Tree structures for mining association rules

FRANS COENEN	frans@csc.liv.ac.uk
GRAHAM GOULBOURNE	graham_g@csc.liv.ac.uk
PAUL LENG	phl@csc.liv.ac.uk

Contact address: Department of Computer Science, The University of Liverpool, Liverpool L69 7ZF, UK.

Telephone: 0151 794 3698

Fax: 0151 794 3715

Email: frans@csc.liv.ac.uk

keywords: Association rules, set-enumeration tree

Tree structures for mining association rules

FRANS COENEN frans@csc.liv.ac.uk
GRAHAM GOULBOURNE graham_g@csc.liv.ac.uk
PAUL LENG phl@csc.liv.ac.uk

Department of Computer Science,University of Liverpool, UK

Abstract. A well-known approach to Knowledge Discovery in Databases involves the identification of *association rules* linking database attributes. Extracting all possible association rules from a database, however, is a computationally intractable problem, because of the combinatorial explosion in the number of sets of attributes for which incidence-counts must be computed. Existing methods for dealing with this may involve multiple passes of the database, and tend still to cope badly with densely-packed database records. We describe here a class of methods we have introduced that begin by using a single database pass to perform a *partial* computation of the totals required, storing these in the form of a set enumeration tree, which is created in time linear to the size of the database. Algorithms for using this structure to complete the count summations are discussed, and a method is described, derived from the well-known *Apriori* algorithm. Results are presented demonstrating the performance advantage to be gained from the use of this approach. Finally, we discuss possible further applications of the method.

keywords: Association rules, set-enumeration tree

1 Introduction

Knowledge Discovery in Databases (KDD) is concerned with the extraction of previously unrecognised and "interesting" information contained within (usually large) data repositories. "Interest" is of course a subjective concept, and a definition of what is interesting is required: it is usually taken as an overall measure of pattern value, combining validity, novelty, usefulness and simplicity [?]. Almost always, what is being sought is some relationship which can be observed between categories of information in the data. A particular way to describe such a relationship is in the form of an *association rule* which relates attributes within the database.

An association rule [?] is a probabilistic relationship, of the form $A \rightarrow B$, between sets of database attributes, which is inferred empirically from examination of records in the database. In the simplest case, the attributes are boolean, and the database takes the form of a set of records each of which reports the presence or absence of each of the attributes in that record. The paradigmatic example is in supermarket shopping-basket analysis. In this case, each record in the database is a representation of a single shopping transaction, recording the set of all items purchased in that transaction. The discovery of an association rule, $PQR \rightarrow XY$, for example, is equivalent to an assertion that "shoppers who purchase items P, Q

and R are also likely to purchase items X and Y at the same time". This kind of relationship is potentially of considerable interest for marketing and planning purposes.

While the shopping-basket example is useful in illustrating both the principal characteristics of the approach and its practical utility, there are many other applications which are susceptible to the method, and even when attributes are not essentially boolean, it may be possible to transform the data into a suitable form. For the purpose of this paper, however, we will assume a set I of n boolean attributes, $\{a_1, \cdots, a_n\}$. Each record in the database contains some subset of these attributes, which may equivalently be recorded as a n-bit vector reporting the presence or absence of each attribute.

An association rule R is of the form $A \rightarrow B$, where A, B are disjoint subsets of the attribute set I. The *support* for the rule R is the number of database records which contain $A \cup B$ (often expressed as a proportion of the total number of records). The *confidence* in the rule R is the ratio:

$$\frac{support\ for\ R}{support\ for\ A}$$

These two properties, support and confidence, provide the empirical basis for derivation of the inference expressed in the rule, and a measure of the interest in the rule. The support for a rule expresses the number of records within which the association may be observed, while the confidence expresses this as a proportion of the instances of the antecedent of the rule. In practical investigations, it is usual to regard these rules as "interesting" only if the support and confidence exceed some threshold values. Hence the problem may be formulated as a search for all association rules within the database for which the required support and confidence levels are attained. Note that the confidence in a rule can be determined immediately once the relevant support values for the rule and its antecedent are computed. Thus the problem essentially resolves to a search for all subsets of I for which the support exceeds the required threshold. Such subsets are referred to as "large", "frequent", or "interesting" sets.

For the databases in which we are most interested, the number of attributes is likely to be 500 or more, making exhaustive examination of all subsets computationally infeasible. In section 2 we review methods described in the literature for avoiding this exponential scaling of the search space. We go on to describe a new class of methods we have developed, which begin by performing a single database pass to carry out a *partial* computation of the support counts needed, storing these in a tree structure. Section 3 describes this structure, first presented in [?], and the algorithm for its construction, which is conservative in both space and time. In section 4 we extend the work previously presented to describe algorithms to complete the summation of support totals. These algorithms are generic, in that they can be applied in versions corresponding to many existing methods for computing association rules, making use of the tree of partial supports as a surrogate for the original database. In section 5 we describe results arising from an implementation of one such algorithm. These demonstrate that the use of the method we describe can offer significant performance advantages. Finally, we review the conclusions we draw from this research and suggest some future directions.

2 Algorithms for computing support

2.1 Exhaustive computation

It is instructive to begin by examining methods for computing the support for *all* subsets of the attribute set I. While such "brute force" algorithms are infeasible in most practical cases, they provide a benchmark against which to measure alternatives. The simplest such algorithm takes the form:

$Algorithm\ BF:$

$\forall\ records\ in\ database\ do$

$begin\ \forall\ subsets\ of\ record\ do$

$add\ 1\ to\ support - count\ for\ that\ subset;$

end

Three properties of this algorithm are of interest in considering its performance:

1. The number of database accesses required.
2. The number of computation steps involved in counting subsets of records.
3. The memory requirement.

For a database of m records, the algorithm involves a single database pass requiring m database accesses to retrieve the records. Assuming the obvious binary encoding of attributes in a record, the enumeration of subsets is straightforward, and will involve a total of up to $m \times 2^n$ computation steps, the actual number depending on the number of attributes present in each record. For each subset, the corresponding binary encoding may be used to reference a simple array of *support-counts*, so the incrementation is trivial, but will require an array of size 2^n.

For small values of n, this algorithm is simple and efficient. However, in applications such as shopping-basket analysis values of n in excess of 1000 are likely, making brute-force analysis computationally infeasible. Even in cases in which the actual number of attributes present in a record is small, making exhaustive enumeration practicable, the size of the support-count array is a limiting factor.

For these reasons, practicable algorithms for determining association rules proceed in general by attempting to compute support-counts only for those sets which are identified as potentially interesting, rather than for all subsets of I. To assist in identifying these *candidates*, it is helpful to observe that the subsets of the attribute set may be represented as a lattice. One form of this is shown as Figure **??**, for a set of four attributes, $\{A, B, C, D\}$.

For any set of attributes to be *interesting*, i.e. to have support exceeding the required threshold level, it is necessary for all its subsets also to be interesting. For example, a necessary (although not sufficient)

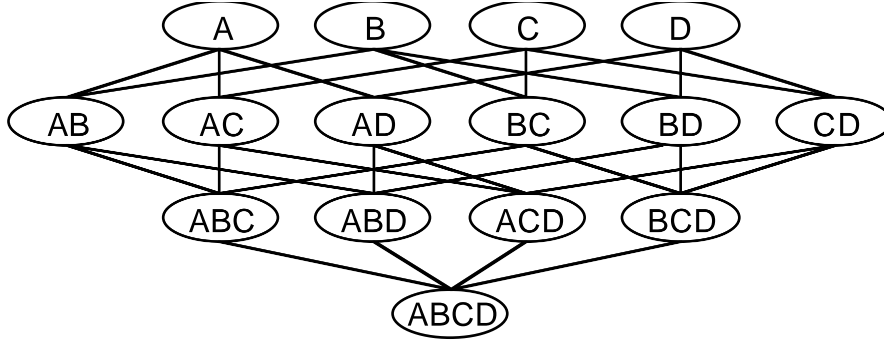


Figure 1: Lattice of subsets of $\{A, B, C, D\}$

condition for ABC to be an interesting set is that AB, AC and BC are all interesting, which in turn requires that each of A, B and C are supported at the required level. This observation, originally made in [?] and [?], provides a basis for pruning the lattice of subsets to reduce the search space; if, for example, it is known that D is not supported, then it is no longer necessary to consider AD, BD, CD, ABD, ACD, BCD or $ABCD$. Algorithms which proceed on this basis reduce their requirement for storage and computation by eliminating candidates for support as soon as they are able to do so. The tradeoff for this is usually a greater requirement for access to the database in a series of passes.

2.2 Breadth-first traversal algorithms

The best-known algorithms which reduce the search space in this way proceed essentially by breadth-first traversal of the lattice, starting with the single attributes in I. A number of such algorithms were described by Agrawal and Srikant [?]. The general form of these is that they perform repeated passes of the database, on each of which a candidate set C_k of attribute sets is examined. The members of C_k are all those sets of k attributes which remain in the search space. Initially, the set C_1 consists of the individual attributes in I. In the "Apriori" algorithm of [?], the kth cycle proceeds as follows (for k=1,2,.. until C_k = empty set):

1. Perform a pass over the database to compute the support for all members of C_k.
2. From this, produce the set L_k of interesting sets of size k.
3. Derive from this the candidate set C_{k+1}, using the *downward closure* property, i.e. that all the k-subsets of any member of C_{k+1} must be members of L_k.

Other algorithms, AIS [?] and SETM [?], have the same general form but differ in the way the candidate sets are derived. The procedure in Apriori may be visualised as a downward iteration over the lattice of figure ??. On each cycle, the support-counts for members of a row are computed, and then those which fail to reach the threshold level are used to prune the next row of the lattice.

Two aspects of the performance of this algorithm are of concern. The first is the number of passes of the database that are required, which will in general be one greater than the number of attributes in the largest interesting set. The second is the size of the candidate sets which may be generated, especially

in the early cycles of the algorithm. A number of simulation results are presented in [?], for databases of 1000 attributes within which each record is usually small and the expected maximal interesting sets have no more than 6 attributes. Even with these constraints, the sizes of C_k may initially become very large; an example illustrated, for a database with the following characteristics:

- Average record-size = 10 attributes
- Expected maximal interesting set = 4 attributes
- Support threshold = 0.75%

led to a size of C_2 in excess of $100,000$ for the best algorithm. Large candidate-set sizes create a problem both in their storage requirement and in the number of computation steps involved as each record is examined. The implementation described for the Apriori algorithm stores the candidate set in a hash-tree, which is traversed for each record in the database in turn, to identify all members of C_k contained in that record.

A variant of Apriori, AprioriTid, attempts to reduce the number of database accesses by replacing each database record by the subset of C_k whose members were included in that record. The advantage of this is that as the size of C_k reduces, database records which include no further potentially interesting sets are eliminated, making later scans of the reduced database relatively fast. In the early stages, however, while C_k is large, the modification is counterproductive because of the large expansion in the size of the restructured database.

Versions of these algorithms perform well in cases for which the candidate sets remain small enough to be contained in memory, and reduce rapidly as the iteration proceeds. It is easy to envisage instances, however, for which this may not be so. As an example, consider a database of n attributes, most of which occur individually in about 2% of all records. Suppose also that I contains a relatively small subset F of f common attributes, each of which occurs in about 50% of all records. For $n = 1000$ and $f = 40$, this corresponds to a typical "shopping basket" of about 20 common items and 20 other items. In this (not improbable) scenario, a support threshold of 0.5% would result in most of the members of $F \times I$ being included in L_2, leading to a C_3 of size about $f^2 \times \frac{n}{2}$, i.e. $800,000$ candidates. Moreover, since each pair in F is likely to occur in about 25% of records, the combinations of these with each other attribute are likely to be supported at about the 0.5% level. Hence most of $F \times F \times I$ may be included in L_3, leading to C_4 of size about $f^3 \times \frac{n}{6}$, i.e. about 10^7 in our example.

It is clear that examples of this kind may lead to candidate sets which exceed main memory capacity, or, indeed, the original size of the database. Even if this is not so, the cost of identifying candidates which are subsets of a record being examined will result in several very computationally expensive cycles of the algorithm. The usual response to an explosion in the size of candidate sets is to increase the support threshold, but this risks eliminating many potentially interesting combinations of less common attributes.

A further (related) disadvantage arises from the implication that a common support-threshold is appropriate in all cases. Intuitively, a support- level of, say, 1% for a combination of attributes $ABCDEFG$ is more interesting than the same support for the pair AB alone. Especially will this be so if the support

for $ABCDEFG$ is almost as high as that for AB, with the implication that $AB- > CDEFG$ with high confidence. If the support threshold is set low enough to catch such cases, then the price is likely to be large candidate sets in the early cycles of the algorithm.

2.3 Lattice partitioning algorithms

Breadth-first algorithms derive from the observation that sets of size k can be interesting only if all their subsets of size $k-1$ are supported at the required level. A corollary of this condition is that once a set of size k has been found to be interesting, this must be so also for all its subsets. It is tempting, consequently, to look for methods which seek to identify *maximal* interesting sets without first examining all their smaller subsets. This may be visualised as a depth-first traversal of the lattice of figure **??** to find the largest interesting set in each path. Depth-first search strategies are described by Zaki et al [?] and Bayardo [?].

An exhaustive depth-first traversal is, of course, computationally infeasible for large n. Zaki et al address this issue by partitioning the lattice using *clusters* of associated attributes. For example, suppose for the set $I = \{A, B, C, D, E\}$, the pairs AB, AC, AD, AE, BC, BD, BE and DE are all frequent sets. These can be grouped into three *equivalence classes*:

```
[A] = {B,C,D,E}
[B] = {C,D,E}
[D] = {E}
```

The members of each equivalence class are the attributes which are paired in frequent sets with the class identifier, and which follow it in the lexicographic order. Thus each equivalence class defines a subset of the search space, the members of which are all those subsets of the equivalence class concatenated with the class identifier. This in turn allows us to identify $ABCDE$, $BCDE$ and DE as a covering set of maximal potentially interesting sets, i.e. every interesting set must be a subset of at least one of these.

The clustering effect of these classes may be relatively weak: note that in this example, the set $ABCDE$ is itself in fact a covering set, and is also, of course, the maximal set of I. A more efficient *clique* clustering technique partitions the sets further, using the downward closure condition. In the above example, this allows us to identify ABC, $ABDE$, BC, BDE and DE as a set of cliques which define maximal targets for our search, within which $\{ABC, ABDE\}$ form a covering set.

A number of algorithms based on this clustering approach are described in [?]. In practice, these require a preliminary preprocessing of the database to identify L_2, the set of frequent pairs. This is then used to identify a covering set of clusters of attributes, using either equivalence class clustering or the more refined clique clustering outlined above. Each cluster generates a subset of the lattice, for which a number of traversal methods are described. A bottom-up traversal algorithm, "ClusterApr", proceeds in the same way as the Apriori algorithm described above, applied to each cluster in turn. Top-down traversal begins by examining the maximal potentially interesting set defined by the cluster, followed

recursively by all its maximal proper subsets, the recursion terminating when interesting subsets are found. A "Hybrid" algorithm is also described, which combines some elements of both approaches.

The common feature of these algorithms is that they enable each cluster to be examined in turn. This has two advantages:

1. Each cluster defines a subset of the total search space, reducing the memory required to store this and the computation involved in counting the support of its members.
2. Because each cluster includes only a subset of I, it is possible to compute the support counts using only a portion of the database.

To enable this, it is necessary for the database to be stored in inverted form, i.e. for each attribute in I to be stored with a list of records containing that attribute as a column of the database. The support for a combination of attributes can be determined by intersecting the relevant database columns. This allows the algorithms to proceed in a single pass of the database, within which the relevant columns of the database are used to determine the interesting sets in each cluster in turn. Note, however, that the computation for each cluster involves multiple intersections of columns of the database to determine the support for combinations of attributes. For large databases and/or large clusters, this may again be impossible without repeated I/O operations.

Experimental results presented in [?] demonstrate the effectiveness of these approaches in comparison with breadth-first and other methods. As with the breadth-first algorithms, however, it is not difficult to identify patterns of data which will cause problems. In particular, the approach depends critically on the ability to partition the attributes into relatively small clusters. Note that the clustering algorithms generally begin by computing L_2, which in itself involves a pass of the database and significant computation. Further problems arise, however, if L_2 is large and especially if it contains large clusters. In principle, it is possible to apply the algorithms to clusters derived from L_3 (or larger sets of attributes) rather than L_2, but this implies further preliminary passes of the database which, as for the breadth-first algorithms, can be very computationally demanding.

The MaxMiner algorithm described by Bayardo [?] also searches for maximal sets, using Rymon's set enumeration framework [?] to order the search space as a tree. This structure is central to the methods which are the basis of the present work, and will be described fully below. In respect of Max-Miner, the method has some similarities to that of Zaki et al in that the set enumeration tree defines a kind of equivalence-class ordering on the candidate sets. Max-Miner reduces this search space by pruning the tree to eliminate both supersets of infrequent sets (as in the breadth- first strategies) and subsets of frequent sets (which are known to be themselves frequent). In a development from Max-Miner, the Dense-Miner algorithm [?] imposes additional constraints on the rules being sought to reduce further the search space in these cases. These algorithms cope better with dense datasets than the others algorithms described, but again require multiple database passes.

2.4 Database partitioning algorithms

The complexity of the problem of finding association rules is essentially a product of two terms: the size of the database (m) and the size of the candidate set search space (2^n). The algorithms described above focus on the latter term, deriving a smaller search space of candidate sets which can be feasibly counted. An alternative approach is to reduce the first term, working with a subset of the database which is small enough to contain in main memory.

The Partition algorithm [**?**] divides the database into a number of non-overlapping partitions of equal size, and proceeds in two passes. In the first pass, each partition is taken in turn, and all the *locally* frequent sets computed. A locally frequent set is a set the support for which exceeds the required (proportional) threshold within at least one partition. It is easy to see that any *globally* frequent set must be locally frequent in some partition, so the set L of locally frequent sets is a superset of the set of all (globally) frequent sets. This set, L, becomes the search space in a second full pass of the database, in which the support-counts of all its members are computed.

An alternative approach, also using a subset of the database, is described by Toivonen [**?**]. In this method, a single sample of the database is taken from which is derived a candidate set for the full database search. To ensure (with high probability) that this candidate set contains all the actual frequent sets, two devices are used. Firstly, the support threshold is lowered when the database sample is processed, leading to a candidate set S which is a superset of the actual (locally) frequent set. Secondly, the set is further extended by adding its *negative border*. The negative border of the set S is the collection of sets which are not members of S, but all of whose subsets are included in S. For example, suppose $I = \{A, B, C, D, E\}$, and the maximal sets in S are ABC, AD and BD. Then the negative border of S contains the sets ABD, CD and E. Referring to the lattice of subsets of I, the negative border may be visualised as the external boundary of the subset of the lattice which has been found to be supported.

This extended set is then used as the candidate set in a single full pass of the database. If no members of the negative border are found to be frequent, then the algorithm terminates after one database pass. However, the candidate generation process cannot guarantee this, and an extra pass may be needed to deal with candidates formed by extending, in the usual way, the frequent sets found in the first pass. For the example given, if the set CD which is on the negative border is found to be frequent, in addition to the sets AC and AD, then it will be necessary to return to the database to consider the set ACD which was originally overlooked.

The advantage gained by these partitioning/sampling methods is that by working with a subset of the database which can be contained in main memory, algorithms which involve repeated access to database records become acceptable. The penalty is that the candidate set derived is necessarily a superset of the actual set of frequent sets, and may contain many "false positives". In the Partition algorithm, this will especially be the case if there are a large number of partitions and/or the database is unevenly distributed, and in these cases it may not always be possible to contain the sets in main memory. Similarly, the candidate set generated by the Toivonen method is potentially much larger than the actual frequent set. Again, for databases in which a significant number of attributes are very frequent, the algorithms

may lead to an unacceptably large search space.

3 PARTIAL SUPPORT

Most of the methods described above proceed essentially on each database pass by defining some candidate set and then examining each record to identify all the members of the candidate set that are subsets of the record, incrementing a support-count for each. The computational cost of this increases with the density of information in database records, i.e. when the average number of attributes present in a record is high, leading to an exponential increase in the number of subsets to be considered, and when candidate sets are large. In principle, however, it is possible to reduce this cost of subset-counting by exploiting the relationships between sets of attributes illustrated in the lattice (Figure **??**). For example, in the simplest case, a record containing the attribute set ABD will cause incrementation of the support-counts for each of the sets ABD, AB, AD, BD, A, B and D. Strictly, however, only the first of these is necessary, since a level of support for all the subsets of ABD can be inferred subsequently from the support-count of ABD.

Let i be a subset of the set I (where I is the set of n attributes represented by the database). We define P_i, the *partial support* for the set i, to be the number of records whose contents are identical with the set i. Then T_i, the *total support* for the set i, can be determined as:

$$T_i = \sum P_j \qquad (\forall j, j \supseteq i)$$

This allows us to postulate a general algorithm for computing total supports. Let P be the set of partial support counts P_i corresponding to sets i which appear as records in the database, and T be the set of total support counts in which we are interested (however this is defined). With the members of P and T initialised to zero:

$Algorithm\ A(Inputs:\ dataset\ DS, countset\ P, T):$

$--\ Returns\ P\ and\ T\ counting\ sets\ in\ DS\ --$

$A1:\ \forall\ records\ j\ in\ DS\ do$

$begin\ add\ 1\ to\ P_j$

$insert\ j\ to\ P$

$end;$

$A2:\ \forall\ j\ in\ P\ do$

$begin\ \forall\ i\ in\ T,\ i \subseteq j\ do$

$begin\ add\ P_j\ to\ T_i$

end

end

For a database of m records, stage 1 of the algorithm (A1) performs m support-count incrementations in a single pass, to compute a total of m' partial supports, for some $m' \leq m$. The second stage of the algorithm (A2) involves, for each of these, the examination of subsets which are members of the target set T. In an exhaustive version of the method, T will be the full set of subsets of I. If the database contains no duplicate records, then the method will be less efficient than our original "Brute Force" algorithm, which enumerates subsets of each record as it is examined. Computing via summation of partial supports, however, offers three potential advantages. Firstly, when n is small ($2^n \ll m$), then A2 involves the summation of a set of counts which is significantly smaller than a summation over the whole database. Secondly, even for large n, if the database contains a high degree of duplication ($m' \ll m$) then the stage 2 summation will again be significantly faster than a full database pass, especially if the duplicated records are densely-populated with attributes. Finally, and most generally, we may use the stage A1 to organise the partial counts in a way which will facilitate a more efficient stage 2 computation, exploiting the structural relationships inherent in the lattice of partial supports.

Consider again the lattice of Figure ??. An idealised version of our method would involve, in A1, incrementing a count at a unique position on this structure, then, in A2, summation through all the sub-lattices. For small values of n an exhaustive computation of this kind is efficient, but the method has an implied storage requirement of order 2^n and a corresponding time complexity which in general will make it infeasible. Before considering practical summation algorithms, we will first describe a method of storing the counts which has a linear storage requirement.

Figure ?? shows an alternative representation of the sets of subsets of I, for $I = \{A, B, C, D\}$, in the form of Rymon's [?] set enumeration tree. In this structure, each subtree contains all the supersets of the root node which follow the root node in lexicographic order. For convenience of drawing, we have represented the children of a node as a linked list of siblings, rather than drawing separate arcs to each child from its parent. This representation also expresses the form of an implementation which requires only two pointers to be stored at each node.

We may also see each subtree as a representation of the *equivalence class* of its root node. This organisation contains all the sets stored in the original lattice, while retaining only part of the structure of the latter. A set-enumeration tree of this kind is used in Bayardo's MaxMiner algorithm [?] to order candidate sets when searching for maximal frequent sets. In MaxMiner, the tree is ordered by increasing frequency of its elements to maximise the clustering effect obtained; this is a point we will return to later.

Using this tree as a storage structure for support-counts in stage 1 of the algorithm is straightforward and computationally efficient: locating the required position on the tree for any set of attributes requires at most n steps. We may go further, however, and begin to take advantage of the structural relationships implied by the tree to begin the computation of total supports. This is because, in locating a node on the tree, the traversal will pass through a number of nodes which are subsets of the target node. In doing so, it is inexpensive to accumulate *interim* support-counts at these nodes. A (stage 1) algorithm for this has the following form:

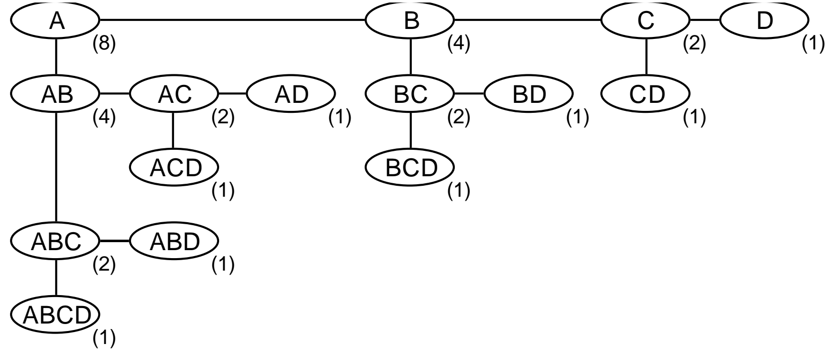


Figure 2: Tree storage of subsets of $\{A, B, C, D\}$

$Algorithm\ B1\ (Inputs:\ dataset\ DS,\ P-tree\ P):$

$--\ Returns\ P-tree\ with\ interim\ counts\ added\ --$

$\forall\ nodes\ i\ in\ P\ do\ Q(i) = 0$

$\forall\ records\ j\ in\ DS\ do$

$i = root\ node\ of\ P:$

$while\ i\ not\ null\ do$

$begin\ if\ j \supseteq i\ then\ increment\ Q(i);$

$if\ j = i\ then\ exit$

$else\ if\ j \supseteq i\ then\ i = eldestchildof(i)$

$else\ i = youngersiblingof(i);$

end

This algorithm will compute interim support-counts Q_i for each subset i of I, where Q_i is defined thus:

$$Q_i = \sum P_j \qquad (\forall j, j \supseteq i, j\ follows\ i\ in\ lexicographic\ order)$$

It then becomes possible to compute total support using the equation:

$$T_i = Q_i + \sum P_j \qquad (\forall j, j \supset i, j\ precedes\ i\ in\ lexicographic\ order)$$

The numbers associated with the nodes of Figure **??** are the interim counts which would be stored in the tree arising from a database the records of which comprise exactly one instance of each of the 16 possible sets of attributes; thus, for example, $Q(BC) = 2$, derived from one instance of BC and one of BCD. Then:

$$\begin{aligned} T(BC) &= Q(BC) + P(ABC) + P(ABCD) \\ &= Q(BC) + Q(ABC) \end{aligned}$$

To store counts for the complete tree, as in Figure ??, of course implies a storage requirement of order 2^n. We can avoid this, however, by observing that for large n it is likely that most of the subsets i will be unrepresented in the database and will therefore not contribute to the partial-count summation. A version of the algorithm to exploit this builds the tree dynamically as records are processed, storing interim totals only for records which appear in the database. Nodes are created only when a new subset i is encountered in the database, or when two siblings i and j share a leading subset which is not already represented. The latter provision is necessary to maintain the structure of the tree as it grows. The formulae for computing total supports still apply, and we need only to sum interim supports that are present in the tree.

We will use the term *P-tree* to refer to this incomplete set-enumeration tree of interim support-counts. A detailed description of the algorithm (C1) for building the P-tree in a single database pass is given in [?]. Building the tree dynamically in this way implies a storage requirement of order m rather than 2^n. This will be reduced further, perhaps substantially, if the database contains a high incidence of duplicates. The computation involved in creating the tree is also linear in n and m.

A simple example illustrating the generation of a P-tree is given in figure ??, illustrating the tree derived from the four-attribute sample database given in Table 1.

	ABCD
A	1000
ABC	1110
BD	0101
CD	0011
BCD	0111
BD	0101
BD	0101
ACD	1011

Table 1

The algorithm begins by creating the node A with support 1, then adds ABC as a child of this node incrementing the support for A by one en route. BD is added, initially, as a sibling of A, but when BCD is added, the "dummy" node B is created, with BCD and BD becoming its children. In this manner we prevent the tree from degenerating into a linked-list. The dummy node B initially inherits the support of its children, and subsequently is treated as a normal node.

A rather similar structure to our *P-tree* has been described independently in contemporaneous work reported in [?]. This structure, the *FP-tree*, has a different form but quite similar properties to the P-tree,

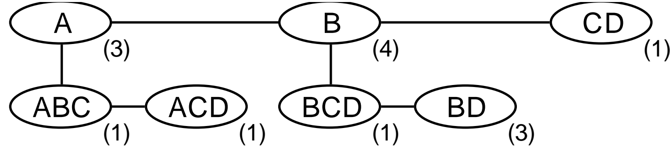


Figure 3: P-tree generation example

but is built in two database passes, the first of which eliminates attributes that fail to reach the support threshold, and orders the others by frequency of occurrence. Each node in the FP-tree stores a single attribute, so that each path in the tree represents and counts one or more records in the database. The FP-tree also includes more structural information, including all the nodes representing any one attribute being linked into a list. This structure facilitates the implementation of an algorithm, "FP- growth", which successively generates subtrees from the FP-tree corresponding to each frequent attribute, to represent all sets in which the attribute is associated with its predecessors in the tree ordering. Recursive application of the algorithm generates all frequent sets.

Unlike the P-tree, which is essentially a generic structure which can subsequently be used as a basis for many algorithms to complete the count-summation, the FP-tree is closely tied to the FP-growth algorithm. Results presented for FP-growth demonstrate its effectiveness in cases when the tree is memory-resident, but the linkage between nodes of the tree makes it difficult to effect a comparable implementation when this is not the case. The problem is exacerbated because of the additional storage overheads of the FP-tree, which in general will store many more nodes than the P-tree, and needs more pointers to be stored at each node. The simpler P-tree structure, conversely, requires only that each node be linked to its parent. In fact, in the algorithm we describe in the next section, completion of the summation is effected by processing the nodes of the tree in any order, so in this case it is possible to store the "tree" on disk as an unordered and unlinked set of nodes. In this case the implementation scales to deal with non-store-resident data simply and efficiently.

4 COMPUTING TOTAL SUPPORTS

Algorithm C1 essentially performs, in a single pass, a reorganisation of the relevant information in the database into a sparse set-enumeration tree, the nodes of which record interim support-counts only for those sets which appear as distinct records in the database or are parent nodes required to maintain the tree structure. For any candidate set T of subsets of I, the calculation of total supports can be completed by walking this tree, adding interim supports as required according to the formulae above.

Consider again the set BC with reference to Figure **??**. The stage 1 computation has already added into the interim support-count the contribution from supersets which follow BC in the tree-ordering, i.e. the contribution from BCD. To complete the summation of total support, we must add in any contribution from preceding supersets, i.e. ABC and $ABCD$. But any contribution from $ABCD$ has already been accumulated in ABC, so the latter is the only addition needed. Now consider the converse

of this. The partial total accumulated at $ABCD$ makes a contribution to the total support for all the subsets of $ABCD$. However, the contribution in respect of the subsets of ABC is already included in the interim total for ABC, so, when considering the node $ABCD$, we need examine only those subsets which include the attribute D. In general, for any node j, we need consider only those subsets of j which are not also subsets of its parent.

Thus, the following algorithm completes the summation of total supports for a candidate set T:

$$Algorithm\ C2\,(Inputs\ P-tree\ P, candidate\ set\ T):$$
$$--\ Returns\ counts\ T_i\ for\ all\ sets\ i\ in\ T\ --$$
$$\forall\ sets\ i\ in\ T\ do\ T_i = 0$$
$$\forall\ nodes\ j\ in\ P\ do$$
$$begin\ k = j - parent\,(j)\,;$$
$$\forall\ i\ in\ T\ ,\ i \subseteq j,\ i \cap\ k\ not\ empty,\ do$$
$$begin\ add\ Q_j\ to\ T_i$$
$$end$$
$$end$$

Note that this algorithm is essentially generic, in that it can be applied in different versions depending on how the candidate set is defined. In general, methods like Apriori which involve multiple database passes should gain from using the P-tree to replace the original database, for two reasons: firstly, because any duplicated records are merged in the P-tree structure, and secondly, because the partial computation incorporated in the P-tree is carried out once only, when the tree is constructed, but reduces the computation required on each subsequent pass. Thus, in the second pass of Apriori, using the P-tree would in the best case require only $r-1$ subsets of a record of r attributes to be considered (i.e. those not covered by its parent) rather than the $r(r-1)/2$ required by the original Apriori. For example, consider the case of a set $ABCD$ that is present in the P-tree as a child of ABC. When the parent node ABC is processed in the course of algorithm C2, its interim support will be added to the total support of all its subsets. But, this interim support incorporates the support for its child $ABCD$. So, when the latter node is processed, we need add its support only to that for the sets AD, BD and CD that have not already been counted. This advantage becomes greater, of course, the larger the sets being considered. It is this property that is the key performance advantage gained from using the P-tree: although the candidate set being considered is not reduced, the number of candidates within the set that need be examined at each step is reduced, reducing the overall time required to search the candidate set.

We can also take advantage of the structure of the P-tree to organise the candidate set to enable computation of total supports to be carried out more efficiently. Figure **??** illustrates the dual of Figure **??**,

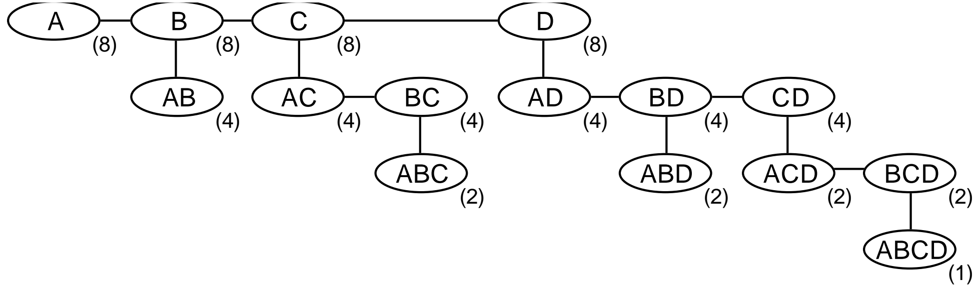


Figure 4: Tree with predecessor-subtrees

in which each subtree includes only supersets of its root node which contain an attribute that precedes all those of the root node. We will call this the T-tree, representing the target sets for which the total support is to be calculated, as opposed to the interim-support P-tree of Figure **??**. Observe that for any node t in the T-tree, all the subsets of t which include an attribute i will be located in that segment of the tree found by a depth-first traversal starting at node i and finishing at node t. This allows us to use the T-tree as a structure to effect an implementation of algorithm C2:

$Algorithm\ TFP\ (Compute\ Total - from\ Partial - supports):$

$(Inputs\ P - tree\ P,\ T - tree\ T)$

$- - Returns\ total\ support\ counts\ for\ a\ given\ T - tree - -$

$\quad \forall\ nodes\ j\ in\ P\ do$

$\quad\quad begin\ k = j - parent(j);$

$\quad\quad\quad i = first\ attribute\ in\ k;$

$\quad\quad\quad starting\ at\ node\ i\ of\ T\ do$

$\quad\quad\quad t = i$

$\quad\quad\quad Countsupport(\ Inputs\ set\ j, T - tree\ T, startnode\ t):$

$\quad\quad\quad\quad while\ t\ not\ null\ do$

$\quad\quad\quad\quad begin\ if\ t \subseteq j\ then\ add\ Q_j\ to\ T_t;$

$\quad\quad\quad\quad\quad\quad if\ t = j\ then\ exit$

$\quad\quad\quad\quad\quad\quad else\ t = eldestchildof(t); Recursivelycall\ Countsupport(j, T, t)$

$\quad\quad\quad\quad\quad t = youngersiblingof(t);$

$\quad\quad\quad\quad end$

$\quad\quad end$

Again, of course, to construct the entire T-tree would imply an exponential storage requirement. In any practical method, however, it is only necessary to create that subset of the tree corresponding to the

current candidate set being considered. Thus, a version of the Apriori algorithm using these structures will consider candidates which are singletons, pairs of attributes, triples, etc., in successive passes. This algorithm, Apriori-TFP, has the following form:

1. $K = 1$
2. Build level K in the T-tree.
3. "Walk" the P-tree, applying algorithm TFP to add interim supports associated with individual P-tree nodes to the level K nodes established in (2) .
4. Remove any level K T-tree nodes that do not have an adequate level of support.
5. Increase K by 1.
6. Repeat steps (2), (3), (4) and (5); until a level K is reached where no nodes are adequately supported.

The algorithm begins by constructing the top level of the T-tree, containing all the singleton subsets, i.e. the single attributes in I. A first pass of algorithm TFP then counts supports for each of these in a single traversal of the P-tree:

$$
\begin{array}{l}
\forall P \in Ptree \\
\quad P' = P \setminus P_{parent} \\
\quad \forall T \in Ttree\ where\ \ (level(T) \equiv 1) \\
\quad\quad if\ \ (T \subset P')\ \ T_{sup} = T_{sup} + P_{sup}
\end{array}
$$

Note again that identification of the relevant nodes in the T-tree is trivial and efficient, as these will be located in a (usually short) segment of the level-1 list. In practice, it is more efficient to implement level 1 of the T-tree as a simple array of attribute-counts, which can be processed more quickly than is the case for a list structure. A similar optimisation can be carried into level 2 and succeeding levels, replacing each branch of the tree by an array, and again this will be more efficient when most of the level 2 nodes remain in the tree, and continue to be an effective device as long as its storage requirements remain practicable. In this form, the T-tree is not fundamentally different from the hash-tree used in [**?**], except that the ordering of the structure provides a more effective localisation of the candidates we need to count.

Following completion of the first pass, the level 1 T-tree is pruned to remove all nodes that fail to reach the required support threshold, and the second level is generated, adding new nodes only if their subsets are contained in the tree built so far, i.e. have been found to have the necessary threshold of support. This procedure, applied similarly at each subsequent level, implements the "downward closure" property of Apriori. The new level of the tree forms the candidate set for the next pass of the algorithm TFP. As

in the first pass, this proceeds via a traversal of the P-tree, from each node P of which is constructed a *search node* P' made up of those digits that are in P but not in P_{parent}.

The algorithm for levels after the first comprises two parts:

Part 1 Process the top level of the T-tree and identify those nodes which merit consideration (by virtue of being a subset of the search node P').

Part 2 Proceed down the child branches of each of the nodes identified in Part 1 until the required level is reached and then apply the appropriate support updates.

The search continues in this manner until there are no characters left in the search node. The complete algorithm is described formally in Table 2 (Part 1) and Table 3 (Part 2). In these descriptions, $T_{i,j}$ denotes a node at level i of the T-tree.

$Inputs\ P - tree\ Ptree, integer\ requiredLevel, T - tree\ Ttree$

$\forall P \in Ptree\ \ where\ \ (numDigits(P) \geq requiredLevel)$

$\quad P' = P \setminus P_{parent}$

$\quad \forall T_{1,j} \in Ttree\ \ \ (nodes\ at\ level\ 1)$

$\quad\quad loop\ while\ P' \neq null$

$\quad\quad\quad if\ T_{1,j} < P'\ \ j++$

$\quad\quad\quad if\ T_{1,j} \equiv P'$

$\quad\quad\quad\quad if\ (requiredLevel \equiv 1)\ \ T_{sup} = T_{sup} + P_{sup}$

$\quad\quad\quad\quad else$ **Part 2** $(currentLevel = 2)$

$\quad\quad\quad\quad P' = null$

$\quad\quad\quad if\ \ (T_{1,j} \subset P')$

$\quad\quad\quad\quad if\ (requiredLevel \equiv 1)\ \ T_{sup} = T_{sup} + P_{sup}$

$\quad\quad\quad\quad else$ **Part 2** $(currentLevel = 2)$

$\quad\quad\quad\quad P' = P' \setminus firstDigit(P') \cdot j++$

Table 2: Total Support Algorithm (Part 1)

Part 2 is only invoked when the required level is not level 1 and some level 1 T-tree node is equal to or a subset of P'. Again we wish to minimise the work required to search the subtree; to do this we can again make use of the lexicographic ordering of the nodes. We proceed in a similar manner to that described for identifying individual branches but using a termination search node equivalent to the last N digits of the discovered P-tree node (where N is equivalent to the current level in the branch of

the T-tree we are investigating). Thus if P is equivalent to ABC and we are currently at level two (the "doubles" level) the termination search node (P'') will be equivalent to BC; and so on. We define a function, *endDigits*, that takes two arguments P and N (where N is the current level) and returns an identifier comprising the last N digits of P; thus $endDigits(ABC, 2) = BC$. The significance of this is that BC is the last subset of ABC at level 2 that need be considered.

The algorithm is described formally in Table 3. The recursive calls in each case terminate when the required level is reached, at which point we step along the nodes at this level, updating the supports as appropriate, until the termination search node P'' is reached.

Input integer currentLevel

$P'' = \ endDigits(P, currentLevel)$

$i = currentLevel$

$loop\ while\ T_{i,j} \neq null$

  $if\ T_{i,j} < P''$

    $if(T_{i,j} \subset P)$

      $if\ currentLevel \equiv requiredLevel\ \ T_{sup} = T_{sup} + P_{sup}$

      $else\ recursively\ call\ Part\ 2\ with\ currentLevel = i++$

    $j++$

  $if\ T_{i,j} \equiv P''$

      $if\ currentLevel \equiv requiredLevel\ \ T_{sup} = T_{sup} + P_{sup}$

      $else\ recursively\ call\ Part\ 2\ with\ currentLevel = i++$

    $stop$

  $if\ T_{i,j} > P''$

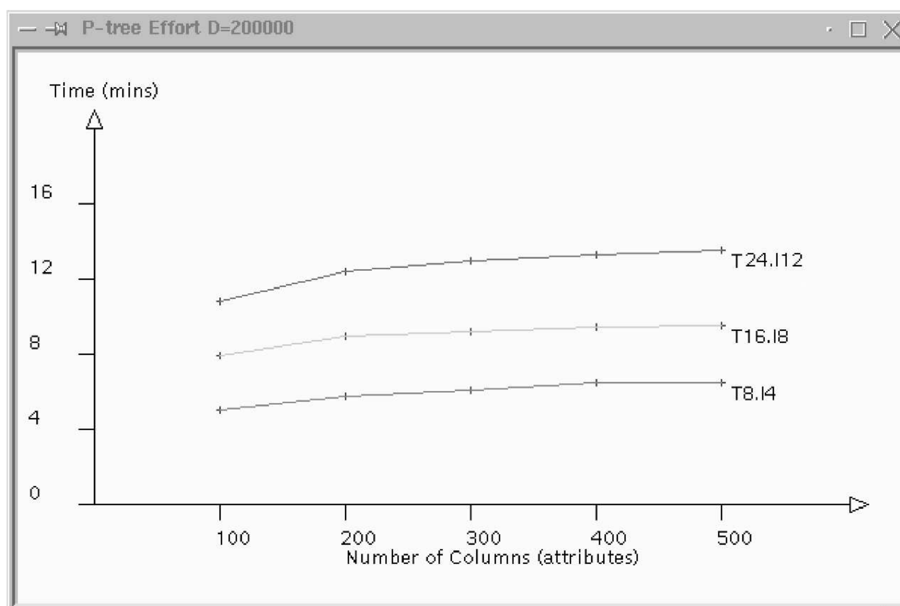
    $stop$

Table 3: Total Support Algorithm (Part 2)

5 RESULTS

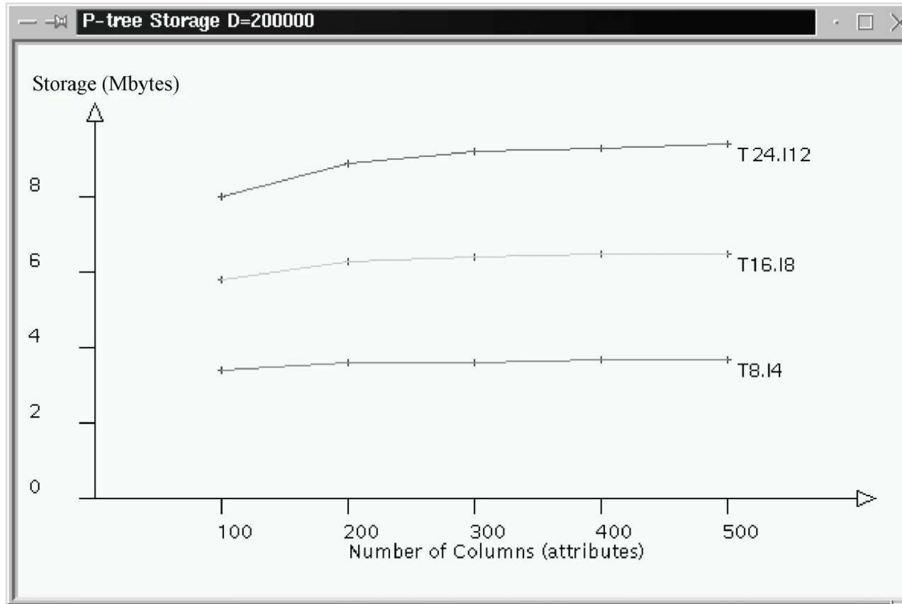
To evaluate the algorithms we have described we have used synthetic databases constructed using the generator described in [?]. This uses parameters T, which defines the average number of attributes found in a record, and I, the average size of the maximal supported set. Higher values of T and I in relation to the number of attributes N correspond to a more densely-populated database.

The first set of experiments illustrate the performance characteristics involved in the creation of the P-tree, arising from an unoptimised, low-performance prototype implementation in Java. Graph 1 shows the time to build the tree, and Graph 2 its storage requirements, for databases of 200,000 records with varying characteristics. These results show that the cost of building the P-tree is almost independent of N, the number of attributes. The three cases illustrated represent different combinations of the parameters T and I as defined above. As is the case for all association-rule algorithms, the cost of the P-tree is greater for more densely-populated data, but in this case the scaling appears to be linear. Graph 3 and Graph 4 examine the performance for databases of 500 attributes, with the same sets of parameters, as the number of database records is increased. These show, as predicted, that the size of the tree and its construction time are linearly related to the database size.

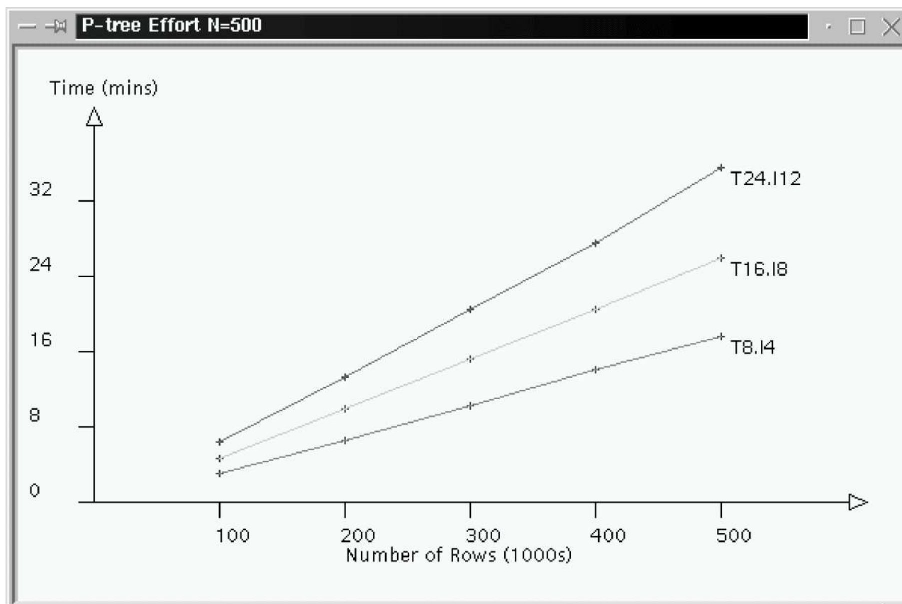


Graph 1. Effort (time) to generate P-tree for data sets with number of rows fixed at 200000

These results all relate to cases in which it is possible to build the complete P-tree in main memory. For larger databases, however, it is easy to describe a construction process which will build separate trees for manageable segments of the database, prior to a final merging into a single tree. Note also, again, that the Total-support algorithms we have described do not require the P- tree to be traversed in any particular order. Indeed, it is not necessary to maintain the tree structure in storage; the only structural information required by Algorithm TFP is the difference of a node from its parent. Provided this information is stored with each node, then the "tree" can finally be stored as a simple array and processed element-by-element in any order. These properties make it straightforward to envisage efficient implementations for cases in which the data is too large to be contained in main memory. A simple implementation would construct a separate P-tree for store-resident partitions of the data, storing each on disk in array form, and would then proceed to apply Apriori-TFP to this set of stored nodes. Note that the algorithm will work correctly even if the same set is represented in more than one tree, and in this simplest form of implementation, it will probably not be cost-effective to merge the separate trees.



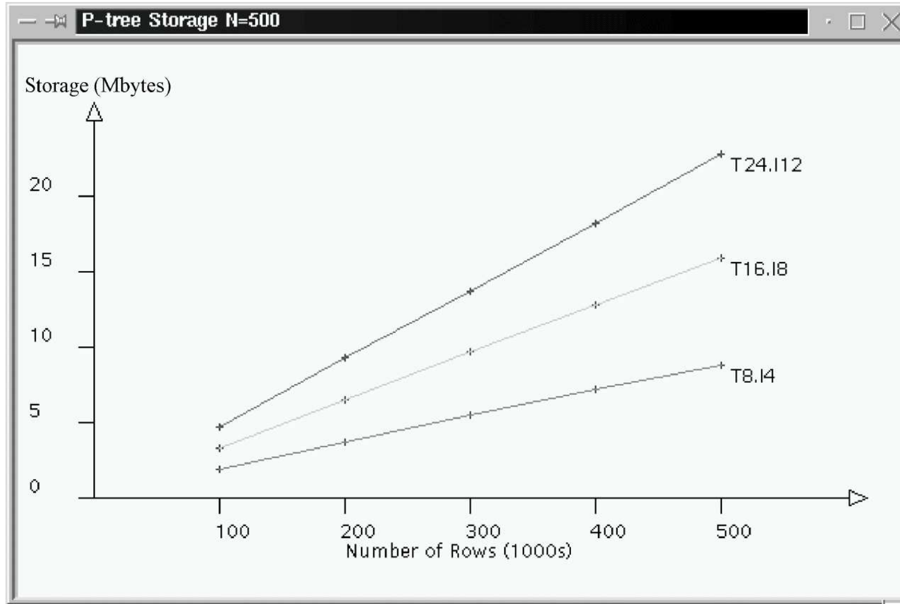
Graph 2. P-tree storage requirements for data sets with number of rows fixed at 200000



Graph 3. Effort (time) to generate P-tree for data sets with number of columns fixed at 500

Finally, to evaluate the performance of the method for computing final support-counts, we have compared the Apriori-TFP algorithm we have described with a version of the original Apriori. To provide a basis for comparison, this implementation of "Apriori" also uses the same T-tree structure as Apriori-TFP for storing the candidates being considered with their support- counts, rather than the hash-tree structure described in [?]. The candidate generation method is the same for both algorithms, so the comparison examines only the effect of using the Apriori-TFP algorithm with the P-tree as a surrogate for the database, instead of the original Apriori applied to the original database. This effect arises from the lower number of support-count updates that may be required when examining a P-tree node, as opposed to the number required in Apriori from examination of the records from which the node

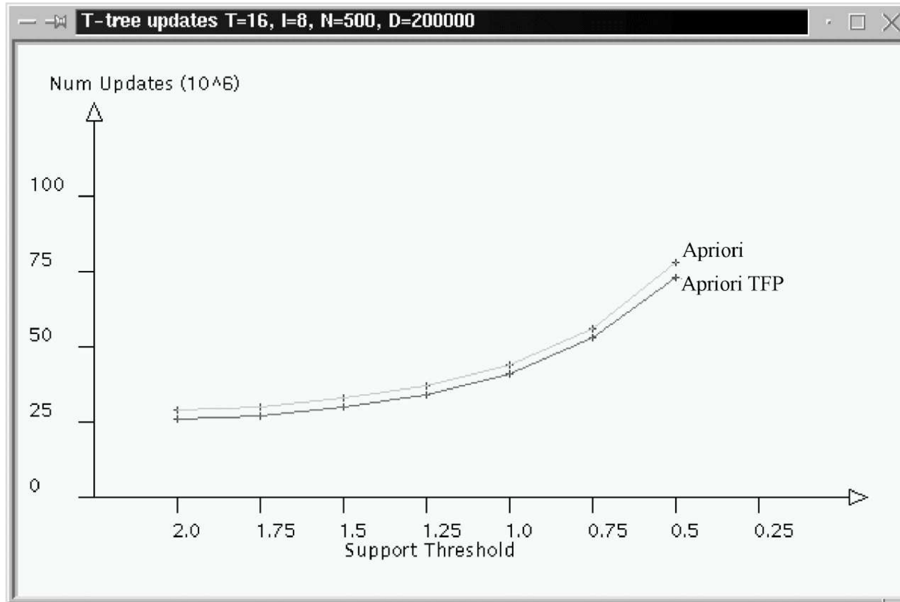
is made up.



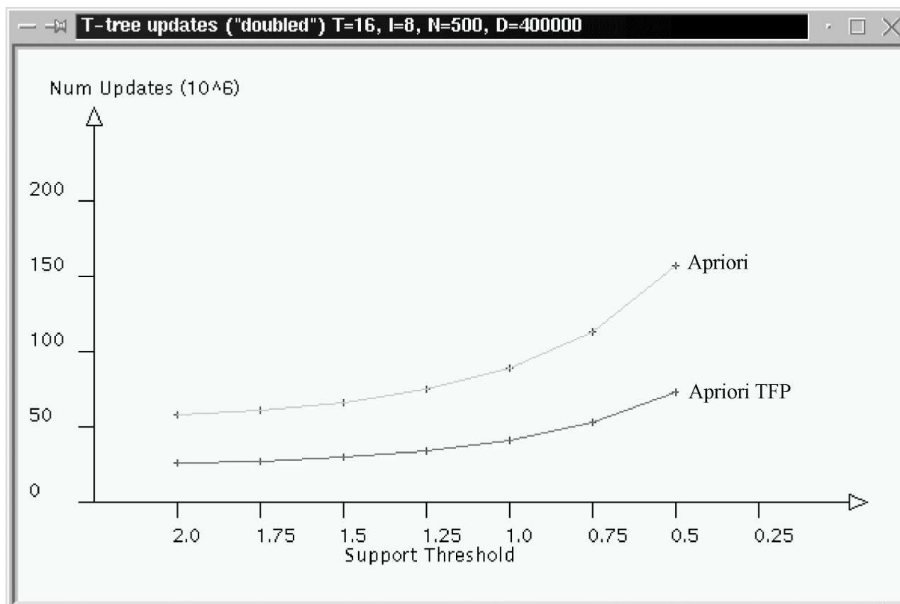
Graph 4. P-tree storage requirements for data sets with number of columns fixed at 500

Graph 5 plots the number of support-count updates for the two methods, for databases of 200,000 records and 500 attributes, with the parameters $T = 16$, $I = 8$, as the support threshold is varied from 2% down to 0.5%. As can be seen, the Apriori-TFP algorithm shows a consistent but relatively small advantage in all cases. The gain is slight because a data set of this relatively low density is likely to contain very few duplicated records, and will also tend to have a rather sparse distribution of sets. Using the P-tree, conversely, will be most effective when there are clusters of records including significant numbers of shared attributes (as we might hope to find when mining potentially interesting data), and, especially, if there are significant numbers of duplicated records. A simple illustration of the latter effect is provided in Graph 6, which shows the results obtained when the dataset of Graph 5 is simply doubled, to create a database of 400,000 records, in which each distinct record occurs at least twice. In this rather artificial example, the P-tree constructed will be exactly the same as that for the original dataset of Graph 5, giving rise to the same performance curve for Apriori-TFP. The Apriori algorithm, conversely, takes no advantage from the duplication of records.

The performance of the method when applied to much more densely-packed data is illustrated in Graph 7, which uses the parameters $T = 10$, $I = 5$, for a database of 200,000 records and only 20 attributes, so that the typical record will include half of all attributes. With this high density of data, the number of updates increases corresponding to the increase in the size of the candidate sets. For Apriori-TFP, however, this increase is offset by savings resulting from the number of duplicates and near-duplicates in the P-tree, giving rise to a significant performance advantage.



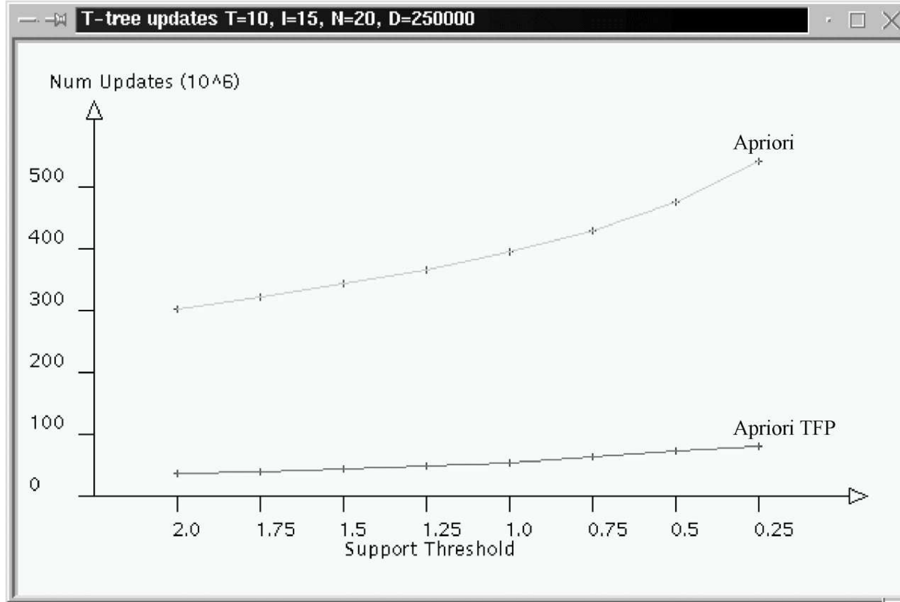
Graph 5. Number of T-tree node support count updates (T16.I8.D200k.N0.5k)



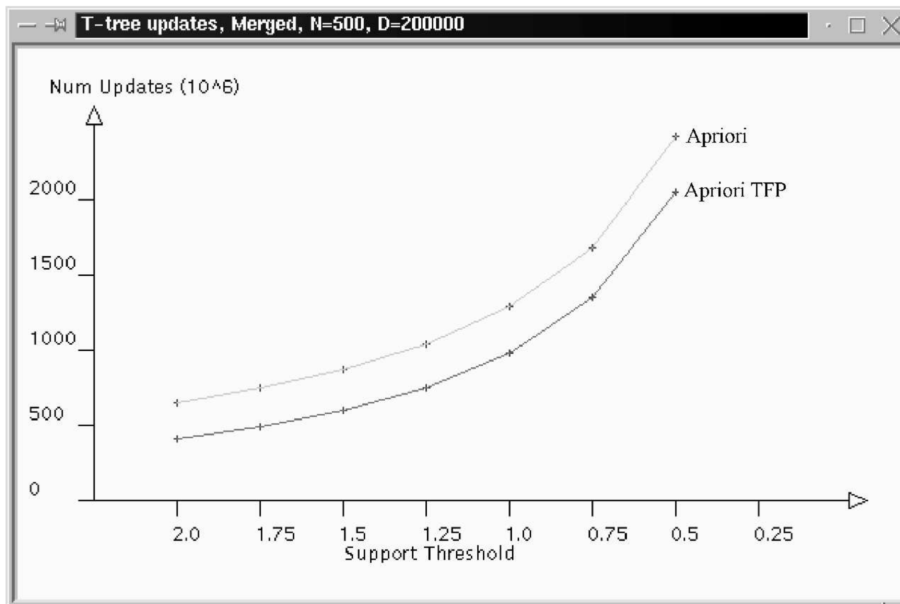
Graph 6. Number of T-tree node support count updates (T16.I8.D200k.N0.5k doubled, i.e. T16.I8.D400k.N0.5k)

Of course, the number of attributes in this example is too small to demonstrate a practical advantage from the method; we include it only to illustrate that using the P-tree will tend to show markedly greater advantage when applied to the more dense datasets which give rise to the greatest problems for all methods. A more realistic case is provided by the final experiment, illustrated in Graph 8. Here, the data generated for the experiments of Graph 5 and Graph 7 was merged, to create a database of 200,000 records and 500 attributes, in which each record comprises the union of one record taken from each of the original databases. This exemplifies a case in which a relatively small subset of a large set of attributes occurs much more frequently than the others. Note, however, that the method by which this data was

constructed will give rise to no more duplicates than occurred in the data of Graph 5. The advantage for Apriori-TFP in this case arises almost wholly because of the clustering effect in the P-tree induced by the subset of more common attributes. The gain, although less than for the very dense data of Graph 7, is still substantial.



Graph 7. Number of T-tree node support count updates (T10.I5.D200k.N0.02k)



Graph 8. Number of T-tree node support count updates using a dataset produced by merging (Graph 5) T16.I8.D200k.N0.5k and T10.I5.D200k.N0.02k (Graph 7)

6 CONCLUSIONS

We have presented here an algorithm for computing support counts using as a starting point an initial, incomplete computation stored as a set- enumeration tree. Although the actual algorithm used to compute

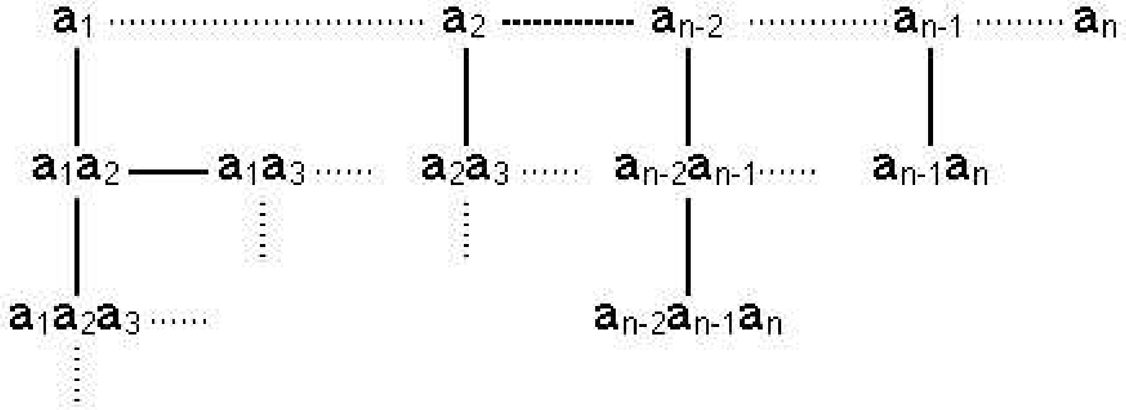


Figure 5: Partitioned support-count tree

the final totals is based on the Apriori algorithm, the method itself is generic, in that, once the P-tree has been created, a variety of methods may be applied to complete the summation. Many of these methods, like the one we have illustrated, will be able to take advantage of the partial computation already carried out in the initial database pass to reduce the cost of further multiple passes.

Note, however, that the advantage gained from this partial computation is not equally distributed throughout the set of candidates. We have observed previously that the tree structure of Figure **??** may be seen as an organisation of the lattice into equivalence classes. The interim support calculation of algorithm C1 computes the support for a subset from within its equivalence class, i.e. the support deriving from the subtree of which it is the root. For sets early in the lexicographic order, most of the support calculation is completed at this stage; in particular, for the attributes of Figure **??**, support for the sets A, AB, ABC and $ABCD$ will be counted totally in stage 1 of the summation.

We can increase the proportion of the summation which is completed in stage 1 by a partitioning of the tree. For example, it would be possible to separate the tree of Figure **??** into four subtrees, rooted at the nodes A, B, C and D, and for stage 1 of the algorithm (C1) to accumulate interim supports within each of these subtrees independently. In this case, a record containing the set ABD, for example, would increment the support-counts for ABD within the A-tree, BD within the B-tree, and D within the (single-node) D-tree. The advantage of this is that completion of the summation, to obtain total supports, can also be carried out independently within each of the subtrees used for the interim summation, rather than across the whole structure.

More generally, suppose our attribute-set $I = \{a_1, \cdots, a_n\}$ is partitioned into a disjoint set of subsets, retaining the original order; i.e each subset i may be a single attribute, or a sequence of attributes from within the original ordering. Corresponding to each subset i we will define a set-enumeration tree, labelled with i, within which supports for the equivalence classes of the members of i will be counted. For this, the following generic algorithm may be described:

$$Algorithm\ D(Inputs : dataset\ DS, set\ of\ P-trees\ PP, T-tree\ T) :$$

$--\ Returns\ total\ support\ counts\ for\ a\ given\ T-tree\ --$

$D1: \forall\ records\ j\ in\ DS\ do$

$begin\ j' = j$

$\forall\ trees\ i\ in\ PP\ while\ j'\ not\ empty\ do$

$begin\ if\ i \cap j'\ not\ empty$

$then\ increment\ support-count\ for\ j'\ within\ i-tree;$

$j' = j' - i$

$endif$

end

$end;$

$D2: \forall\ trees\ i\ in\ PP\ do$

$C2(i, T)$

$--compute\ total\ support\ for\ all\ sets\ within\ tree\ i\ --$

The algorithm is generic in two respects. Firstly, the degree of partitioning is undefined, so that, at one extreme, only one tree is used, representing the full attribute-set: in this case, the algorithm corresponds to C. Conversely, separate trees may be defined for each attribute of I. Secondly, the actual procedures used to increment partial support-counts in D1, and subsequently to complete the computation in D2, are essentially orthogonal to this algorithm, so it is possible to envisage the application of different algorithms to the computation of supports within each of the trees defined. For example, it would be possible to apply an exhaustive version of algorithm A to part of the structure, and the Apriori-TFP algorithm to another, or indeed to use other storage structures and counting procedures. In general, each "tree" may in fact be any storage organisation within which partial supports for a class of subsets are counted.

Consider the case in which separate trees are defined for each attribute, illustrated in Figure **??**. The size of tree required for the complete storage of support-counts for the equivalence class of attribute a_i is 2^{n-i}. However, if the support-count storage is conservative, i.e. counts are accumulated only for the sets of attributes j' that are encountered as the database is examined, then the storage for the equivalence-class of a_i will be of order m', where $m' \leq$ the number of records in the database which contain a_i (again, reduced by the existence of duplicates). Thus, the storage requirement for each equivalence class is less than or equal to min $\{2^{n-i}, T_{a_i}\}$.

Partitioning the tree in this way allows us (in one pass) to organise the data into sets each of which can be processed independently and may be small enough to be retained in central memory. At the high-order end of the organisation, i.e. for values of i close to n, the 2^{n-i} limit becomes computable. Thus, for large i, the appropriate storage regime may be a complete array of subset-counts, using the exhaustive algorithm A for efficient counting. Conversely, for small i, the conceptual support-count tree

is large but sparse, enabling us to employ a method such as Apriori-TFP or another conservative storage method which will record only those sets that are encountered.

The gain from this approach can be maximised by applying an ordering heuristic similar to that used by Bayardo [?] and also in DIC [?]. Suppose that the order of frequency of each attribute is known, at least approximately. Then, if we order I so that a_1 is the least common attribute, through to a_n, the most common, then the most common attributes will be clustered at the high-order end of the structure of Figure **??**, at which efficient exhaustive computation is feasible. Conversely, the sparseness of storage at the low-order end of the structure is increased, enabling a number of conservative storage strategies to be considered.

Essentially, this approach is a combination of two heuristics:

1. For combinations of a small set of very frequent attributes, the exhaustive computation of total supports from partial supports using algorithm A is efficient.
2. For less frequent sets of attributes, a partitioning of the database corresponding to equivalence classes can be done in a single pass. Each resulting partition will contain at most T_{a_i} records, and can subsequently be processed independently.

Algorithms using this approach may therefore be categorised as 1+ pass. In one full pass of the database, computation of support for the commonest attributes is completed, while at the same time the database is reorganised into partitions to facilitate consideration of less common attributes. Because each of these partitions contains a relatively small fraction of the database, it may become possible to carry out all the stage 2 summations, for each partition in turn, within main memory. A further advantage is that, as the most common attributes have been processed separately, many of the remaining database partitions may contain relatively few frequent sets. For example, it may be that within the a_1-partition of Figure **??**, the only set reaching the support threshold is the root node, a_1 itself. This likelihood allows us to consider various alternative heuristic storage and evaluation strategies for dealing with these partitions efficiently. For example, the a_1 "tree" may in fact be most conveniently stored as a simple unordered set of records which is processed subsequently with the expectation that it will contain only a few short combinations that are frequent.

7 Acknowledgements

Graham Goulbourne was supported in this work by a PhD studentship awarded by Royal and Sun Alliance Ltd.

References

[Agrawal et al. 1993] Agrawal, R. Imielinski, T. Swami, A. Mining Association Rules Between Sets of Items in Large Databases. Proc ACM SIGMOD-93, 207-216. May 1993.

[Agrawal and Srikant 1994] Agrawal, R. and Srikant, R. Fast Algorithms for Mining Association Rules. Proc 20th VLDB Conference, Santiago, 487-499. 1994.

[Bayardo 1998] Bayardo, R.J. Efficiently mining long patterns from databases. Proc ACM-SIGMOD Int Conf on Management of Data, 85-93, 1998.

[Bayardo et al. 1999] Bayardo, R.J., Agrawal, R. and Gunopolos, D. Constraint-based rule mining in large, dense databases. Proc 15th Int Conf on Data Engineering, 1999.

[Brin et al. 1997] Brin S., Motwani, R., Ullman, J.D. and Tsur, S. Dynamic itemset counting and implication rules for market basket data. Proc ACM SIGMOD Conference, 255-264, 1997.

[Fayyad et al. 1996] Fayyad, U., Piatetsky-Shapiro, G. and Smythe, P. Knowledge Discovery and Data Mining: Towards a Unifying Framework. Proceedings of the Second International Conference on Data Mining and Knowledge Discovery, AAAI Press, 82-95, 1996.

[Goulbourne et al. 2000] Goulbourne, G. Coenen, F. Leng, P. Algorithms for Computing Association Rules using a Partial-Support Tree. J. Knowledge-Based Systems, 13, 141-149, 2000

[Han et al. 2000] Han, J., Pei, J. and Yin, Y. Mining Frequent Patterns without Candidate Generation. Proc ACM SIGMOD 2000 Conference, 1-12, 2000.

[Houtsma and Swami 1993] Houtsma, M. and Swami, A. Set-oriented mining of association rules. Research Report RJ 9567, IBM Almaden Research Centre, San Jose, October 1993.

[Mannila et al. 1994] Mannila, H., Toivonen, H. and Verkamo, A. I. Efficient algorithms for discovering association rules. Proc. AAAI Workshop on Knowledge Discovery in Databases, Eds. U. M. Fayyad and R. Uthurusamy, Seattle, 181-192, 1994.

[Rymon 1992] Rymon, R. Search Through Systematic Set Enumeration. Proc. 3rd Int'l Conf. on Principles of Knowledge Represenation and Reasoning. 539-550, 1992.

[Savasere et al. 1995] Savasere, A., Omiecinski, E. and Navathe, S. An efficient algorithm for mining association rules in large databases. Proc 21st VLDB Conference, Zurich, 432-444. 1995.

[Toivonen 1996] Toivonen, H. Sampling large databases for association rules. Proc 22nd VLDB Conference, Bombay, 134-145, 1996.

[Zaki et al. 1997] Zaki, M.J., Parthasarathy, S. Ogihara, M. and Li, W. New Algorithms for fast discovery of association rules. Technical report 651, University of Rochester, Computer Science Department, New York. July 1997.