SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **39**:551–571
Published online 21 October 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.902

Improved methods for extracting frequent itemsets from interim-support trees

F. Coenen[1], P. Leng[1], A. Pagourtzis[2], W. Rytter[3,4] and D. Souliou[2,*,†]

[1] *Department of Computer Science, University of Liverpool, Ashton Building, Ashton Street, Liverpool L69 3BX, U.K.*
[2] *School of Electrical and Computer Engineering, National Technical University of Athens, 15780 Zografou, Athens, Greece*
[3] *Institute of Informatics, University of Warsaw, Poland*
[4] *Department of Mathematics and Informatics, Copernicus University, Torun, Poland*

SUMMARY

Mining association rules in relational databases is a significant computational task with lots of applications. A fundamental ingredient of this task is the discovery of sets of attributes (*itemsets*) whose frequency in the data exceeds some threshold value. In this paper we describe two algorithms for completing the calculation of frequent sets using a tree structure for storing partial supports, called interim-support (IS) tree. The first of our algorithms (*T*-Tree-First (TTF)) uses a novel tree pruning technique, based on the notion of (*fixed-prefix*) *potential inclusion*, which is specially designed for trees that are implemented using only two pointers per node. This allows to implement the IS tree in a space-efficient manner. The second algorithm (*P*-Tree-First (PTF)) explores the idea of storing the frequent itemsets in a second tree structure, called the *total support tree* (*T*-tree); the main innovation lies in the use of multiple pointers per node, which provides rapid access to the nodes of the *T*-tree and makes it possible to design a new, usually faster, method for updating them. Experimental comparison shows that these techniques result in considerable speedup for both algorithms compared with earlier approaches that also use IS trees (*Principles of Data Mining and Knowledge Discovery*, *Proceedings of the 5th European Conference*, *PKDD*, *2001*, Freiburg, September 2001 (*Lecture Notes in Artificial Intelligence*, vol. 2168). Springer: Berlin, Heidelberg, 54–66; *Journal of Knowledge-Based Syst.* 2000; 13:141–149). Further comparison between the two new algorithms, shows that the PTF is generally faster on instances with a large number of frequent itemsets, provided that they are relatively short, whereas TTF is more appropriate whenever there exist few or quite long

[*] Correspondence to: D. Souliou, School of Electrical and Computer Engineering, National Technical University of Athens, 15780 Zografou, Athens, Greece.
[†] E-mail: dsouliou@cslab.ece.ntua.gr

Contract/grant sponsor: Polish Ministry of Science and Higher Education; contract/grant number: N206 004 32/0806
Contract/grant sponsor: European Social Fund (75%) and National Resources (25%)—Operational Programme 'Education and Initial Vocational Training' (ΕΠΕΑΕΚ II); contract/grant number: 'Pythagoras' NTUA, 68/810

frequent itemsets; in addition, TTF behaves well on instances in which the densities of the items of the database have a high variance. Copyright © 2008 John Wiley & Sons, Ltd.

Received 5 April 2006; Revised 4 June 2008; Accepted 28 July 2008

KEY WORDS: association rules; frequent itemsets; data mining; set-enumeration trees

1. INTRODUCTION

An important data mining task initiated in [1] is the discovery of association rules over huge listings of sales data, also known as *basket data*. This task initially involves the extraction of *frequent* sets of items from a database of transactions, i.e. from a collection of sets of such items. An example of a database with transactions that are subsets of $\{a,b,c,d,e,f,g,h\}$ is given in Table I. The number of times that an itemset appears in transactions of the database is called its *support*. The minimum support an itemset must have in order to be considered as frequent is called the *support threshold*, a non-negative integer denoted by t. The support of an association rule $A \Longrightarrow B$, where A and B are sets of items, is the support of the set $A \cup B$. The *confidence* of rule $A \Longrightarrow B$ is equal to $support(A \cup B)/support(A)$ and represents the fraction of transactions that contain B among transactions that contain A. A *valid* rule is one with support at least the support threshold t and with confidence at least a confidence threshold c.

1.1. Examples of association rules

Let $\mathscr{D}$ be the database shown in Table I. Let $t=4$ be the support threshold and $c=0.5$ be the confidence threshold. Rules $\{b\} \Longrightarrow \{c\}$ and $\{a\} \Longrightarrow \{d\}$ have both adequate support, because $support(\{b,c\})=7$ and $support(\{a,d\})=9$. However, the former rule is not valid since $confidence(\{b,c\})=support(\{b,c\})/support(\{b\})=7/17<0.5$; on the other hand, the latter rule is valid as $confidence(\{a,d\})=support(\{a,d\})/support(\{a\})=9/15>0.5$.

Table I. A database containing 32 transactions. Each transaction is described by a subset of $\{a,b,c,d,e,f,g,h\}$.

a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*
1	1	0	1	1	0	1	1	1	1	1	1	1	1	0	0	0	1	1	0	1	1	1	1
1	1	0	0	1	1	1	0	0	0	0	1	0	1	0	0	0	1	0	1	1	1	0	1
0	1	0	1	0	0	0	1	0	1	1	0	0	1	0	0	1	0	0	0	1	0	1	0
0	1	1	0	0	0	1	0	0	1	0	0	1	1	1	0	1	1	0	1	1	0	0	0
1	1	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	0	1	1	0	1	0	0
1	1	0	1	1	0	1	0	1	0	1	1	1	0	1	1	0	1	1	1	0	1	1	0
0	1	0	1	1	1	1	1	0	1	0	0	0	0	1	1	0	1	0	0	1	0	0	0
1	0	0	1	0	1	1	1	0	1	1	1	0	1	0	1	1	0	0	0	1	0	1	0
1	0	0	1	1	0	0	1	0	0	1	1	0	0	1	0	1	0	1	0	1	0	0	0
1	0	0	1	0	1	1	0	1	0	0	1	1	0	0	0	0	0	0	0	1	0	0	1
0	0	0	1	0	0	1	0	1	0	0	0	1	0	1	1								

Association rule mining, in general, involves the extraction of all valid rules from a database. The major part of this task is the discovery of the frequent itemsets; once the support of all these sets has been counted, determining valid rules can be done as follows. For each frequent itemset X ($support(X) \geq t$), consider all itemsets $Y \subseteq X$ (all such subsets are necessarily frequent as well). If $support(X)/support(Y) \geq c$ it turns out that the following rule is valid:

$$Y \Longrightarrow X \setminus Y$$

Thus, we see that the above procedure finds all valid rules.

Of course, there is no polynomial-time (w.r.t. the input size) algorithm for generating all frequent itemsets, as their number can be exponential in the size of the database. For example, consider a database with n items and n transactions; if there exist m transactions of the form $111\ldots1$, then all $2^n - 1$ possible itemsets have support at least m and are consequently frequent if $m > t$. Therefore, this problem has motivated a continuing search for effective heuristics.

The best-known algorithm, from which most others are derived, is Apriori [2]. Apriori performs repeated passes of the database, successively counting the support for single items, pairs, triples, etc. At the end of each pass, itemsets that fail to reach the support threshold are eliminated, and *candidate* itemsets for the next pass are constructed as supersets of the remaining frequent sets. As no frequent set can have an infrequent subset, this heuristic ensures that all sets that may be frequent are considered. The algorithm terminates when no further candidates can be constructed.

Apriori remains potentially very costly because of its multiple database passes and, especially, because of the possible large number of candidates in some passes. Attempts to reduce the scale of the problem include methods that begin by partitioning [3] or sampling [4] the data, and those that attempt to identify *maximal* frequent sets [5,6] or *closed* frequent sets [7] from which all others can be derived. A number of researchers have made use of the *set-enumeration tree* structures to organize candidates for more efficient counting. The FP-growth algorithm of Han *et al.* [8,9] counts frequent sets using the structure *FP-tree*, in which tree nodes represent individual items and branches represent itemsets. FP-growth reduces the cost of support-counting because branches of the tree that are subsets of more than one itemset need only be counted once. In contemporaneous work, commencing with [10], we have also employed set-enumeration tree structures to exploit this property. Our approach begins by constructing a tree, the P-tree, [11,12], which contains an incomplete summation of the support of sets found in the data. The P-tree, described in more detail below, shares the same performance advantage of the FP-tree but is a more compact structure. Results presented in [13] demonstrate that algorithms employing the P-tree can achieve comparable or superior speed to FP-growth, with lower memory requirements.

Unlike the FP-tree, which was developed specifically to facilitate the FP-growth algorithm, the P-tree is a generic structure, which can be the basis of many possible algorithms for completing the summation of frequent sets. In this paper we describe and compare two algorithms for this purpose, namely:

1. The T-Tree-First (TTF) algorithm.
2. The P-Tree-First (PTF) algorithm.

Both algorithms make use of the incomplete summation contained in the P-tree to construct a second set-enumeration tree, the T-tree, which finally contains frequent itemsets together with their total support. The algorithms differ in the way they compute the total support: algorithm T-Tree-First iterates over the nodes of T-tree, and for each of them it traverses the P-tree; algorithm PTF

DOI: 10.1002/spe

starts by traversing the P-tree and for each node that it visits, it updates all relevant nodes at the current level of the T-tree.

Earlier algorithms that use similar tree structures are Apriori-TFP (ATFP) [13] and an anonymous algorithm presented in [11]; here we will refer to the latter as 'Interim-Support' (IS).

The contribution of this work lies in the introduction of techniques that can considerably accelerate the process of computing frequent itemsets. In particular, the main innovation in the first of our algorithms (TTF) is a tree pruning technique, based on the notion of *fixed-prefix potential inclusion*, which is specially designed for trees that are implemented using only two pointers per node. This allows to implement the IS tree in a space-efficient manner. The second algorithm (PTF) introduces the use of multiple pointers per node in the T-tree; this accelerates the access of the nodes of the T-tree and makes it possible to find and update appropriate T-tree nodes following a new, usually faster, strategy.

We perform experimental comparison of the two algorithms against the earlier algorithms IS and ATFP and show that in most cases the speedup is considerable. We also compare the two new algorithms with each other and discuss the merits of each. Our results show that PTF is faster than TTF if there are a lot of frequent itemsets in the database (small support threshold), provided that they are *short*, i.e. they contain few items. On the other hand, TTF gains ground as the support threshold increases and behaves even better for instances of variable item density, which have been pre-sorted according to these densities; it also behaves much better than PTF in instances with long frequent itemsets.

2. NOTATION AND PRELIMINARIES

A database $\mathscr{D}$ is represented by an $m\times n$ binary matrix. The columns of D correspond to items (attributes), and the rows correspond to the transactions (records). The columns are indexed by consecutive letters $a, b, \ldots$ of the alphabet (see Table I for an example). The set of columns (items) is denoted by $\mathscr{C}$. An *itemset* I is a set of items $I\subseteq\mathscr{C}$. For an itemset I we define:

- $E(I)$ (E-value of I) is the number of transactions that are exactly equal to I. This value is also called *exact support of* I.
- $P(I)$ (P-value of I) is the number of transactions that have I as a prefix. Also called IS *of* I.
- $T(I)$ (T-value of I) is the number of transactions that contain I. Also called *total support* or, simply, *support of* I.

Both the terms *P-value* and *P-tree* have been used in other contexts with other meanings. Here, the derivation is the notion of a *partially* counted support value. In this paper we consider the problem of finding all itemsets I with total support $T(I)\geq t$, for a given database $\mathscr{D}$ and threshold t, starting with a P-tree containing P-values for all sets present as transactions in $\mathscr{D}$.

For an item x we define the *density of* x *in* $\mathscr{D}$ to be the fraction of transactions of $\mathscr{D}$ that contain x, that is $T(\{x\})/m$. We also define the *density of a database* $\mathscr{D}$ to be the average density of the items of $\mathscr{D}$; note that the density of $\mathscr{D}$ is equal to the fraction of the total number of items appearing in the transactions of $\mathscr{D}$ over the size of $\mathscr{D}$ ($=nm$).

We will make use of the following order relations:

- *Inclusion order*: $I\subseteq J$, the usual set inclusion relation.
- *Lexicographic order*: $I\leq J$, I is lexicographically smaller or equal to J if seen as strings.

- *Prefix order*: $I \sqsubseteq J$, I is a prefix of J if seen as strings. Note that $I \sqsubseteq J \Leftrightarrow I \subseteq J \& I \leq J$.

We will also use the corresponding operators without equality: $I \subset J$, $I \sqsubset J$ and $I < J$.

Notice that for any itemset I:

$$T(I) = \sum_{J:I \subseteq J} E(J)$$

and therefore:

$$T(I) = \sum_{J:I \subseteq J \& I \leq J} E(J) + \sum_{J:I \subseteq J \& J < I} E(J) = P(I) + \sum_{J:I \subseteq J \& J < I} E(J) \tag{1}$$

This property will play an important role in our algorithms.

3. THE INTERIM-SUPPORT TREE

Both the new algorithms TTF and PTF have a common first part, which is a pre-processing of the database that results in the storage of the whole information into a structure called the P-tree or IS *tree*. The P-tree is a set-enumeration tree the nodes of which are distinct itemsets of the database as well as some common prefixes of these itemsets. For each node, the IS (P-value) of the corresponding itemset is also stored.

The notion of IS trees was introduced in [11], where details of the construction of the P-tree were given, and more fully in [12]. The algorithm is summarized below.

```
                         Algorithm $P$-Tree-Build
Input: Database $\mathcal{D}$.
Output: $P$-Tree of itemsets in $\mathcal{D}$.

(* Start with $P$-tree of a single node representing the empty set *)

for each transaction $i$ in $\mathcal{D}$ do
    $c$ := $P$-tree_rootnode
    $inserted$ := $false$
    while not $inserted$ do
        if $c = i$ then increment $P(c)$; $inserted$ := $true$
        else if $c \subset i$ then increment $P(c)$; $c$ :=eldest_child_of.$c$
        else if $c < i$ then $c$ :=next_sibling_of.$c$
        else create new node for $i$; $inserted$ := $true$
return $P$-tree;
```

Note that in this algorithm, for the sake of clarity, we use the notation i and c to denote an itemset that also is or will become the label of a node in the tree. The tree is constructed in a single

DOI: 10.1002/spe

pass of $\mathscr{D}$. As each transaction is examined, the tree is traversed in a top-down (preorder) manner until either a node with an identical itemset is found or the traversal passes the position in the tree at which the new itemset should be located. During this traversal, the support of all ancestors (preceding subsets) of the itemset is incremented.

If the itemset is not found in the tree, a new node is added to the tree to represent it. At this point the traversal has reached a node c, which is either null (i.e. a non-existent child or a sibling) or lexicographically follows the new itemset i. A node labelled i is inserted at the position in the tree structure occupied by c. The following three different cases apply for dealing with the previous node c and recording the IS of i:

- *c is null*: The new node i is given support $P(i)=1$.
- $i \subset c$: c becomes the child of i. $P(i)=P(c)+1$.
- *Otherwise*: c becomes the next sibling of i. $P(i)=1$.

Finally, if i has been added as a sibling of c, and i and c share a leading substring d that is not already in the tree, a node d is inserted at the position now occupied by i, with i and c becoming its children, and $P(d)=P(i)+P(c)$.

In any case, during the insertion of an itemset at most two new nodes will be created in the P-tree. On the other hand, if the database contains several identical itemsets, the P-tree can be much smaller than the original database.

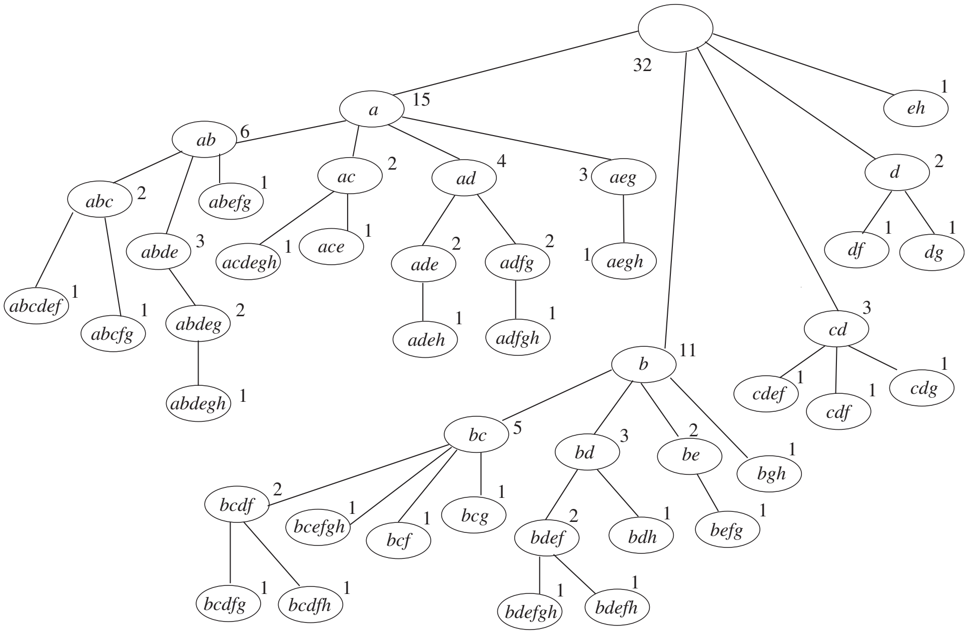

Figure 1. P-tree with interim supports for the database of Table I.

The P-tree that corresponds to the database of Table I is shown in Figure 1. Note that Figure 1 shows the logical structure of the P-tree. However, for the sake of memory efficiency the P-tree is implemented using two pointers per node: *down* and *right*. For a node v, its down pointer links v to one of its children—the lexicographically smaller. This child's right pointer points to another child of v, and so on. For example, in the implementation of a P-tree containing itemsets 'a', 'ab', 'ac', and 'abc' node 'a' points down to 'ab', which in turn points down to 'abc' and right to 'ac'.

The significance of the P-tree is that it performs a large part of the counting of support totals very efficiently in a single database pass. The size of the P-tree is linearly related to the original database, and will be smaller in cases where the data include many duplicated itemsets. Most importantly, it involves no loss of relevant information, thus the P-tree can be used as a surrogate for the original database in any chosen algorithm.

The FP-tree of Han *et al.* [8,9] was developed independently and contemporaneously with our P-tree [10,11] and shares similar performance advantages. There are three significant differences between the two structures. First, the construction of the FP-tree requires two database passes, the first of which eliminates attributes that fail to meet the required support threshold, so that it no longer contains a complete representation of the information in the database. Second, the nodes of the FP-tree correspond to individual items, whereas in the P-tree a sequence of items, which is partially closed (i.e. which has no leading subsequence with greater support in the tree) will be stored as a single tree node. Thus, for example, two transactions $\{a, b, c, d, e\}$ and $\{a, b, c, x, y\}$, which share a common prefix $\{a, b, c\}$, would require in all seven nodes in the FP-tree. In the P-tree, conversely, only three nodes would necessarily be created: a parent for $\{a, b, c\}$, and child nodes for $\{d, e\}$ and $\{x, y\}$. Finally, in order to implement the FP-growth algorithm, the FP-tree must store pointers at each node to link all nodes representing the same item, and also to link a node to its parent and child nodes. The nodes of the P-tree, conversely, require only pointers to the eldest child and the next sibling. Both the latter differences lead to a more compact tree structure and hence faster traversal.

More importantly, the simpler and less pointer-rich organization of the P-tree makes it a more flexible structure than the FP-tree, which was developed specifically to implement the FP-growth algorithm. This flexibility, for example, allows us to implement an algorithm, ATFP, which applies an Apriori-like procedure to the nodes in the P-tree. In the results presented in [13] both the memory requirements and the construction time for the P-tree were less than those for a corresponding FP-tree, and the execution time for ATFP was similar to or less than FP-growth and much less than Apriori. In this paper we will use ATFP as a benchmark against which to measure the performance of the new algorithms proposed.

A further advantage of the relatively simple P-tree structure is that it facilitates scaling to deal with data that cannot be contained in the main memory. In this case, the original database is segmented into partitions for each of which a separate P-tree is constructed. This process again requires only a single pass of the database to produce a set of Partition-P-trees (PP-trees). Subsequently, algorithms that require to traverse the P-tree can operate by separately traversing each of the PP-trees, accumulating support counts from each to produce the overall totals. Partitioning the FP-tree is necessarily more complex, though methods for doing this are described in [9]. In [14] we described an implementation of ATFP using a tree partitioning strategy. The results obtained showed that segmenting the data enabled effective scaling of the method, and demonstrated improved performance over a partitioned version of FP-growth. Similar partitioning strategies can be applied to the algorithms described in this paper, and thus, though the experiments described relate to databases that can be contained in the main memory, the methods can be applied on a larger scale.

A number of other researchers have made use of the FP-tree and similar structures. The CFP-tree described in [15] stores frequent closed itemsets in a form that facilitates subsequent query processing. The main contribution of this work is a structure that can be re-used efficiently, rather than the efficiency of the construction algorithm. Reusability is also a feature of the P-tree, which, as we have mentioned, retains all relevant information from the original data. In [16] a structure is described, also (coincidentally) called a P-tree, which is quite similar to our P-tree, but (like the FP-tree) stores only one item at each node. The approach described constructs FP-trees from the P-tree rather than from the original data, producing a single overall FP-tree, for which further partitioning might become necessary if the data are too large to contain in the main memory.

4. THE *T*-TREE-FIRST (TTF) ALGORITHM

The TTF algorithm first iterates over the nodes of T-tree and for each of them it traverses the P-tree. In this section we give a detailed description of TTF.

The algorithm first scans the database and creates the P-tree, as explained in the previous section.

It then starts building the T-tree (recall that the T-tree will finally contain all frequent itemsets together with their total supports). Each level of the T-tree is implemented as a linear list, where itemsets appear in lexicographic order; nodes of such a list neither point to nor are pointed from nodes that are in the list of another level. In the beginning, the algorithm builds level 1 of the T-tree, which contains all frequent singletons; to this end it counts their support traversing the P-tree. It then builds the remaining T-tree level by level using procedure **Iteration**(k).

The algorithm is presented below. A fundamental ingredient of TTF is the function **CountSupport**, which is described separately.

```
                    Algorithm T-Tree-First (TTF)
Input: Database $\mathscr{D}$, threshold $t$.
Output: The family $\mathscr{F}$ of frequent itemsets.

Build $P$-tree from database $\mathscr{D}$;
(* Build the 1-st level of $T$-tree *)
for $i=1$ to $n$ do
    if CountSupport($P$-tree, $\{i\}$) $\geq t$ then add $\{i\}$ to $\mathscr{F}_1$;
(* Build the remaining levels of $T$-tree *)
for $k=2$ to $n$ do
    Iteration($k$);
    if $\mathscr{F}_k=\emptyset$ then exit
    else $\mathscr{F}=\mathscr{F}\cup\mathscr{F}_k$;
return $\mathscr{F}$;
```

Some details of the procedure **Iteration**(k) need to be clarified. Its goal is to build $\mathscr{F}_k$, that is, the kth level of the T-tree. The procedure uses the heuristic first described in [2]. Itemsets in $\mathscr{F}_k$ must have all their $(k-1)$-size subsets in $\mathscr{F}_{k-1}$. Therefore, one can start from the existing itemsets in $\mathscr{F}_{k-1}$ and try to augment them with one more item in order to create all potentially frequent itemsets. To avoid duplications the algorithm may proceed by considering for each frequent itemset X_{k-1} in $\mathscr{F}_{k-1}$ all X_{k-1}'s supersets $X_k = \{x\} \cup X_{k-1}$ for items x that are greater than any item of X_{k-1}.

As already observed in [2], it makes sense to consider such supersets only if X_{k-1} and the node following it, denoted by X'_{k-1}, differ at the last item. The candidate superset X_k is then the union of X_{k-1} and X'_{k-1}. Then it is checked whether all the ($k-2$ many) remaining $(k-1)$-subsets of X_k are frequent; this task is carried out by a special function called **ExistSubsets**, which is not described in detail here. If some of the examined subsets of X_k are not present in $\mathscr{F}_{k-1}$, X_k is not added to $\mathscr{F}_k$.

```
Procedure Iteration(k) (* Building the kth level of T-tree *)
for each itemset $X_{k-1} \in \mathscr{F}_{k-1}$ do
    $X'_{k-1} := next(X_{k-1})$;
    while $X'_{k-1} \neq$ NULL do
        if $X_{k-1}$ and $X'_{k-1}$ differ only at the last item then
            $X_k := X_{k-1} \cup X'_{k-1}$;
            if ExistSubsets($X_k, \mathscr{F}_{k-1}$) then
                $T(X_k) :=$ CountSupport($P$-tree, $X_k$);
                if $T(X_k) \geq t$ then add $X_k$ to $\mathscr{F}_k$;
            $X'_{k-1} := next(X'_{k-1})$;
        else exit while;
```

In order to complete the description of TTF it remains to describe its most critical part, that is, function **CountSupport**, which counts the total support of an itemset X in the P-tree in a recursive manner. An essential ingredient of **CountSupport** is the notion of *fixed-prefix potential inclusion*:

Fixed-Prefix Potential Inclusion. $I \overset{\text{pot}}{\subseteq}_K J$: $\exists J'$, $commonprefix(J, J') = K \& I \subseteq J'$.

Examples: ‘bdf’ $\overset{\text{pot}}{\subseteq}_{\text{‘ab’}}$ ‘abc’, ‘bdf’ $\overset{\text{pot}}{\nsubseteq}_{\text{‘ab’}}$ ‘abd’.

In words, $I \overset{\text{pot}}{\subseteq}_K J$ means that there is an itemset greater than J, sharing with J a common prefix K, that contains I.

A second interesting inclusion relation can be defined in terms of $\overset{\text{pot}}{\subseteq}_K$:

Potential Inclusion. $I \overset{\text{pot}}{\subseteq} J \overset{\text{def}}{=} I \overset{\text{pot}}{\subseteq}_J J$ i.e. $\exists J', J \sqsubseteq J' \& I \subseteq J'$.

Examples: ‘bdf’ $\overset{\text{pot}}{\subseteq}$ ‘abde’, ‘bdf’ $\overset{\text{pot}}{\nsubseteq}$ ‘abdg’.

In words, $I \overset{\text{pot}}{\subseteq} J$ means that there is an extension of J that contains I.

The use of the above inclusion relations can significantly reduce the number of moves needed to count the support of an itemset in trees with two pointers per node. Suppose that we are looking for appearances (i.e. supersets) of an itemset I in the P-tree and we are currently visiting a node that contains itemset J:

- Nodes that are below the current node contain itemsets J', which have J as prefix. Therefore, if $I \overset{\text{pot}}{\nsubseteq} J$ there is no point visiting the subtree rooted at the current node.
- Nodes that are to the right of the current node (siblings) contain itemsets that have $par(J)$ (parent of J) as prefix—and so does J—and are greater than J. If $I \overset{\text{pot}}{\nsubseteq}_{par(J)} J$ there is no point visiting the subtrees rooted at these nodes.

These two tests result in much better tree pruning compared with the one applied by the IS algorithm [11]. As an example, suppose that we are trying to find the support of itemset $X =$ ‘bd’ in a P-tree in which there is a node ‘ab’ with children ‘abde’ and ‘abefg’. Then, once the tree traversal reaches node ‘abde’ it adds its support to $T(X)$ and does not move to the right, that is, it avoids visiting ‘abefg’. On the other hand, the IS algorithm would also examine ‘abefg’ (and other siblings if such existed) because it only terminates its search whenever it finds itemsets lexicographically equal or greater than X.

```
Function CountSupport($pnode$, $X$): integer
(* Counts the total support of itemset $X$
in the subtree of $P$-tree rooted at $pnode$*)
$T := 0$;
if $pnode \neq NULL$ then
    $J := pnode \rightarrow itemset$;
    if $X \overset{\text{pot}}{\subseteq} J$ then (* makes sense to search children *)
        if $X \subseteq J$ then $T := T + P(J)$
            (* inclusion is a special case of potential inclusion *)
        else $T := T +$ CountSupport($pnode \rightarrow down$, $X$);
    if $X \overset{\text{pot}}{\subseteq}_{par(J)} J$ then (* makes sense to search right siblings *)
        $T := T +$ CountSupport($pnode \rightarrow right$, $X$);
return $T$;
```

Finally, we explain how to check potential inclusion and fixed-prefix potential inclusion. It can be shown that the following tests suffice. The proof is omitted.

- $X \overset{\text{pot}}{\subseteq} J$: If $X \subseteq J$ then $X \overset{\text{pot}}{\subseteq} J$ is true. Otherwise let x be the lexicographically smaller item of X that is not item of J (such x exists). If for all items j of J are lexicographically smaller than x then $X \overset{\text{pot}}{\subseteq} J$ is true, otherwise it is false.

- $X \overset{\text{pot}}{\subseteq}_K J$: assume $K \sqsubseteq J$ (otherwise the inclusion $X \overset{\text{pot}}{\subseteq}_K J$ is obviously false). Let x be the first item of $X \setminus K$ and j be the first item of $J \setminus K$. If $x > j$ the inclusion $X \overset{\text{pot}}{\subseteq}_K J$ holds, otherwise it is false.

5. THE *P*-TREE-FIRST (PTF) ALGORITHM

The PTF algorithm also begins by constructing the P-tree exactly as TTF, but then it follows an inverse approach in order to update the T-tree. In particular, during the processing of level-k of the T-tree, each node of the P-tree is visited once. Let I be the itemset of a visited node; the algorithm updates all nodes of level-k that are subsets of I, except for those that are also subsets of $par(I)$ (parent of I)—the latter have already been updated while visiting $par(I)$.

Level-k itemsets of the T-tree are constructed from the itemsets of level-$(k-1)$ by adding single items to each of them. This is done without checking the frequency of all subsets of a candidate. This is in contrast to TTF where special care was taken in order to create as few candidates as possible; here it is more important to save time by avoiding checking the subsets. Then, the P-tree is traversed as described above in order to compute support for all nodes of level-k. Nodes with support smaller than the threshold are removed before the generation of level-$(k+1)$. An illustration of this process for the database of Table I is shown in Figure 2.

```
                    Algorithm P-Tree-First (PTF)
Input: Database $\mathscr{D}$, threshold $t$.
Output: The family $\mathscr{F}$ of frequent itemsets.

Build $P$-tree from database $\mathscr{D}$;
add $\emptyset$ to $\mathscr{F}_0$; (* create a dummy level with one empty itemset *)
(* Build level-$k$ of the $T$-tree *)
for $k=1$ to $n$ do
    Iteration($k$);
    if $\mathscr{F}_k=\emptyset$ then exit for
    else $\mathscr{F}=\mathscr{F}\cup\mathscr{F}_k$;
return $\mathscr{F}$;
```

Our innovation here is the use of multiple pointers at each node of the T-tree in contrast to earlier approaches (e.g. ATFP [13]) where two pointers per node are used. In particular, each node of the T-tree contains $n-k$ pointers, where n is the number of items and k is the level of the node; there is one pointer for each item that is lexicographically greater than the greatest item of the node. For example, in the T-tree for the database of Table I, a node that contains itemset ‘bde’ must also contain three pointers, one for each of ‘f’, ‘g’, ‘h’. If ‘bdeg’ is found to be frequent, it will be stored in the node pointed by the ‘g’ pointer of node ‘bde’.

The use of multiple pointers provides rapid access to the nodes of the T-tree, allowing for a new strategy for T-tree update. In particular, while building level k, once a node I of the P-tree is visited, all its k-subsets (subsets of size k) are generated; once such a k-subset is generated, it is sought in the T-tree and, if present, its support is updated accordingly. Whenever such an itemset J has a prefix J', which is not frequent (hence neither J can be frequent) the algorithm discovers this quite early and the update process terminates. For example, if the algorithm visits a node of the P-tree with itemset 'acdfghk' and the current level of the T-tree is level-6 the algorithm should update all size-6 subsets of 'acdfghk'. Consider 'acdfgh'; the algorithm will try to find this node starting from 'a' in level-1, continuing to 'ac' in level-2, and then to 'acd', 'acdf' and 'acdfg'. If 'acd' is non-frequent, i.e. does not exist in level-3, the algorithm stops and considers the lexicographically next size-6 subset of 'acdfghk'. In fact, PTF saves even more comparisons by considering 'acfghk' as the next subset because there is no need to check any subset that contains 'acd'. Note that, in such a case, we use a 'non-frequent' itemset, called NF, which keeps the last prefix that was found to be missing from the T-tree. On the other hand, ATFP traverses a potentially large list of candidate itemsets in order to check whether any of them is a k-subset of I (note that this also happens in the original Apriori algorithm [2] where I is the current transaction scanned). This could be much slower than the above described procedure, especially if I has few k-subsets in that list. A detailed description of the update of level-k of the T-tree is as follows:

```
Procedure Iteration($k$) (* Building $k$th level of $T$-tree *)
for each itemset $X_{k-1} \in \mathcal{F}_{k-1}$ do
    for each item $x$ greater than all items of $X_{k-1}$ do
        add $X_k := X_{k-1} \cup \{x\}$ to $\mathcal{F}_k$;
        let the $x$th down pointer of $X_{k-1}$ point to $X_k$;
(* Update total supports of nodes in $\mathcal{F}_k$ *)
for each node $I$ of the $P$-tree do
    $NF := \{\}$;
    for each itemset $J \subseteq I$ with $|J| = k$ in lex. order do
        if $J \subseteq par(I)$ or $(NF \subseteq J$ and $NF \neq \{\})$ then
            proceed to the lex. next $J \subseteq I$ such that
                $J$ is not subset of $par(I)$ and does not contain $NF$
        else
            repeat
                descend the $T$-tree following prefixes of $J$
            until $J$ is found or some $J' \sqsubseteq J$ is missing;
            if $J$ is found then $T(J) := T(J) + P(I)$
            else $NF := J'$; (* $J'$ is missing and $NF$ is set so that
                no itemset $J$ containing $J'$ will be considered
                in any subsequent inner for-loop *)
remove from $\mathcal{F}_k$ all nodes with support $< t$ (threshold);
```

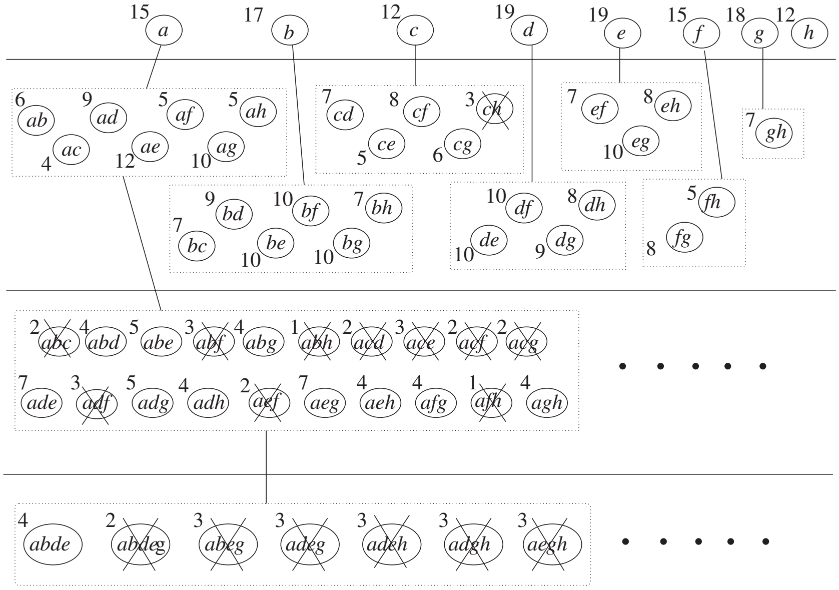

Figure 2. T-tree with total supports for the database of Table I.

6. COMPLEXITY OF THE ALGORITHMS

We will next provide some bounds on the complexity of algorithms TTF and PTF. Let us first remind the reader that there can be no algorithm for this problem that runs in time polynomial w.r.t. the size of the database (equivalently, w.r.t. m and n), as there are instances in which the number of frequent itemsets is $2^n - 1$. We shall therefore examine whether our algorithms' running time is polynomial w.r.t. m, n and the number R of frequent itemsets; note that the output size is at most Rn. We will prove that this holds for TTF, but probably not for PTF.

Theorem 1. TTF *has time complexity* $O(mn^2R)$ *and* PTF *has time complexity* $O(mn2^n)$.

Proof. The dominating term in the complexity, for both algorithms, is the frequency calculation process.

We first show that the complexity of this process is $O(mn^2R)$ for TTF. The proof is based on the fact that for each frequent itemset I_k of size k, at most $n-k$ supersets of size $k+1$ may be added to the list of potentially frequent itemsets of size $k+1$. This is because these itemsets are of the form $I_{k+1} = \{x\} \cup I_k$ where x can be any item lexicographically greater than all items of I_k. Thus, the total number of potentially frequent itemsets of size $k+1$ is bounded by $R_{k+1} \le R_k(n-k) \le R_k n$ and their overall number is thus bounded by Rn. TTF, for each of the (at most Rn) potentially frequent itemsets, examines a part of the P-tree, which contains at most $2m$ nodes in total. Again, comparison of the corresponding itemsets requires $O(n)$ time and the bound follows.

On the other hand, during Iteration(k), PTF does the following: for any of the (at most $2m$ many) nodes of the P-tree that contains, e.g. an itemset I_t of size t, it considers all possible k-size subsets of I_t, i.e. $\binom{t}{k} \leq \binom{n}{k}$ many itemsets. For each of these itemsets, it performs at most k moves in the T-tree in order to locate the itemset and update its frequency (if present). Summing over all levels, the frequency calculation costs at most $2m\sum_{k=1}^{n}\binom{n}{k}k = mn2^n$. □

Although the above result suggests that TTF is of lower complexity than PTF this is not always the case, as can be demonstrated by appropriate examples. In fact, the presented bounds are not directly comparable because if R is large (e.g. $\Theta(2^n)$) then the complexity of PTF is smaller, whereas if $R \ll 2^n$ then it is larger but probably too overestimated. Experimental comparison of the two algorithms is therefore meaningful.

7. EXPERIMENTAL COMPARISON

We implemented four algorithms in ANSI-C: TTF, IS, PTF and ATFP. We run several experiments using a Pentium 1.6 GHz PC. We have used four types of data sets: synthetic, synthetic of variable density, realistic data sets, and sparse data sets. The obtained results are presented below.

7.1. Synthetic data sets

We first experimented with data sets created by using the IBM Quest Market-Basket Synthetic Data Generator (described in [2]). We follow a standard notation according to which a data set is described by four parameters: T represents the average transaction length (roughly equal to the database density times the number of items), I represents the average length of maximal frequent itemsets, N represents the number of items, and D represents the number of transactions in the database. We generated data sets T10.I4.N50.D10K and T10.I4.N20.D100K and performed experiments with all four algorithms. The execution time of each algorithm for these two data sets and threshold varying from 1 to 5% is shown in Figure 3.

These results show that both algorithms TTF and PTF are faster than the earlier algorithms IS and ATFP, except for rather large thresholds. As regards TTF and IS (which also iterates over the TTF), the reason for this behavior is that IS performs fewer tests at each P-tree node that it visits; thus, whenever a contiguous part of the tree is traversed by both TTF and IS, IS is the one that does it faster. Now, whenever the frequent itemsets are few, they are also (most probably) of small size; a small itemset has higher chances of appearing in a contiguous part of the P-tree, which therefore cannot be pruned by TTF. As regards PTF and ATFP (which iterates over (PTF), we observe that ATFP can be faster than PTF if there are only few frequent itemsets because in such a case it can be faster to traverse the list of candidate itemsets than generating all subsets of a node.

Comparing now the two new algorithms, we observe that PTF is faster than TTF for small thresholds ($\leq 2\%$). This is due to the fact that whenever the number of frequent itemsets is large, TTF performs a lot of P-tree traversals, while PTF performs only one full P-tree traversal per T-tree level. As the size of the P-tree can be rather large (even comparable to the size of the database) its traversal is quite slow; hence, whenever TTF performs many traversals, even partial, the overall slowdown is considerable. On the other hand, PTF performs several T-tree traversals at each level but these are fast, thanks to the use of multiple pointers. The two algorithms have comparable running time for thresholds above 2%. This is because for relatively sparse T-tree

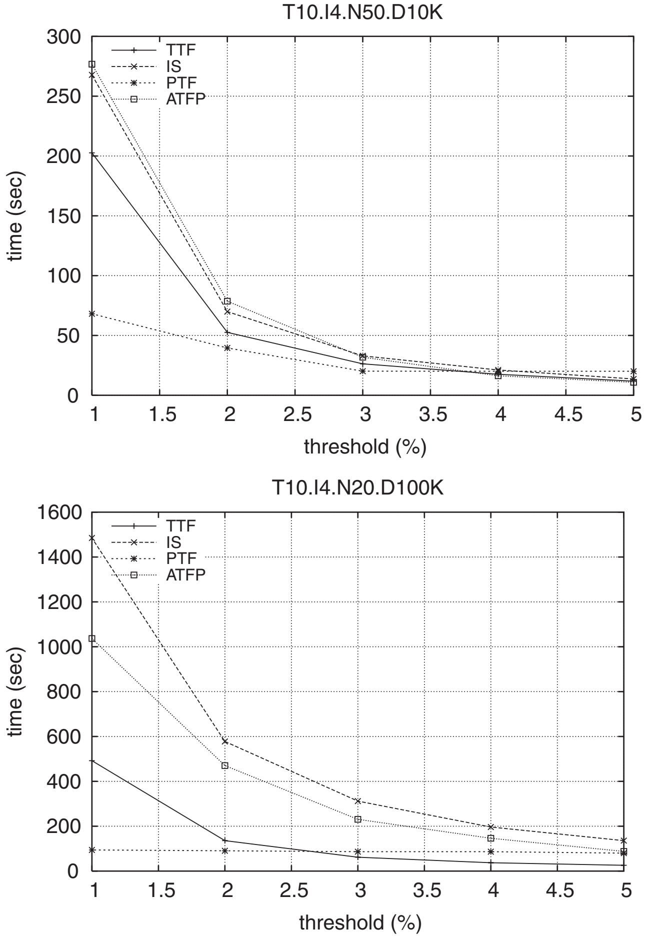

Figure 3. Results for data sets T10.I4.N50.D10K (top) and T10.I4.N20.D100K (bottom).

the P-tree traversals performed by TTF are few; in this case the economizing techniques of TTF balance, or even beat the advantages of PTF.

7.2. Variable-density data sets

To further compare TTF and PTF we implemented a probabilistic generator in order to create data sets of *variable item density* (each item has a different expected density). This generator fills the ith item of a row with probability $p_f-(i-1)p_s$, i.e. the probability decreases linearly as we move from the first to the last item of a row; p_f represents the probability of appearance of the first item and p_s is the decrement step. The expected density of the database is equal to $p_f-(n-1)/2p_s=(p_f-p_l)/2$, where p_l is the probability of appearance of the last item and n is the number of items in each row.

We have generated four variable-density data sets, one for each of the following four types (where letter ‘V’ stands for ‘variable density’): V.T4.N20.D10K, V.T6.N20.D10K, V.T4.N20.D100K, and V.T6.N20.D100K; the corresponding first item selection probabilities and decrement steps (in parentheses) are 0.4 (0.02), 0.6 (0.03), 0.4 (0.02) and 0.6 (0.03), respectively.

We performed experiments with support thresholds ranging from 0.5 to 5%. For each data set type/threshold combination we have measured the execution time of PTF and TTF, averaging over 10 experiments, one for each data set of the type.

Results for the data sets with 10K transactions appear in Figure 4. Figure 5 shows the results for the data sets with 100K transactions.

The comparison of the two algorithms is much more interesting when it comes to variable-density data sets. As before, PTF behaves better for small thresholds (roughly smaller than 2%) but TTF is

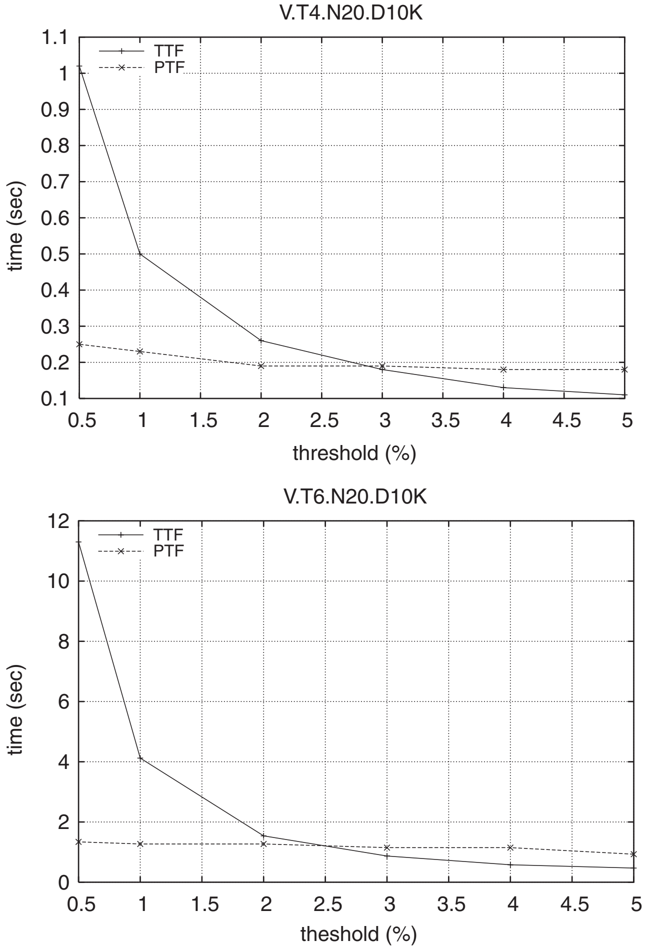

Figure 4. Results for data sets V.T4.N20.D10K (top) and V.T6.N20.D10K (bottom).

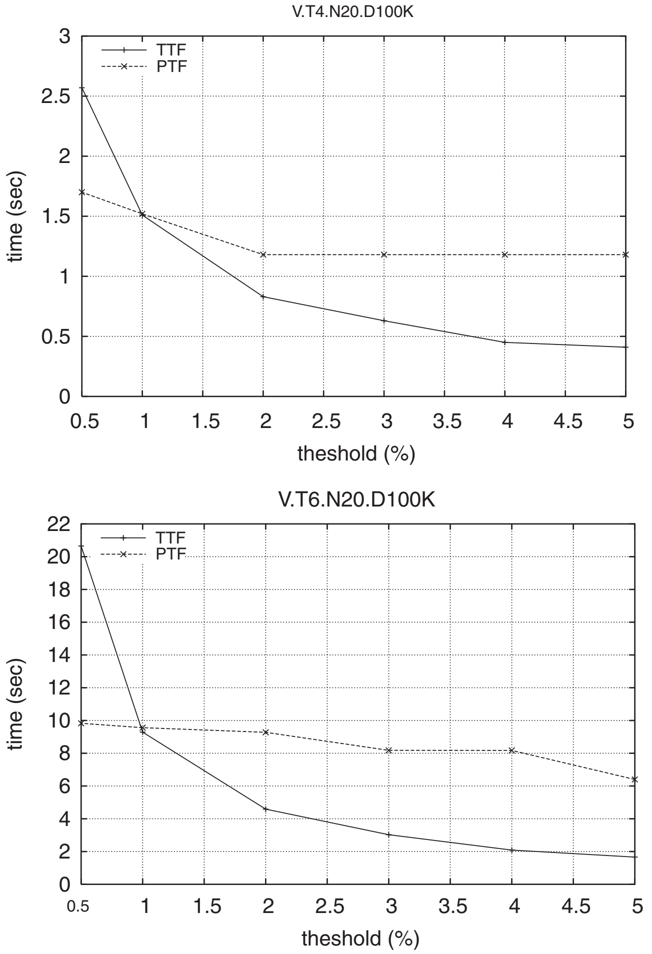

Figure 5. Results for data sets V.T4.N20.D100K (top) and V.T6.N20.D100K (bottom).

faster for larger thresholds. Besides, PTF exhibits almost constant running time in most experiments. Now, whenever the T-tree is small and sparse, it happens that the few full P-tree traversals performed by PTF can take longer than the (more but not too many) partial P-tree traversals of TTF. The main reason is that potentially frequent itemsets consist mainly of lexicographically smaller items, hence the partial P-tree traversals of TTF are limited to a small part of the P-tree and are therefore much faster. On the other hand, TTF performs a full P-tree traversal at each level of the T-tree that contains potentially frequent itemsets, regardless of the number of these itemsets, hence it needs almost the same time as before, since it considers a similar number of levels.

Comparing the performance of the two algorithms with respect to uniformity of item densities, one observes that while PTF exhibits roughly the same performance for both uniform and variable item densities, TTF is considerably faster on instances of variable item density; indeed, our results show that for variable-density data sets, TTF outperforms PTF for support thresholds above 3%, even above 2 or 1% in some cases. This is due to the fact that the performance of PTF is mainly determined by the rank of the higher level of frequent itemsets, whereas the performance of TTF depends heavily on the part of the P-tree that must be visited each time—which is much smaller for variable-density instances—because frequent itemsets consist mainly of lexicographically smaller items.

Let us note here that for our experiments we built the variable-density data sets in such a way that the lexicographically greater items are of smaller density. This property is essential for the performance of TTF, as it guarantees that most frequent itemsets consist mainly of lexicographically small items, which appear in a small part of the P-tree. Therefore, to make TTF work well for real data sets, a sorting of the items in the order of decreasing density should be performed in a preprocessing step.

7.3. Realistic data sets

In a third set of experiments we tested the behavior of both algorithms against widely used data sets, such as the ones contained in UCI Irvine Machine Learning repository. We have used two UCI data sets, namely, `chess` and `mushroom` with typical suggested support threshold values (70, 75, 80, 85% for chess and 20, 25, 30, 35% for mushroom). Figures 6 and 7 show the time performance of the TTF algorithm. We observe that the decrease on execution time, when increasing the support threshold, is much steeper on `chess` than it is on `mushroom` data set. This is due to the lower similarity between transactions `mushroom` compared with transactions of `chess`. This results in a larger variety of itemset frequencies for the former data set, while for the latter the vast majority of itemsets has a frequency of around 20%.

Unfortunately, we did not manage to obtain results for the PTF algorithm on these data sets because of memory overflow. This is mainly due to the particular structure of these data sets: both the data sets (especially `chess`) contain transactions that are very similar to each other and hence

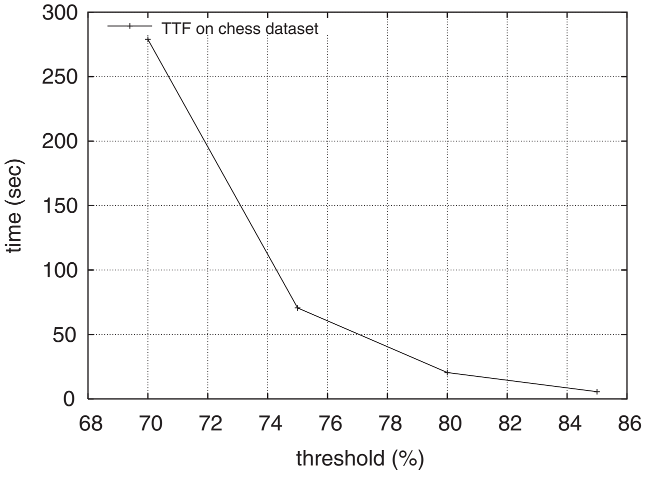

Figure 6. TTF performance on `chess` data set.

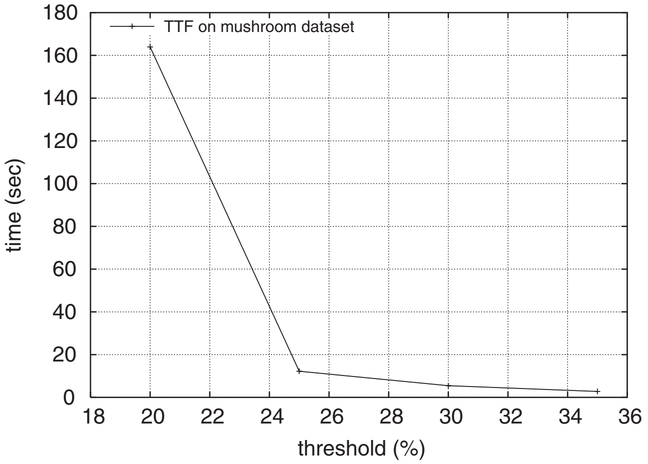

Figure 7. TTF performance on mushroom data set.

there are many frequent itemsets that are quite long (i.e. contain a large number of items) even in the case of large thresholds. Therefore, the T-tree becomes too large to fit into the main memory, as the whole tree must be stored and there are multiple pointers for each node; recall that, in contrast, in the case of TTF only the last level of the T-tree needs to be kept. Moreover, the existence of 'deep' levels in the T-tree results in huge numbers of generated subsets while traversing nodes of the P-tree, which causes considerable slowdown.

7.4. Sparse data sets

PTF, however, behaves very well when we have to work with large data sets of smaller density than the one of chess and mushroom data sets. We used the IBM Quest Generator in order to generate data sets of such structure. Figure 8 shows the behavior of both the algorithms; the superiority of PTF is clear when we have to work with such data sets.

8. CONCLUSIONS

In this work we have developed and implemented two Apriori-style algorithms for the problem of frequent itemsets generation, called TTF and PTF, which are based on the IS tree approach [11]. The two algorithms follow inverse approaches: TTF iterates over the itemsets of T-tree, and for each of them traverses the relevant part of the P-tree in order to count its total support; PTF starts by traversing the P-tree and for each visited node it updates all relevant nodes at the current level of the T-tree.

We have introduced several new techniques that result in faster algorithms compared with earlier attempts that use similar tree structures [11,13]. The most important of them are the *fixed-prefix potential inclusion* technique, which is used in algorithm TTF, and the use of *multiple pointers* in the T-tree, employed by PTF. The former allows faster support counting for P-trees that are built using only two pointers per node, thus being particularly memory efficient. The latter provides fast

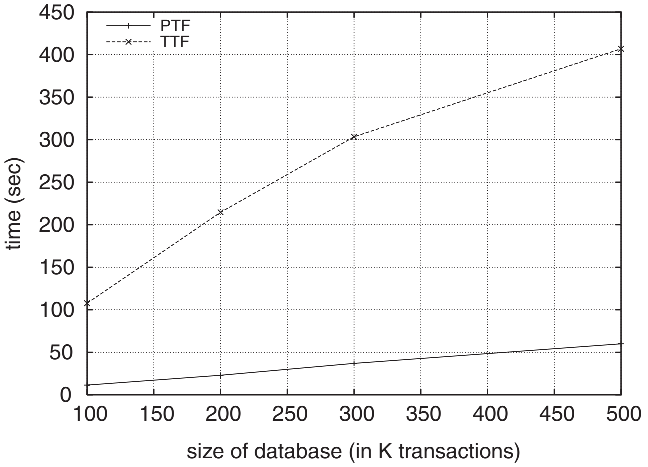

Figure 8. PTF and TTF performance on synthetic data sets with 100 items.

access to the T-tree and makes PTF a generally efficient algorithm. We show experimentally that our new algorithms achieve considerable speedup compared with the earlier algorithms.

The main difference between the two algorithms is that TTF performs a partial P-tree traversal for each potentially frequent itemset, while PTF performs only one, but full, P-tree traversal for each level of potentially frequent itemsets. As a result, PTF is considerably faster than TTF in instances where there are a lot of frequent itemsets, while TTF gains ground in instances where there are fewer potentially frequent itemsets, especially if for each of them it suffices to check only a small part of the P-tree. For example, the latter case may occur whenever item densities have a high variance. However, PTF fails to perform well in the case of long frequent itemsets because the size of the T-tree becomes prohibitive; this calls for further optimization techniques.

In conclusion, each of the two heuristics has its own merits and deserves further exploration. As a suggestion for further research, it would be interesting to investigate possible combinations of the two inverse approaches of TTF and PTF. For example, it seems reasonable to use PTF as long as the current level of the T-tree contains a lot of frequent itemsets and the level depth is small, whereas it may be wise to turn to TTF once the current level becomes sparse or if the level depth increases above a certain value.

ACKNOWLEDGEMENTS

Wojciech Rytter was supported for this work by grant N206 004 32/0806 of the Polish Ministry of Science and Higher Education. Aris Pagourtzis and Dora Souliou were supported for this research by 'Pythagoras' grant, co-financed by the European Social Fund (75%) and National Resources (25%)—Operational Programme 'Education and Initial Vocational Training' (ΕΠΕΑΕΚ II).

DOI: 10.1002/spe

REFERENCES

1. Agrawal R, Imielinski T, Swami A. Mining association rules between sets of items in large databases. *Proceedings of the ACM SIGMOD Conference on Management of Data*, Washington, DC, May 1993; 207–216.
2. Agrawal R, Srikant R. Fast algorithms for mining association rules. *Proceedings of the 20th International Conference on Very Large Data Bases* (*VLDB'94*), Santiago de Chile, Chile, Bocca JB, Jarke M, Zaniolo C (eds.). Morgan Kaufmann, 12–15 September 1994; 487–499. ISBN: 1-55860-153-8.
3. Savasere A, Omiecinski E, Navathe S. An efficient algorithm for mining association rules in large databases. *VLDB'95*, *Proceedings of 21st International Conference on Very Large Data Bases*, Zurich, Switzerland, Dayal U, Gray PMD, Nishio S (eds.). Morgan Kaufmann, 11–15 September 1995; 432–444. ISBN: 1-55860-379-4.
4. Toivonen H. Sampling large databases for association rules. *VLDB'96*, *Proceedings of 22nd International Conference on Very Large Data Bases*, Mumbai, India, Vijayaraman TM, Buchmann AP, Mohan C, Sarda NL (eds.). Morgan Kaufmann, 3–6 September 1996; 1–12. ISBN: 1-55860-382-4.
5. Bayardo RJ. Efficiently mining long patterns from databases. *SIGMOD 1998*, *Proceedings ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, U.S.A., Haas LM, Tiwary A (eds.). ACM Press, 2–4 June 1998; 85–93. ISBN: 0-89791-995-5.
6. Agrawal R, Aggarwal C, Prasad V. Depth first generation of long patterns. *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Boston, MA, U.S.A., 20–23 August 2000. ACM: New York, 2000; 108–118. ISBN: 1-58113-233-6.
7. Zaki MJ. Generating non-redundant association rules. *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Boston, MA, U.S.A. ACM, 2000; 34–43. ISBN: 1-58113-233-6.
8. Han J, Pei J, Yin Y. Mining frequent patterns without candidate generation. *ACM SIGMOD* 2000; **29**(2):1–12. ISSN 0163-5808.
9. Han J, Pei J, Yin Y, Mao R. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery* 2004; **8**:53–87.
10. Goulbourne G, Coenen F, Leng P. Algorithms for computing association rules using a partial-support tree. *Proceedings of the ES99*, Cambridge, U.K. Springer: London, 13–15 December 1999; 132–147.
11. Goulbourne G, Coenen F, Leng P. Algorithms for computing association rules using a partial-support tree. *Journal of Knowledge-Based Systems* 2000; **13**:141–149.
12. Coenen F, Goulbourne G, Leng P. Tree structures for mining association rules. *Data Mining and Knowledge Discovery* 2004; **8**:25–51.
13. Coenen F, Goulbourne G, Leng P. Computing association rules using partial totals. *Principles of Data Mining and Knowledge Discovery*, *Proceedings of the 5th European Conference*, *PKDD 2001*, Freiburg, September 2001 (*Lecture Notes in Artificial Intelligence*, vol. 2168), De Raedt L, Siebes A (eds.). Springer: Berlin, Heidelberg, 2001; 54–66.
14. Ahmed S, Coenen F, Leng P. Tree-based partitioning of data for association rule mining. *Knowledge and Information Systems* 2006; **10**:315–331.
15. Liu G, Lu H, Lou W, Yu J. On computing, storing and querying frequent patterns. *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Washington, DC, U.S.A. ACM, 20–23 August 2000; 607–612. ISBN: 1-58113-737-0.
16. Huang H, Wu X, Relue RR. Association analysis with one scan of databases. *Proceedings of the 2002 IEEE International Conference on Data Mining* (*ICDM 2002*), Maebashi City, Japan. IEEE Computer Society, 9–12 December 2002; 629–632. ISBN: 0-7695-1754-4.

DOI: 10.1002/spe