Gehrke, "MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases," In *Proc. ICDE 2001*, pp. 443-452, 2001.

[6] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, and T. Yiu, "MAFIA: A Performance Study of Mining Maximal Frequent Itemsets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003. (Available as CEUR Workshop Proc. series, Vol. 90, http://ceur-ws.org/vol-90)

[7] B. Goethals, *the FIMI'03 Homepage*, `http://fimi.cs.helsinki.fi/`, 2003.

[8] G. Grahne and J. Zhu, "Efficiently Using Prefix-trees in Mining Frequent Itemsets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003. (Available as CEUR Workshop Proc. series, Vol. 90, http://ceur-ws.org/vol-90)

[9] J. Han, J. Pei, Y. Yin, "Mining Frequent Patterns without Candidate Generation," *SIGMOD Conference 2000*, pp. 1-12, 2000

[10] R. Kohavi, C. E. Brodley, B. Frasca, L. Mason and Z. Zheng, "KDD-Cup 2000 Organizers' Report: Peeling the Onion," *SIGKDD Explorations,* 2(2), pp. 86-98, 2000.

[11] Guimei Liu, Hongjun Lu, Jeffrey Xu Yu, Wang Wei, and Xiangye Xiao, "AFOPT: An Efficient Implementation of Pattern Growth Approach," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003. (Available as CEUR Workshop Proc. series, Vol. 90, http://ceur-ws.org/vol-90)

[12] S. Orlando, C. Lucchese, P. Palmerini, R. Perego and F. Silvestri, "kDCI: a Multi-Strategy Algorithm for Mining Frequent Sets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003. (Available as CEUR Workshop Proc. series, Vol. 90, http://ceur-ws.org/vol-90)

[13] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, *Efficient Mining of Association Rules Using Closed Itemset Lattices*, *Inform. Syst.*, 24(1), 25–46, 1999.

[14] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, *Discovering Frequent Closed Itemsets for Association Rules*, In *Proc. ICDT'99*, 398-416, 1999.

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

[16] A. Pietracaprina and D. Zandolin, "Mining Frequent Itemsets using Patricia Tries," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003. (Available as CEUR Workshop Proc. series, Vol. 90, http://ceur-ws.org/vol-90)

[17] S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirakawa, "A New Algorithm for Generating All the Maximum Independent Sets," *SIAM Journal on Computing,* Vol. 6, pp. 505–517, 1977.

[18] T. Uno, T. Asai, Y. Uchida, H. Arimura, "LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003. (Available as CEUR Workshop Proc. series, Vol. 90, http://ceur-ws.org/vol-90)

[19] T. Uno, T. Asai, Y. Uchida, H. Arimura, "An Efficient Algorithm for Enumerating Closed Patterns in Transaction Databases," to appear in Proc. of Discovery Science 2004, 2004.

[20] M. J. Zaki, C. Hsiao, "CHARM: An Efficient Algorithm for Closed Itemset Mining," *2nd SIAM International Conference on Data Mining (SDM'02)*, pp. 457-473, 2002.

[21] Z. Zheng, R. Kohavi and L. Mason, "Real World Performance of Association Rule Algorithms," *KDD 2001*, pp. 401-406, 2000.

AIM2: Improved implementation of AIM

Sagi Shporer
School of Computer Science
Tel-Aviv University
Tel Aviv, Israel
shporer@tau.ac.il

Abstract

We present AIM2-$\mathcal{F}$, an improved implementation of AIM-$\mathcal{F}$ [4] algorithm for mining frequent itemsets. Past studies have proposed various algorithms and techniques for improving the efficiency of the mining task. We have presented AIM-$\mathcal{F}$ at FIMI'03, a combination of some techniques into an algorithm which utilize those techniques dynamically according to the input dataset. The algorithm main features include depth first search with vertical compressed database, diffset, parent equivalence pruning, dynamic reordering and projection. Experimental testing suggests that AIM2-$\mathcal{F}$ outperforms existing algorithm implementations on various datasets.

1. Introduction

Finding association rules is one of the driving applications in data mining, and much research has been done in this field [7, 3, 5]. Using the support-confidence framework, proposed in the seminal paper of [1], the problem is split into two parts — (a) finding frequent itemsets, and (b) generating association rules.

Let I be a set of items. A subset $X \subseteq I$ is called an itemset. Let D be a transactional database, where each transaction $T \in D$ is a subset of $I : T \subseteq I$. For an itemset X, support(X) is defined to be the number of transactions T for which $X \subseteq T$. For a given parameter minsupport, an itemset X is call a *frequent itemset* if support$(X) \geq$ minsupport. The set of all frequent itemsets is denoted by $\mathcal{F}$.

We have presented AIM-$\mathcal{F}$ [4] for mining frequent itemsets. The AIM-$\mathcal{F}$ algorithm build upon several ideas appearing in previous work, a partial list of which is the following: Apriori [2], Lexicographic Trees and Depth First Search Traversal [6], Dynamic Reordering [5], Vertical Bit Vectors [7, 3], Projection [3], Difference sets [9], Dynamic Reordering [5], Parent Equivalence Pruning [3, 8] and Bit-vector projection [3].

High level pseudo code for the AIM-$\mathcal{F}$ algorithm appears in Figure 1.

```
AIM-F(n : node, minsupport : integer)
(1)  t = n.tail
(2)  for each α in t
(3)      Compute s_α = support(n.head ⋃ α)
(4)      if (s_α = support(n.head))
(5)          add α to the list of items removed by PEP
(6)          remove α from t
(7)      else if (s_α < minsupport)
(8)          remove α from t
(9)  Sort items in t by s_α in ascending order.
(10) While t ≠ ∅
(11)     Let α be the first item in t
(12)     remove α from t
(13)     n'.head = n.head ⋃ α
(14)     n'.tail = t
(15)     Report n'.head ⋃{All subsets of items
             removed by PEP} as frequent itemsets
(16)     AIM-F(n')
```

Figure 1. AIM-$\mathcal{F}$

2. Implementation Improvements

We now describe the difference between AIM-$\mathcal{F}$ and AIM2-$\mathcal{F}$ implementations:

- Integer to String conversions - Experiments run time analysis have shown that the conversion of integers to strings is a major CPU consumer. To reduce conversion time two steps are taken:
  - Item name conversion - When printing an itemset all the item names in the itemset are

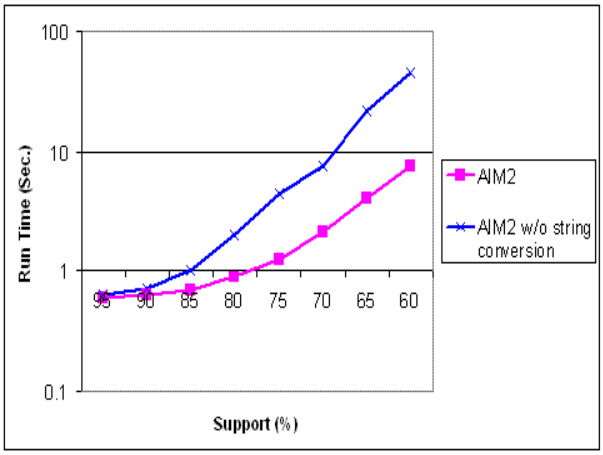


Figure 2. Connect dataset: Testing AIM2-$\mathcal{F}$ with and without the string conversion improvements

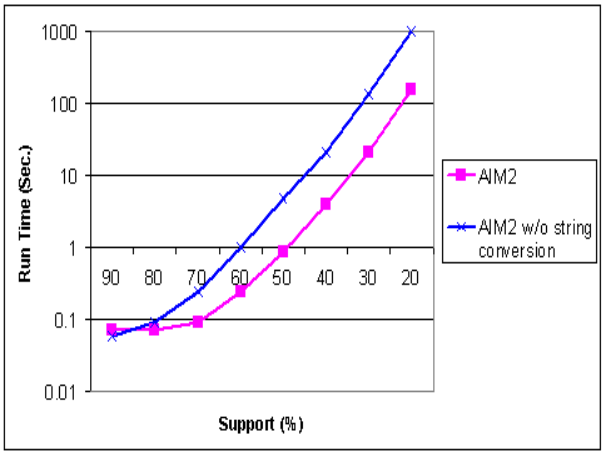


Figure 3. Chess dataset: Testing AIM2-$\mathcal{F}$ with and without the string conversion improvements

printed. In this mining task the items are numbers, and need to be converted to strings. Instead of creating the string every time before printing, the conversion is done once for every item, when the item is loaded during the dataset reading process.

– Support conversion - To print the support it must be converted to a string. To enable fast conversion of the support value to string, a static lookup table from integer to string was added. The lookup table contains the 64K integer values above the *minSupport*. Every entry in the lookup table has the string representation of the entry attached. Every time a support value needs to be converted to string, it is first checked if the value appears in the lookup table, if so, the string is taken from the table, with a very low cost.

In figures 2 and 3 we compare the AIM2-$\mathcal{F}$ algorithm runtime with and without the string conversion improvement. It is clear that this improvement alone contribute up to an order of magnitude improvement. As the size of the input increases (lower support) the contribution of the string conversion improvements increases.

- Late $F2$ matrix construction - The size of the $F2$ matrix is I^2 where I is the number of items. In datasets where the number of items is very large the $F2$ matrix can not be constructed. The improvement in AIM2-$\mathcal{F}$ is that the $F2$ matrix is built only for items for which $support(i) \geq minSupport$. This enables the construction of the $F2$ for larger datasets.

- Input buffer reuse - In AIM-$\mathcal{F}$ the dataset load method allocated an input buffer for every transaction read. Switching to a single input buffer that is re-used for all the transactions reduced the loading time in AIM2-$\mathcal{F}$ by nearly 50%. However the loading time is usually insignificant comparing to the overall runtime (unless the support is very high).

References

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, pages 207–216, 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, pages 487–499, 1994.

[3] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: a maximal frequent itemset algorithm for transactional databases. In *ICDE*, 2001.

[4] A. Fiat and S. Shporer. Aim: Another itemset miner. In *FIMI*, 2003.

[5] R. J. B. Jr. Efficiently mining long patterns from databases. In *SIGMOD*, pages 85–93, 1998.

[6] R. Rymon. Search through systematic set enumeration. In *KR-92*, pages 539–550, 1992.

[7] P. Shenoy, J. R. Haritsa, S. Sundarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *SIGMOD*, 2000.

[8] M. J. Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering*, 12(2):372–390, 2000.

[9] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *KDD*, pages 326–335, 2003.

Reducing the Main Memory Consumptions of FPmax* and FPclose

Gösta Grahne and Jianfei Zhu
Concordia University
Montreal, Canada
{grahne, j_zhu}@cs.concordia.ca

Abstract

In [4], we gave FPgrowth, FPmax* and FPclose for mining all, maximal and closed frequent itemsets, respectively. In this short paper, we describe two approaches for improving the main memory consumptions of FPmax* and FPclose. Experimental results show that the two approaches successfully reduce the main memory requirements of the two algorithms, and that in particular one of the approaches does not incur any practically significant extra running time.*

1. Introduction

In FIMI'03 [2], many implementations of algorithms for mining all, maximal and closed frequent itemsets were submitted and tested independently by the organizers. The experimental results in [3] showed that our algorithms FPgrowth*, FPmax* and FPclose [4] have great performance on most datasets, and that FPmax* and FPclose were among the fastest implementations. Our experimental results in [7] also showed that the three algorithms are among the algorithms that consume the least amount of main memory when running them on dense datasets.

However, we also found in [7] that FPmax* and FPclose require much more main memory than other algorithms in [2] especially when the datasets are sparse. This is because the FP-trees constructed from the sparse datasets sometimes are fairly big, and they are stored in main memory for the entire execution of the algorithms FPmax* and FPclose. The sizes of some auxiliary data structures for storing maximal and closed frequent itemsets, the MFI-trees and the CFI-trees also always increase even though many nodes in the trees become useless.

In this short paper, we describe two approaches for reducing the main memory usages of FPmax* and FPclose. We also give the experimental results which show that FPmax* with either of the approaches needs less main memory for running on both synthetic dataset and real dataset.

2. Improving the Main Memory Requirements

We first give a detailed introduction to the main memory requirements of the two algorithm implementations in [4]. Then two approaches for improving the main memory consumption are introduced. Since FPmax* and FPclose have similar main memory requirements, here we only consider main memory improvements in the implementation of FPmax*.

The Basic Case

In [4], when implementing FPgrowth*, FPmax* and FPclose, each node P of an FP-tree, an MFI-tree or a CFI-tree has 4 pointers that point to its parent node, left-child node, right-sibling node and the next node that corresponds to the same itemname as P. The left-child node and right-sibling node pointers are set for the tree construction. The parent node pointer and next node pointer in an FP-tree are used for finding the conditional pattern base of an item. In MFI-trees and CFI-trees, they are used for maximality checking and closedness testing.

In all algorithms, the FP-tree $T_\emptyset$ constructed from the original database $\mathcal{D}$ is always stored in main memory during the execution of the algorithms. For FPmax*, during the recursive calls, many small FP-trees and MFI-trees will be constructed. The biggest MFI-tree is $M_\emptyset$ whose size increases slowly. At the end of the call of FPmax*($T_\emptyset, M_\emptyset$), $M_\emptyset$ stores all maximal frequent itemsets mined from $\mathcal{D}$.

We can see that the main memory requirement of FPmax* in the basic case is at least the size of $T_\emptyset$ plus the size of $M_\emptyset$ which contains all maximal frequent itemsets in $\mathcal{D}$.

Approach 1: Trimming the FP-trees and MFI-trees Continuously

To see if we can reduce the main memory requirement of FPmax*, let's analyze FPmax* first.

Suppose during the execution of FPmax*, an FP-tree T and its corresponding MFI-tree M are constructed. The

items in T and M are $i_1, i_2, \ldots, i_n$ in decreasing order of their frequency. Note that the header tables of T and M have the same items and item order. Starting from the least frequent item i_n, FPmax* mines maximal frequent itemsets from T. A candidate frequent itemset X is compared with the maximal frequent itemsets in M. If X is maximal, X is inserted into M. When processing the item i_k, FPmax* needs the frequency information that contains only items $i_1, i_2, \ldots, i_{k-1}$, and the frequency information of $i_{k+1}, \ldots, i_n$ will not be used any more. In other words, in T, only the nodes that correspond to $i_1, i_2, \ldots, i_k$ are useful, and the nodes corresponding to $i_{k+1}, \ldots, i_n$ can be deleted from T. If a candidate maximal frequent itemset X is found, X must be a subset of $i_1, i_2, \ldots, i_k$. Thus in M, only the nodes corresponding to $i_1, i_2, \ldots, i_k$ are used for maximality checking, and the nodes corresponding to $i_{k+1}, \ldots, i_n$ will never be used, and therefore can be deleted.

Based on the above analysis, we can reduce the main memory requirement of FPmax* by continuously trimming the FP-trees and MFI-trees. After processing an item i_k, all i_k-nodes in T and M are deleted. This can be done by following the head of the link list from i_k in the header tables $T.header$ and $M.header$. Remember that the children of a node are organized by a right-sibling linked list. To speed up the deletions we make this list doubly linked, i.e. each node has pointers both to its right and left siblings.

Before calling FPmax*, $T_\emptyset$ has to be stored in the main memory. By deleting all nodes that will not be used any more, the sizes of FP-trees, especially the size of $T_\emptyset$, become smaller and smaller. The sizes of the MFI-trees still increase because new nodes for new maximal frequent itemsets are inserted, however, since obsolete nodes are also deleted, the MFI-trees will grow more slowly. At the end of the call of FPmax*, the sizes of $T_\emptyset$ and $M_\emptyset$ are all zero. We assume that the sizes of the recursively constructed FP-trees and MFI-trees are always far smaller than the size of the top-level trees $T_\emptyset$ and $M_\emptyset$, and that the main memory consumption of these trees can be neglected. Besides $T_\emptyset$, the main memory also stores $M_\emptyset$. At the initial call of FPmax*, the size of $M_\emptyset$ is zero. Then $M_\emptyset$ never reaches its full size because of the trimming. We estimate that the average main memory requirement of FPmax* with approach 1 is the size of $T_\emptyset$ plus half of the size of $M_\emptyset$.

In [4], we mentioned that we can allocate a chunk of main memory for an FP-tree, and delete all nodes in the FP-tree at a time by deleting the chunk. Time is saved by avoiding deleting the nodes in the FP-tree one by one. Obviously, this technique can not be used parallel with approach 1. Therefore, FPmax* with approach 1 will be slower than the basic FPmax*, but its peak main memory requirement will be smaller than that of the basic FPmax*.

Approach 2: Trimming the FP-trees and MFI-trees Once

In approach 2, we use the main memory management technique by trimming the FP-trees and MFI-trees only once. We still assume that main memory consumption of the recursively constructed FP-trees and MFI-trees can be neglected, and only the FP-tree $T_\emptyset$ and the MFI-tree $M_\emptyset$ are trimmed.

Suppose the items in $T_\emptyset$ and $M_\emptyset$ are $i_1, i_2, \ldots, i_n$. In our implementation, we allocate a chunk of main memory for those nodes in $T_\emptyset$ and $M_\emptyset$ that correspond to $i_{\lfloor n/2 \rfloor}, \ldots, i_n$. The size of the chunk is changeable. During the execution of FPmax*, $T_\emptyset$ and $M_\emptyset$ are not trimmed until item $i_{\lfloor n/2 \rfloor}$ in $T_\emptyset.header$ is processed. The main memory of the chunk is freed and all notes in the chunk are deleted at that time.

In this approach, before processing $i_{\lfloor n/2 \rfloor}$ and freeing the chunk, $T_\emptyset$ and a partial $M_\emptyset$ are stored in the main memory. On the average, the size of $M_\emptyset$ is half of the size of the full $M_\emptyset$. After freeing the chunk, new nodes for new maximal frequent itemsets are inserted and they are never trimmed. However, considering the fact that MFI-tree structure is a compact data structure, the new nodes are for the $\lfloor n/2 \rfloor$ most frequent items, and $M_\emptyset$ already has many branches for those nodes before trimming, we can expect that the size of $M_\emptyset$ will be a little bit more than half of the size of the complete $M_\emptyset$. Therefore the peak main memory consumption is a little bit more than the size of $T_\emptyset$ plus half of the size of $M_\emptyset$. Compared with approach 1, the FPmax* with approach 2 is faster but consumes somewhat more main memory.

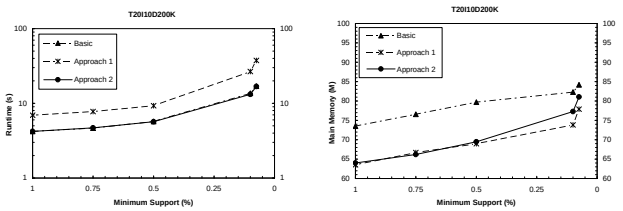
3. Experimental Evaluation

We now present a comparison of the runtime and main memory consumptions of the basic case and the two approaches. We ran the three implementations of FPmax* on many synthetic and real datasets. The synthetic datasets are sparse datasets, and the real datasets are all dense. Due to the lack of space, only the results for one synthetic dataset and one real dataset are shown here.

The synthetic dataset *T20I10D200K* was generated from the application on the website [1]. It contains 200,000 transactions and 1000 items. The real dataset *pumsb** was downloaded from the FIMI'03 website [2]. It was produced from census data of Public Use Microdata Sample (PUMS).

All experiments were performed on a 1GHz Pentium III with 512 MB of memory running RedHat Linux 7.3.

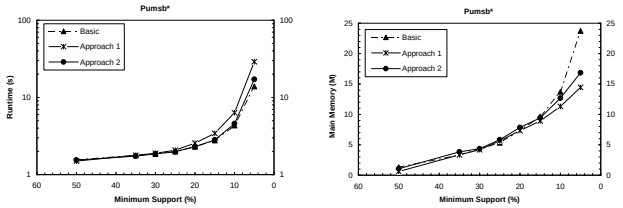
Figure 1 shows the runtime and the main memory usage of running FPmax* with the implementations of the basic case and the two approaches on the dataset *T20I10D200K*. As expected, in the runtime graph, FPmax* with approach 1 took the longest time. Its runtime is almost twice the runtime of the basic case and approach 2. However, approach 1



Runtime Main Memory Consumption

Figure 1. *T20I10*

consumes the least amount of main memory. The peak main memory of approach 1 is always less than the basic case for about 10 megabytes, or about 15%. The speed of approach 2 is similar to that of the basic case, since approach 2 only trims the FP-tree $T_\emptyset$ and the MFI-tree $M_\emptyset$ once. The main memory consumption of approach 2 is similar to that of approach 1, which means the approach 2 successfully saves main memory.



Runtime Main Memory Consumption

Figure 2. *pumsb**

The runtime and main memory usage of running FPmax* on real dataset *pumsb** are shown in Figure 2. The results are similar to those results on synthetic dataset. Dataset *pumsb** is a very dense dataset, its FP-trees and MFI-trees have very good compactness, and there are not many nodes in the trees. Therefore, in the two graphs in Figure 2, the differences of the runtime and main memory consumptions for the basic case and the two approaches are not very big.

4. Conclusions

We have analyzed the main memory requirements of the FPmax* and FPclose implementation in [4]. Two approaches for reducing the main memory requirements of FPmax* and FPclose are introduced. Experimental results show that both approach 1 and approach 2 successfully decrease the main memory requirement of FPmax*. While the continuous trimming of the trees in approach 1 slows down the algorithm, the "one-time-trimming" used in approach 2 shows speed similar to the original method.

We also noticed that the PatriciaMine in [6] using Patricia trie structure to implement the FP-growth method [5] shows great speed and less main memory requirement. We are currently considering implementing FPmax* and FPclose using a Patrica trie.

References

[1] `http://www.almaden.ibm.com/software/quest/Resources/index.shtml`.

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

[3] B. Goethals and M. J. Zaki (Eds.). *Proceedings of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI '03)*. CEUR Workshop Proceedings, Vol 80 `http://CEUR-WS.org/Vol-90`.

[4] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Melbourne, FL, Nov. 2003.

[5] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceeding of Special Interest Group on Management of Data* , pages 1-12, Dallas, TX, May 2000.

[6] A. Pietracaprina and D. Zandolin. Mining frequent itemsets using Patricia tries. In *Proceedings of the 1st Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Melbourne, FL, Nov. 2003.

[7] J. Zhu. Efficiently mining frequent itemsets from very large databases. Ph.D. thesis, Sept. 2004.

kDCI: on using direct count up to the third iteration

Claudio Lucchese
HPC Lab of ISTI-CNR
Pisa, Italy.
claudio.lucchese@isti.cnr.it

Salvatore Orlando
Università Ca' Foscari
Venezia, Italy.
orlando@dsi.unive.it

Raffaele Perego
HPC Lab of ISTI-CNR
Pisa, Italy.
raffaere.perego@isti.cnr.it

1. Problem Statement and Solution

In Apriori-like algorithms, one of the most consuming operation during the frequent itemset mining process is the candidate search. At each iteration k the whole dataset $\mathcal{D}$ has to be scanned, and for each transaction t in the database, every of its subsets of length k is generated and searched within the candidates. If a candidate is matched, it means that the transaction subsumes the candidate, and therefore its support can be incremented by one.

This search is very time demanding even if appropriate data structures are used to gain a logarithmic cost. In [3, 2] we introduced a *direct count* technique which allows constant time searches for candidates of length 2. Given the set of n frequent single items, candidates of length 2 are stored using an upper triangular matrix $n \times n$ DC_2 with $\binom{n}{2}$ cells, such that $DC_2(i, j)$ stored the support of the 2-itemset $\{ij\}$. As shown in [1] the direct count procedure can be extended to the third iteration using an $n \times n \times n$ matrix DC_3 with $\binom{n}{3}$ cells, where $DC_3(i, j, l)$ is the support of the 3-itemset $\{ijl\}$.

We thus introduced such technique in the last version of kDCI, which is level-wise hybrid algorithm. kDCI stores the dataset with an horizontal format to disk during the first iterations. After some iteration the dataset may become small enough (thanks to anti-monotone frequency pruning) to be stored in the main memory in a vertical format, and after that the algorithm goes on performing tid-lists intersections to retrieve itemsets supports, and searches among candidates are not needed anymore. Usually the dataset happens to be small enough at most at the fourth iteration.

2. Experiments and Conclusion

The experiments show that the improvement given by this optimization is sensible in some cases. The time needed for the third iteration is halved. Note that the time spent during the firsts iteration is significant for the global effectiveness of the algorithm on most datasets. In fact, in the test performed, the total time was reduced from 19 sec. to 14 sec., which means an overall speed up of about 20%.

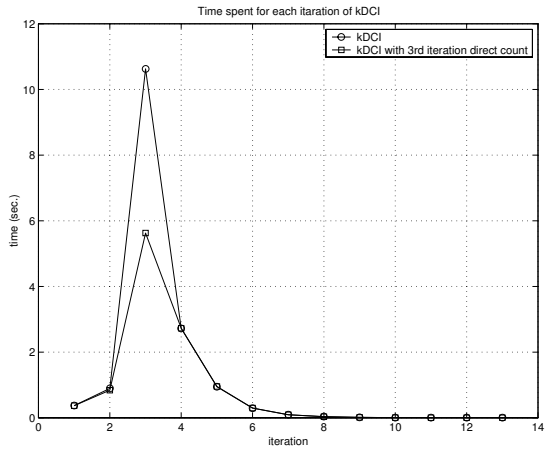


Figure 1. The dataset used was T10I4D100K with a minimum absolute support of 10.

We acknowledge the authors C.Targa, A.Prado and A.Plastino of [1], who showed the effectiveness of such optimization.

References

[1] C.Targa, A.Prado, and A.Plastino. Improving direct counting for frequent set mining. Technical report, Instituto de Computação, UFF RT-02/09 2003.

[2] Claudio Lucchese, Salvatore Orlando, Paolo Palmerini, Raffaele Perego, and Fabrizio Silvestri. kdci: a multi-strategy algorithm for mining frequent sets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, November 2003.

[3] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets. In *Proc. The 2002 IEEE International Conference on Data Mining (ICDM02)*, page 338345, 2002.

Recursion Pruning for the Apriori Algorithm

Christian Borgelt

Department of Knowledge Processing and Language Engineering
School of Computer Science, Otto-von-Guericke-University of Magdeburg
Universitätsplatz 2, 39106 Magdeburg, Germany
Email: borgelt@iws.cs.uni-magdeburg.de

Abstract

Implementations of the well-known Apriori algorithm for finding frequent item sets and associations rules usually rely on a doubly recursive scheme to count the subsets of a given transaction. This process can be accelerated if the recursion is restricted to those parts of the tree structure that hold the item set counters whose values are to be determined in the current pass (i.e., contain a path to the currently deepest level). In the implementation described here this is achieved by marking the active parts every time a new level is added.

1. Introduction

The implementation of the Apriori algorithm described in [2] uses a prefix tree to store the counters for the different item sets. This tree is grown top-down level by level, pruning those branches that cannot contain a frequent item set. This tree also makes counting efficient, because it becomes a simple doubly recursive procedure: To process a transaction for a node of the tree, (1) go to the child corresponding to the first item in the transaction and process the rest of the transaction recursively for that child and (2) discard the first item of the transaction and process it recursively for the node itself (of course, the second recursion is more easily implemented as a simple loop through the transaction). In a node on the currently added level, however, we increment a counter instead of proceeding to a child node. In this way on the current level all counters for item sets that are part of a transaction are properly incremented.

2. Recursion Pruning

Since the goal of the recursive counting is to determine the values of the counters in the currently deepest level of the tree (the one added in the current pass through the data), one can restrict the recursion to those nodes of the tree that have a descendant on the currently deepest level. Visiting other nodes is not necessary, since no changes are made to these nodes or any of their descendants — only the nodes in the currently deepest level of the tree are changed.

To implement this idea, which I got aware of at FIMI 2003, either from the presentation by F. Bodon [1] or from a subsequent discussion with B. Goethals, I added markers to each node of the prefix tree, which indicate whether the node has a descendant on the currently deepest level. Fortunately only one bit is necessary for such a marker, which could be incorporated into an already existing field, so that the memory usage is unaffected.

These markers are updated each time a new level is added to the tree, using a recursive traversal, which marks all nodes that have only marked children. New nodes are, of course, unmarked, and nodes on the previously deepest level that did not receive any children are marked to seed the recursion. Note that the recursion can exploit the markers set in previous passes, because a node that did not have a descendant on the deepest level in the previous pass cannot acquire a descendant on the currently deepest level.

Of course, the other pruning methods for the counting process described in [2] are applied as well.

3. Experimental Results

I ran experiments on the same five data sets I already used in [2], relying on the same machine and operating system, though updated to a newer version (an AMD Athlon XP 2000+ machine with 756 MB main memory running S.u.S.E. Linux 9.1 and gcc version 3.3.3). Strangely enough, however, the new versions of the operating system or the compiler lead to longer(!) execution times for an identical program, an effect that seems to be a nasty recurring feature of the S.u.S.E. Linux distribution. Therefore the experiments were repeated with the old program version to get comparable results.

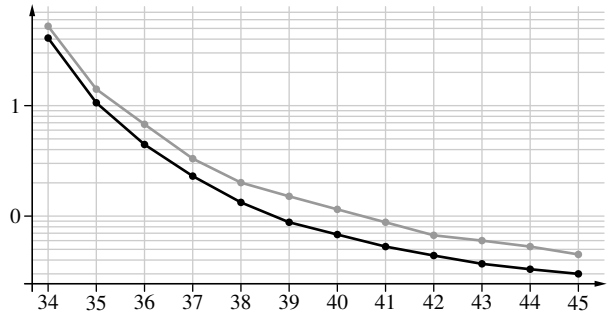


Figure 1. Results on BMS-Webview-1

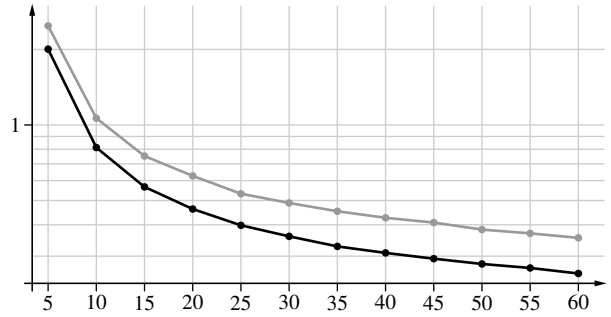


Figure 2. Results on T10I4D100K

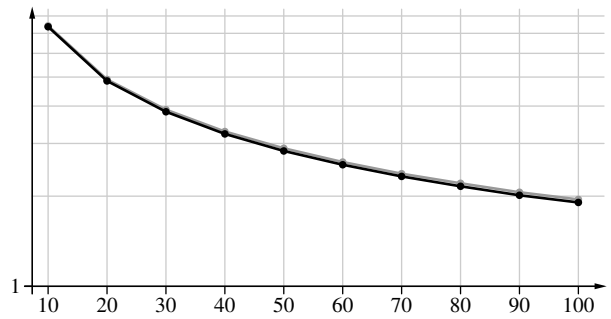


Figure 3. Results on census

Figure 4. Results on chess

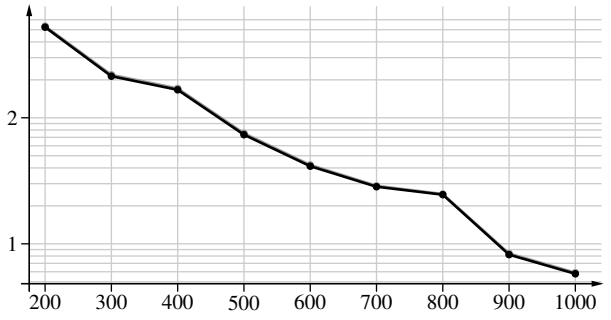


Figure 5. Results on mushroom

The results for these data sets are shown in Figures 1 to 5. The horizontal axis shows the minimal support of an item set (number of transactions), the vertical axis the decimal logarithm of the execution time in seconds. Each diagram shows as grey and black lines the time without and with recursion pruning, respectively.

As can be seen from these figures, recursion pruning can lead to significant improvements on some data set. (Note that the vertical scale is logarithmic, so that the 20-40% reduction, which results for webview1, for example, appears to be smaller than it actually is.) For census, chess, and mushroom, however, the gains are negligible.

4. Programs

The implementation of the Apriori algorithm described in this paper (Windows™ and Linux™ executables as well as the source code) can be downloaded free of charge at

http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html

The special program version submitted to the workshop uses the default parameter setting of this program.

References

[1] F. Bodon. A Fast Apriori Implementation. *Proc. 1st IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL)*. CEUR Workshop Proceedings 90, Aachen, Germany 2003. http://www.ceur-ws.org/Vol-90/

[2] C. Borgelt. Efficient Implementations of Apriori and Eclat. *Proc. 1st IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL)*. CEUR Workshop Proceedings 90, Aachen, Germany 2003. http://www.ceur-ws.org/Vol-90/

WebDocs: a real-life huge transactional dataset.

Claudio Lucchese[2], Salvatore Orlando[1], Raffaele Perego[2], Fabrizio Silvestri[2]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Venezia, Italy,orlando@dsi.unive.it

[2] ISTI-CNR, Consiglio Nazionale delle Ricerche, Pisa, Italy, {r.perego,c.lucchese,f.silvestri}@isti.cnr.it

Characteristics of the dataset

This short note describes the main characteristics of WebDocs, a huge real-life transactional dataset we made publicly available to the Data Mining community through the FIMI repository. We built WebDocs from a spidered collection of web html documents. The whole collection contains about 1.7 millions documents, mainly written in English, and its size is about 5GB.

The transactional dataset was built from the web collection in the following way. All the web documents were preliminarly filtered by removing html tags and the most common words (stopwords), and by applying a stemming algorithm. Then we generated from each document a distinct transaction containing the set of all the distinct terms (items) appearing within the document itself.

The resulting dataset has a size of about $1,48GB$. It contains exactly $1.692.082$ transactions with $5.267.656$ distinct items. The maximal length of a transaction is 71.472. Figure 1 plots the number of frequent itemsets as a function of the support threshold, while Figure 2 shows a bitmap representing the horizontal dataset, where items were sorted by their frequency. Note that to reduce the size of the bitmap, it was obtained by evaluating the number of occurrences of a group of items having subsequent Id's in a subset of subsequent transactions and assigning a level of gray proportional to such count.

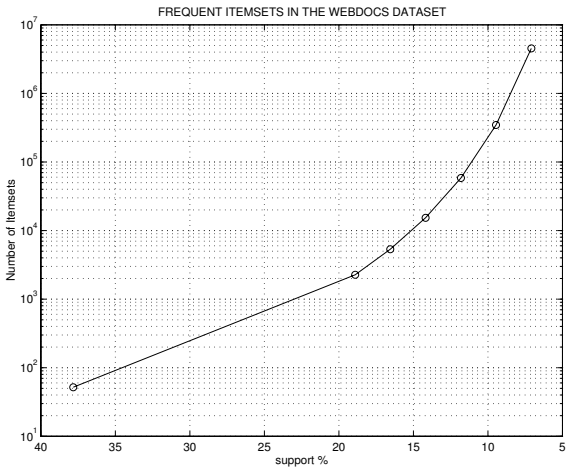


Figure 1. Number of frequent itemsets discovered in the WebDocs dataset as a function of the support threshold.

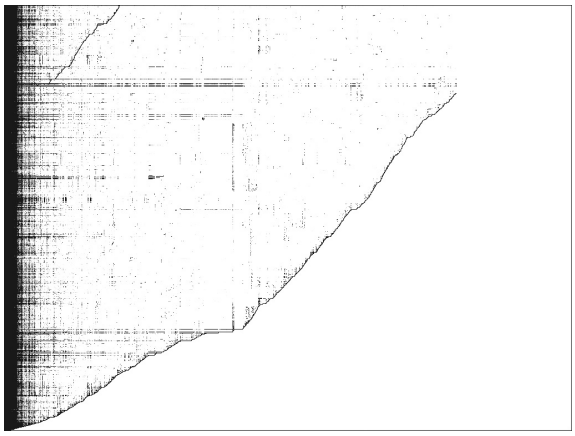


Figure 2. Bitmap representing the dataset.