ALGORITHMS FOR COMPUTING ASSOCIATION RULES USING A PARTIAL-SUPPORT TREE

Graham Goulbourne, Frans Coenen and Paul Leng
Department of Computer Science,
University of Liverpool, UK
graham_g, frans, phl@csc.liv.ac.uk

Abstract

This paper presents new algorithms for the extraction of association rules from binary databases. Most existing methods operate by generating many "candidate" sets, representing combinations of attributes which may be associated, and then testing the database to establish the degree of association. This may involve multiple database passes, and is also likely to encounter problems when dealing with "dense" data due to the increase in the number of sets under consideration. Our methods uses a single pass of the database to perform a partial computation of support for all sets encountered in the database, storing this in the form of a set enumeration tree. We describe algorithms for generating this tree and for using it to generate association rules.

KEYWORDS: Association Rules, Partial Support, Set Enumeration

1 INTRODUCTION

Modern businesses have the capacity to store huge amounts of data regarding all aspects of their operations. Deriving *association rules* [1] from this data can lead to previously unknown and interesting information being made available, which may provide the basis for some commercial "edge" to be gained over competitors. This paper describes a new method for the extraction of such rules from binary data. An association rule R is of the form $A \rightarrow B$, where A and B are disjoint subsets of a set I of binary attributes. 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 basis for the inference expressed 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.

In this paper we present a method which, in a single database pass, reorganises an $m \times n$ database table into a simple tree structure and performs a partial computation of support-counts for sets of attributes found in the table. We describe an algorithm for generating this tree, using a conservative storage regime which may require significantly less space than the original database, and we present experimental results to demonstrate its performance. We also describe an algorithm which computes final support totals from the partial supports stored in the tree, and a mode of operation whereby all interesting sets can be generated given a partial support tree and using the summation algorithm described.

The paper is organised as follows. Section 2 provides a review of work within the field of association rule mining. Section 3 describes the theory of the partial support notion and explains its advantages. Section 4 describes our algorithm for generating a partial-support tree, and presents experimental results for the performance of the method. Section 5 presents an algorithm for computing final support-count totals using this tree, and Section 6 describes the generation of interesting sets from this data. Finally Section 7 outlines some directions for future research which we think will improve still further the algorithms outlined in this paper and some concluding remarks.

2 ALGORITHMS FOR COMPUTING SUPPORT

For a database of n attributes, to compute support counts for all 2^n subsets of I is likely to be infeasible, so practicable algorithms proceed in general by computing support-counts only for those sets which are identified as potentially interesting. To assist in identifying these candidate sets, observe that the subsets of the attribute set may be represented as a lattice. One form of this is shown as Figure 1, for a set of four attributes, $\{A, B, C, D\}$.

The best-known algorithm, "Apriori" [2], proceeds essentially by breadth-first traversal of this lattice, starting with the single attributes in I. Repeated passes of the database are performed, 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. The kth cycle of the algorithm then first performs a pass over the database to compute the support for all members of C_k, and from this, produces the set L_k of interesting sets of size k. This is then used to derive the candidate sets C_{k+1}, using the "downward closure" property, that all the subsets of any member of C_{k+1} must be members of L_k. Other algorithms, AIS [1] and SETM [3], have the same general form but differ in the way the candidate sets are derived.

Two aspects of the performance of these algorithms are of concern: 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, and the size of the candidate sets which may be generated, especially in the early cycles of the algorithm. The number of database passes can be reduced by strategies which begin by examining subsets of the database. The Partition algorithm [6] 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, for which the support exceeds the required (proportional) threshold within at least one partition, are computed. The set L of locally frequent sets is a superset of the set of all (globally) frequent sets, and 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 [7]. 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

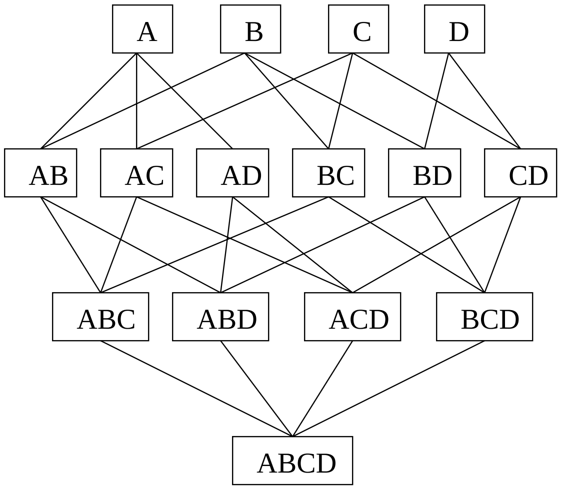

Figure 1: Lattice of subsets of $\{A, B, C, D\}$

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*, i.e. those sets that are not members of S, but all of whose subsets are included in S. This extended set is used as the candidate set for a full database pass, which, if no members of the negative border are found to be frequent, completes the search.

The drawback of these methods is that the candidate set derived is necessarily a superset of the actual set of frequent sets, so again the search space may become very large, especially with densely packed database records. This has led researchers to look for methods which seek to identify *maximal* interesting sets without first examining all their smaller subsets, i.e. a depth-first search of the lattice of Figure 1. An exhaustive depth-first traversal is, of course, computationally infeasible for large n. Zaki et al [4] address this issue by partitioning the lattice, using *clusters* of associated attributes. Each cluster generates a subset of the lattice, for which a number of traversal methods are described. However, the approach depends critically on the ability to partition the attributes into relatively small clusters. Also, the clustering algorithms generally begin by computing L_2, which in itself involves a pass of the database and significant computation.

Bayardo's [5] Max-Miner algorithm also searches for maximal sets, using Rymon's set enumeration framework [8] to order the search space as a tree. Max-Miner reduces the search space by pruning the tree to eliminate both supersets of infrequent sets and subsets of frequent sets. Max-Miner copes better with dense datasets than the other algorithms described, but at the cost of multiple database passes.

3 PARTIAL SUPPORT

The algorithms described above all depend to some degree on a reduction of the search space of candidates for support to a number which can be retained in memory and counted feasibly. In most cases, each pass of the database proceeds by examining each record to identify all the members of the candidate set that are subsets of the record. This can be computationally expensive, especially when records are densely populated, i.e. when the average number of attributes present in a record is high. In principle, however, it is possible to reduce this cost by exploiting the relationships between sets of attributes illustrated in the lattice. 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 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 an exhaustive algorithm for computing all (total) supports:

```
Algorithm A1:
      Stage 1: for all records j in database do
               begin add 1 to P(j)
               end;
      Stage 2: for all distinct j found in database do
               begin for all i ⊂ j do
                     begin add P(j) to T(i)
                     end
               end
```

For a database of m records, the algorithm performs m support-count incrementations in a single pass (stage 1), to compute a total of m' partial supports, for some $m' \leq m$. Stage 2 involves, for each of these, a further 2^i additions, where i is the number of attributes present in the set being considered. If the database contains no duplicate records, then the method will be less efficient than an exhaustive computation which enumerates subsets of each record as it is examined. Computing via summation of partial supports will be superior, however, in two cases. Firstly, when n is small ($2^n \ll m$), then stage 2 involves the summation of a set of counts which is significantly smaller than a summation over the whole database, especially if the database records are densely-populated. Secondly, even for large n, if the database contains a high degree of duplication ($m' \ll m$) then the stage 2 summation will be significantly faster than a full database pass.

Computing partial supports as described above allows us, in one pass of the database, to capture all the relevant information in a form which enables efficient computation of the totals we need, exploiting the structural relationships inherent in the lattice of partial supports, and taking advantage of the duplication of records which we expect to be a significant feature of many databases. For effective implementation of the method, however, we require a method of storing the partial supports in a form which facilitates the stage 2 computation. For this purpose, we make use of Rymon's [8] set enumeration framework. Figure 2 shows the sets of subsets of I, for $I = \{A, B, C, D\}$, in this form. In this structure, each subtree contains all the supersets of the root node which follow the root node in lexicographic order.

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:

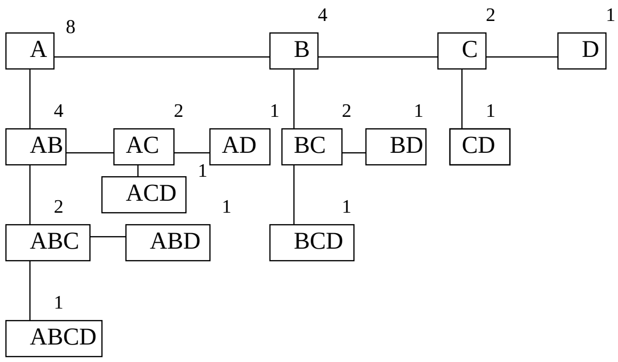

Figure 2: Tree storage of subsets of $\{A, B, C, D\}$

Algorithm A2:

```
    for all records j in database do
        starting at root node of tree:
            begin if j ⊇ node then increment Q(node);
                if j = node then exit
                  else if j ⊇ node then recurse to child node
                  else recurse to next sibling;
            end
```

This algorithm will, in stage 1, compute interim support-counts Q_i for each subset i of I, where $Q(i)$ is defined thus:

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

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

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

Storage of the complete tree of subsets of I, of course, has a requirement of order 2^n, which will in general be infeasible. 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 algorithm A2 to exploit this builds the tree dynamically as records are processed, storing partial 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. Building the tree dynamically 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. In the following sections we describe implementations of algorithms for building the partial support tree in this way, and then for completing the support-count computation using the tree.

4 ALGORITHM FOR GENERATING PARTIAL SUPPORT TREE

In this section we present an algorithm to produce a partial support tree with dummy nodes. The nodes in the tree represent either unique rows in the table, or dummy nodes inserted to prevent

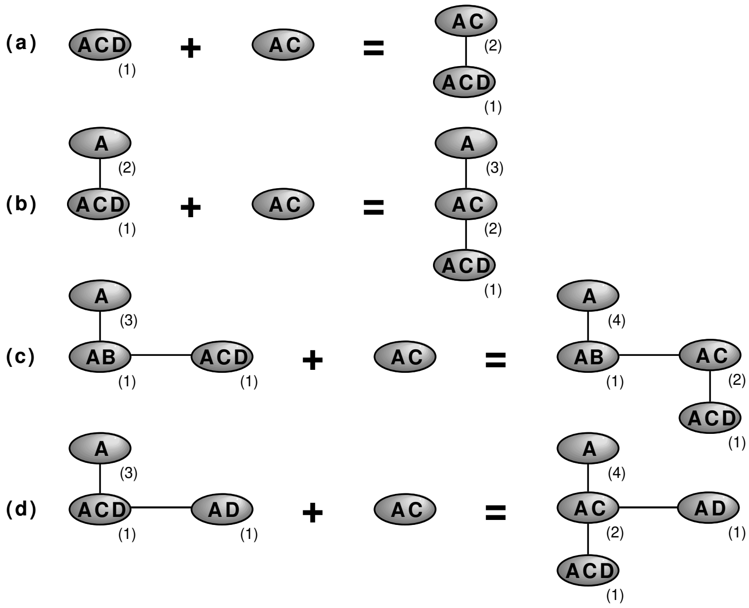

Figure 3: Application of Rule 2

the tree from degenerating into a "linked list". The nodes are arranged so that parent nodes are subsets of their descendent nodes. Sibling nodes are ordered lexicographically, consequently we conceive of "elder" and "younger" siblings. Examples are given later in the text. The algorithm operates by passing through the database row by row in an iterative manner. On each iteration if the row under consideration is already represented by a node in the tree the support associated with this node is incremented by one. Otherwise a new node is created for the row and inserted in the appropriate location in the tree. This requires a search through the tree as a consequence of which the new node is inserted as a parent, child, elder sibling or younger sibling of some existing node. On route supports associated with existing nodes may be incremented. Where necessary an additional (dummy) node may also be created to preserve the desired tree structure. Alternatively, when inserting a new node it may be necessary to break certain portions of the tree and reconnect them so as to conform to the defined structure of the tree. To achieve this the tree is traversed in an iterative manner also. On each iteration the nature of the search is defined by a set of five basic rules. These rules are described in detail below (where R represents the bit pattern of the table row currently under consideration, and B a bit pattern attached to an existing tree node under consideration). The search commences at the root of the tree and may progress down either the sibling branch or child branch of this node. So that knowledge of the nature of the current branch is maintained a flag is set with the values 0 (root), 1 (child) or 2 (sibling). Note; the $<$ and $>$ operators are used to define lexicographic not numeric ordering.

Rule 1 ($R = B$, i.e. an identical node is found) : Simply increment the support associated with the current node and return.

Rule 2 ($R < B$ and $R \subset B$, i.e. new row is a parent of current node) :

1. Create a new node for R and place the existing node associated with B on the new node's child branch.
2. Place the new node either as a new root node, or add it to the child or sibling branch of the previously investigated node (as indicated by the flag). This is illustrated in

Figure 3(a), (b) and (c). In Figure 3 (a) the new node is added as a new root node, in (b) as a child of the previously investigated node and in (c) as a elder sibling.

3. If necessary move one or more of the younger siblings of the previously existent node up to become younger siblings of the newly created node. This is illustrated in Figure 3(d) where the node AD, which was an elder sibling of ACD is moved up to become an elder sibling of the newly created node AC.

Rule 3 ($R < B$ and $R \not\subset B$, i.e. new row is an elder sibling of existing node) :

1. If R and B have a leading sub string S and S is not equal to the code associated with the parent node of the current node representing B then:
   - Create a dummy node for S with support equivalent to that associated with the node for B.
   - Place the new dummy node either as a new root node, or add it to the child or sibling branch of the previously investigated node (as indicated by the flag). A number of examples illustrating this are given in Figure 4. In Figure 4(a) we create a dummy node (B) as a new root node, in 4(b) we create a dummy node (BC) as child node of node B, and in Figure 4(c) we create a dummy node (BC) as an elder sibling node of AB.
   - Then create a new node for R and place this so that it is a child of the newly created dummy node.
   - Finally place the previously existent node for B as a younger sibling of the node for R.
2. Else create a new node for R and place the existing node associated with B on the new node's sibling branch. The new node is then placed either as a new root node, or is added to the child or sibling branch of the previously investigated node (as indicated by the flag). Examples are given in Figure 4 (d), (e) and (f).

Rule 4 ($R > B$ and $R \subset B$, i.e. new row child of current node) : Increment the support for the current node (B) by one and:

1. If node associated with B is has no child node create a new node for R and add this to the existing node's child branch.
2. Otherwise proceed down child branch (with flag set appropriately) and apply the rules to the next node encountered.

Rule 5 ($R > B$ and $R \not\subset B$, i.e. new row is a younger sibling of current node) :

1. If node associated with B has no sibling node, and:
   - If current node (node for B) does not have a parent node and R and B have a leading sub string S then:
     - Create a dummy node for S with support equivalent to that associated with the node for B.
     - Place the new dummy node either as a new root node, or add it to the child or sibling branch of the previously investigated node (as indicated by the appropriately set flag). This is illustrated in Figure 5(a) where the dummy node C is created.
     - Create a new node for R and place this so that it is a younger sibling of the newly created dummy node.
     - Place the previously existent node for B as a child of the node for R.

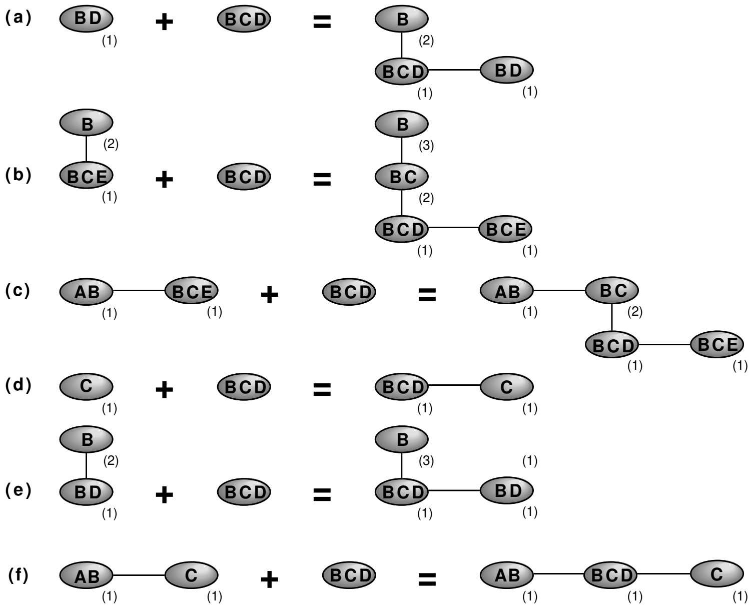

Figure 4: Application of Rule 3

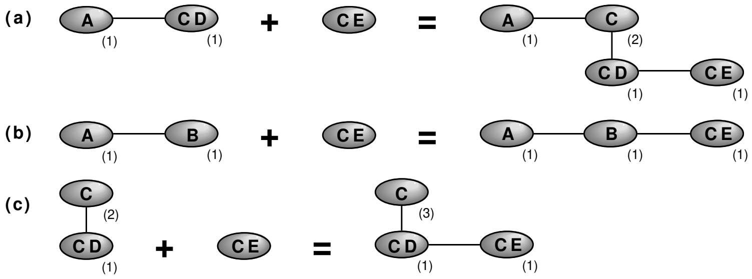

Figure 5: Application of Rule 5

- Otherwise create a new node for R and add this to the existing node's sibling branch (Figure 5(b) and (c)).

2. Otherwise proceed down sibling branch and (with flag set appropriately) and apply the rules to the next node encountered.

Thus given a bit Pattern R and a bit pattern B associated with an existing node there are five broad possible outcomes; each with a number of variations according to the nature of the tree so far, and the relationship between R and B.

4.1 ANALYSIS

To determine the efficiency/effectiveness of the algorithm it was tested with a series of randomly generated tables each with 20,000 rows ($m = 20,000$) and varying numbers of columns from $n = 10$ to $n = 100$ with each element having 0.5 probability of having a value of 1. For each test the amount of storage required by the resulting tree (in bytes) and the number of *comparison tests* required to generate the tree was determined and recorded. Similar tests were run using a *benchmark* algorithm ([9]) where the partial supports were stored in a linked list structure; and using a

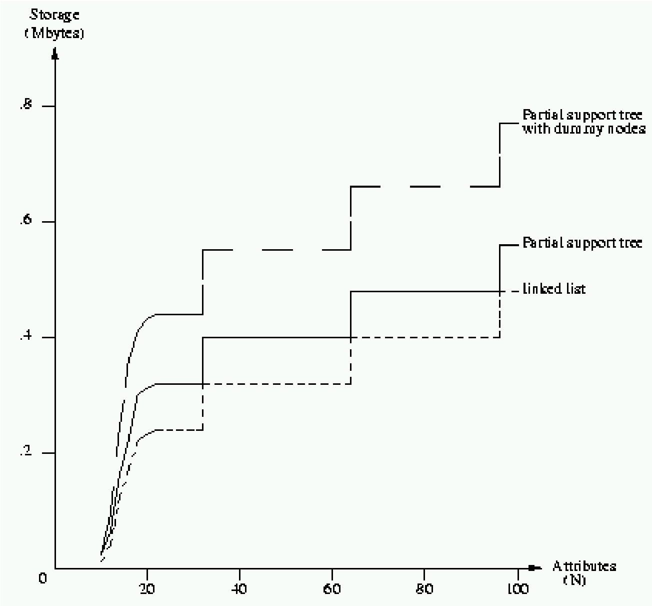

Figure 6: Comparison of storage requirements for linked list and partial support tree algorithms ($M = 20000$)

precursor of the current algorithm ([10]) which did not incorporate the dummy node concept.

A graph indicating the storage requirements for the three algorithms derived from the test data is given in Figure 6. It should be noted that substrings of 32 table attributes are stored as 32 bit integers. An entire attribute set is thus stored in the form of a K element array where K is calculated as follows:

$$K = \frac{n}{32}(rounded\ up\ to\ the\ nearest\ integer)$$

As a consequence of the above the "staircase" effect which is a feature of the graph presented in Figure 6 is produced where a new "step" occurs at intervals of 32. From the graph it can be seen that the tree algorithm with no dummy nodes requires more storage than the linked list algorithm. While the algorithm with dummy nodes requires more than that without dummy nodes. This is not surprising because each linked list node has one branch, while each tree node has two branches. From the graph in Figure 6 it can also be seen that, in each case, the slope levels off where N is equivalent to about 17. This change occurs because at this point 2^N overtakes M, i.e. to the left of $N = 17$ duplicates are guaranteed while to the right this is not the case. Inspection of the test data shows that for $N > 22$ the number of duplicates is such that storage can be calculated using:

$$M \times (K + S)$$

where S represents the additional storage requirements to store the support associated with an individual node plus "housekeeping" requirements. Thus when 2^N is greater than M (which in practice is likely to be the case) storage will be a function of M rather than N.

A second graph is presented in Figure 7 which compares the number of "tests" associated with each of the algorithms. Considering only the linked list and (non-dummy node) partial support tree algorithms first, from the graph, it can be seem that the partial support tree works better than the linked list algorithm to the left of $N = 15$ (i.e. where the likelihood of subsets and duplicates is high), but as N increases beyond 15 this advantage is lost. For both the linked

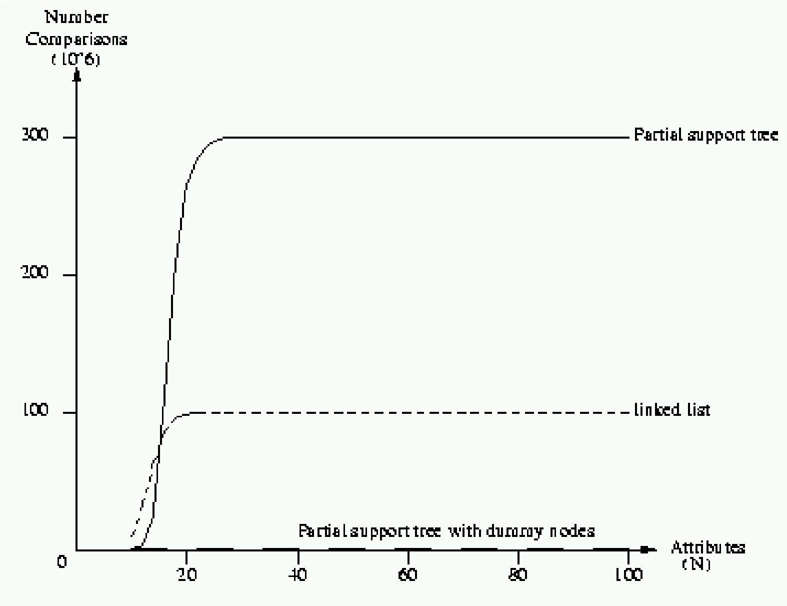

Figure 7: Comparison of number of "tests" (comparisons) for linked list and partial support tree algorithms ($M = 20000$)

list and partial support graphs the gradient of the slope again decreases at $N = 17$, the point beyond which duplicates are no longer guaranteed. Consequently, in both cases the number of comparisons becomes a function of M rather than N. In the case of the partial support tree the tree in fact becomes "flat", i.e. it degenerates into what is effectively a linked list due to the lack of subsets in the test data. However, for each row, checks are still made for the possibility of adding child branches, corresponding to only one check in the linked list algorithm. This makes the tree algorithm extremely "expensive". If we now consider the partial support tree with dummy nodes this does not degenerate into a "linked list", but maintains a genuine tree structure, and the amount of work to generate the tree is significantly less. For example where $N = 20$, the number of comparisons required to produce the linked list structure is approximately 98×10^6, for the non-dummy node tree 264×10^6 and with dummy nodes 0.8×10^6. The efficiency gains can thus be said to be "significant". There is still a gradient change at $N = 17$, however, the scale of the graph in Figure 7 is such that this is not perceptible.

The overall advantages offered by the tree algorithm is, of course, also dependent on the "added value" offered by the tree organisation when determining supports — an algorithm whereby this may be achieved is described in the following section.

5 SUMMATION OF PARTIAL SUPPORTS

The tree structure now contains an image of the database together with the partially supported sets. To enable rule generation the total support for particular sets must be determined.

The proposed summation algorithm operates in a depth first manner by proceeding down child branches first and then exploring elder sibling branches. When considering a child node we proceed according to the rules presented below where: R is a set of attributes for which we wish to determine the support, B is a node in the tree (which will be the root node at commencement of the calculation), and N is the support calculated so far (0 at the start). The $flag$ variable indicates whether an exact match ($flag = 0$) has been found or not ($flag = 1$). Where an exact match has been found the search is terminated. The search is also terminated when all possibilities have been exhausted (despite the absence of an exact match).

Child summation rules:

Rule 1(a): If there is no node B set $flag$ to 1 and return (and possibly continue the search down a sibling branch).

Rule 1(b): If the code associated with B is identical to R, add the support for B to N, set $flag$ to 0 and return (terminate search).

Rule 1(c): If the code associated with B is a superset of R, add the support for B to N, set $flag$ to 1 and return (to possibly continue the search down a sibling branch).

Rule 1(d): Proceed down child branch emanating from B and reapply these rules. If as a result of this exploration $flag$ is set to 0 (i.e. an identical match has been found) return. Otherwise go on to explore the sibling branch emanating from B (see below).

Investigation of a sibling branch proceeds in a similar manner:

Sibling summation rules:

Rule 2(a): If there is no sibling node B set $flag$ to 1 and return (to possibly continue the search down another sibling branch).

Rule 2(b): If the code associated with B is identical to R, add the support for B to N, set $flag$ to 0 and return (thus terminating the search).

Rule 2(c): If the code associated with B is a superset of R, add the support for B to N, set $flag$ to 1 and return (to possibly continue the search elsewhere).

Rule 2(d): If the code associated with B is (lexicographically) after R no occurrences of R will exist in any of the child/sibling branches emanating from S, set $flag$ to 1 and return (and possibly continue the search down another sibling branch).

Rule 2(e): Proceed down child branch emanating from current sibling node B and apply the rules presented above for investigating child nodes. If as a result of this exploration $flag$ is set to 0 simply return (i.e. an identical match has been found therefore terminate the search). Otherwise go on to explore the immediate sibling emanating from B by reapplying these rules.

For example considering the tree presented in Figure 2 and if $R = \{AD\}$, the search would be restricted to the A sub-tree, and would proceed as follows. We commence with the root node A and apply the child summation rules. Repeated application of rule 1(d) will bring us to node $ABCD$ as a result of which $N = 1$ and $flag = 1$ (rule 1(c)). We will now back track to ABC and (rule 1(d)) proceed down the sibling branch. Using the sibling summation rules, rule 2(c) will apply as a result of which $N = N + 1 = 2$ and $flag = 1$. Backtracking again will return us to AB and we will then (rule 1(d)) then continue down the sibling branch of this node to AC. Application of rule 2(e) will then lead us down the child branch emanating from AC to ACD and (as a result of rule 1(c)) branch emanating from AC to ACD and (as a result of rule 1(c)) $N = N + 1 = 3$ and $flag = 1$. We will then again backtrack, to AC, and proceed down the sibling branch of this node to investigate AD using the sibling summation rules. Rule 2(b) will be applicable, $N = N + 1 = 4$ and $flag = 0$ hence the search will be terminated

6 INTERESTING SET GENERATION

To determine all the interesting sets in a database presented in the form of a tree generated using the algorithm presented in Section 4 we proceed in an "apriori" manner, using the summation algorithm presented in Section 5, as follows:

1. Generate the set K_1 of all elements with adequate support. If K_1 is empty return (there are no interesting sets), otherwise continue.
2. From the set K_1 generate a candidate set C_2 of two element sets.
3. Prune the set C_2, using the summation algorithm, to produce the set K_2 — the sets of two element sets which are appropriately supported. If K_2 is empty return (there are no interesting sets with more than one element), otherwise continue.
4. From the set K_2 generate a candidate set C_3 of three element sets.
5. Prune the set C_3, using the summation algorithm, to produce the set K_3 — the sets of two element sets which are appropriately supported.
6. Continue in this manner until the set K_n has been produced (n is the number of columns in the database) or a particular set K is empty.

In this manner all interesting sets will be discovered, and consequently appropriate association rules can be generated.

7 CONCLUSION AND FUTURE WORK

In this paper we have presented an algorithm for using the structure of sets to produce a partial support tree. From this tree it is possible to generate support for interesting sets in a database table. The work presented addresses two of the fundamental issues within association rule mining;

- The reduction of the number of database passes required to generate rules, the above can be catagorised as 1+ pass. In one full pass of the database, computation of the partial support for the attributes is completed, at the same time the databse is reorganised into a structure that can be efficiently summed.
- The ability to cope with dense data without encountering an exponential increase in the size of the search space.

As can be seen from the results presented in Figures 6 and 7 generating the partial support tree with dummy nodes is extremely efficient. The higher the level of duplication within the database the more the algorithm improves.

Future work will concentrate on both the data structure and the summation algorithm. The possibility of partitioning the data structure into distinct trees, groups of trees or trees mixed with other structures is to be examined. This restructuring may lead to the possibility of various "lazy" techniques for the summation algorithm. This would reduce even further the "restricted" exhaustive search outlined above.

References

[1] Agrawal, R. Imielinski, T. Swami, A. Mining Association Rules Between Sets of Items in Larg e Databases. SIGMOD-93, 207-216. May 1993.

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

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

[4] 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.

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

[6] 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.

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

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

[9] Coenen, F. (1999). Partial Support. Dept of Computer Science, University of Liverpool. Working Paper 4. `http://www.csc.liv.ac.uk/~graham_g/kdfm.html`.

[10] Coenen, F. (1999). Partial Support Using a "Partial Support Tree". Dept of Computer Science, University of Liverpool. Working Paper 5. `http://www.csc.liv.ac.uk/~graham_g/kdfm.html`.