# Policy enforcement in releasing information for large databases

*Adrian Perrig*
Dept. of Computer Science
Carnegie Mellon University
adrian@cs.cmu.edu

*Qifa Ke*
Dept. of Computer Science
Carnegie Mellon University
ke+@cs.cmu.edu

*Dawn Xiaodong Song*
Dept. of Computer Science
Carnegie Mellon University
skyxd@cs.cmu.edu

March 2, 1999

**Abstract**

Many databases such as medical databases contain sensitive information. We need to enforce access policies upon releasing the databases. In this report we propose a way to automatically find the generalized association rules from the database and automatically generalize the database so that the newly generated database will enforce the required access policies. For mining the generalized association rules, we developed an improved algorithm based on IBM's Cumulate algorithms (frequency counting). We also propose a fast greedy algorithm to generalize the database with minimum information loss. We developed our own data generation engine. We tested our algorithms on the datasets generated by this engine. Our test dataset is as big as 1,000,000 tuples. The result shows that our method out-preforms the exising method. In many cases, our algorithm is twice as fast as the Cumulate algorithm.

## 1 Introduction

In the area of medical information, 37 states maintain state-level data databases that include a tuple for each medical encounter in the state for the last 7-10 years. That is, a copy of each and every hospital visit is recorded in these databases. Obviously, these are very large databases containing very sensitive information. Obviously, these databases are of great value if we could release them for research or other statistical queries or other legal use. But at the same time, we need to guarantee that the release of information will not reveal any sensitive information.

We can give a better example here. Say, some researchers want a release from a medical database. Almost every state has a written policy (or related legislation) that prevents them from revealing the identities of HIV patients, stated as: RELEASE TO ”researcher” PROHIBIT diagnosis WHEN diagnosis=”HIV”. However, enforcement of this policy must distinguish the explicit statement from the overall intent. Many medications are specific to HIV and so releasing these medications allows one to infer the HIV diagnosis; likewise, HIV patients are often located in the same area of the hospital, so in these cases, room number is also an indicator of HIV status. So what is actually meant by the original stated policy is not to allow one to infer that the diagnosis is HIV. The simple policy stated above in number 1 can be expanded to imply many similar policies such as these: 2. RELEASE TO ”researcher” PROHIBIT medication WHEN diagnosis=”HIV”; 3. RELEASE TO ”researcher” PROHIBIT room WHEN diagnosis=”HIV”.

It’s hard to believe that it is practically feasible in this setting to know a priori all attributes whose values may be related to the inference which the policy seeks to control. After all, the typical medical database has over 350 attributes. Instead, we need some data-mining and classification method to find the correlation among attributes that can infer the diagnosis which is “HIV”. Then we need some suppression or generalization or adding noise to generate a new database. This new database can be released to the recipient. He can do any query on the released database. Our goal is to enforce the policy which means we want to guarantee that no matter what queries the recipient do he can’t get any information that invalidates the policy.

There are many other potential applications as well. For simplicity reason, we stay with the ”HIV” example in this report. Also we assume the readers are familiar with the basic concept of association rules, support, confidence, taxonomy and generalized association rules. When we mention about *association rules* in this report, we always mean *generalized association rules*. We use both *item* and *literal* to refer to the basic component of the association rules. They refer to the leave nodes as well as the higher nodes in the taxonomy. The ancestor items of an association rule is the item set of the left-hand side of the association rule. The size of an association rule is the number of ancestor items of the association rule.

This report is organized as the following: in section 2 we give the problem definition; in section 3.2 we give a detailed description of our approach and prove the correctness of the algorithm; in section 4 we describe our data generation engine, which generates database according to some data model; in section 5 we describe our data structure and explain why it’s a good data structure; in section 6 we give the experimental results runs on different size of data set generated by our data generation engine; then finally in the discussion section, we compare with other possible methods to deal with the problem of mining association rules, and we also propose some possible further improvement of the algorithms.

## 2 Problem Statement

**Assumption of the database** The database we consider can have categorical attributes with taxonomies over them. In reality, many databases have natual taxonomies, also as pointed out in [SA95]. For simplicity reason, in this report and in the implementation we assume all the attributes are binary with a taxonomy over them. Since it’s easy to convert a database with categorical attributes to a database with binary attributes, this assumption will not give us any limitation. We also assume that all our interesting association rules have HIV as their consequence because we assume here the policy is simply not to reveal HIV. This won't lose any generality either because it's trivial to change to other more complicated consequences of the association rules.

**Generalizing a Database** Database *DB* has a set of attributes $\{A_i\}$ with a taxonomy *T* over the attributes. *T* can be modeled as a *DAG* which defines a pre-order $\preceq$ on the DAG. $T_i \preceq T_j$ *iff.* $T_j$ is an ancestor of $T_i$ in the DAG. For each attribute $A_i$, we define a generalization rule $A_i \mapsto P_i$, where $A_i \preceq P_i$ in $T$ and $\mapsto$ means *generalize-to*. A tuple $\{x_i\} \mapsto \{y_i\}$ *iff.* for every *i*, $x_i \mapsto y_i$ according to the generalization rules. Hence, given a set of generalization rules, the database *DB* $\mapsto$ *PDB* *iff.* each tuple in *DB* $\mapsto$ the correspondent tuple in *PDB*.

**Information Loss** Given a database *DB* with a set of attributes $\{A_i\}$ and a set of generalization rules $\{A_i \preceq P_i\}$, *DB* $\mapsto$ *PDB*. The information loss of *PDB* from *DB* is $\sum_i (R(A_i) - R(P_i))$, where *R* is the rank function of the taxonomy with the R(root) = 0. This formula can also easily be extended to a weighted sum.

**Valid generalization rule set** Given a database *DB*, *S* is the complete set of association rules with the form $A \Rightarrow HIV$ with given threshold of support and confidence. A generalization rule set $G$ is *valid iff.* in the generalized database any association rule in *S* doesn't hold any more. It's easy to prove that this also implies that in the generalized database there will be no association rule that will imply HIV. Hence the generalized database will not reveal HIV.

**Problem Statement** Given a database , our goal is to find a valid generalization rule set so that the generalized database obeys the policy and has minimum information loss.

# 3 Our approach and algorithm

## 3.1 Our approach

Given the goal, our approach has two main steps. First, we need to find the association rules which imply HIV. Second, we use these association rules to generate the valid generalization rule set with minimize the information loss.

There has been a lot of work done in mining association rules. One of the most popular and state-of-the-art algorithm is the "Frequent-set" and its variations and extensions [ATA93] [AS94] [BMUT97] [AM98] [SA95] [SVA97]. Two of them [SA95][SVA97] are most related to our problem. [SA95] introduces several algorithms extended from "Frequent-set" mining generalized association rules. [SVA97] can find association rules satisfying given constraints. But our problem has some special properties that we can take advantage of and develop faster algorithms. First, we only need to look for association rules with given consequence, in our example it is "HIV". Second, one interesting observation is that if $A \Rightarrow HIV$ and $A \wedge B \Rightarrow HIV$, and in the generalized database, $A \mapsto P$, so that $A \Rightarrow HIV$ doesn't hold any more that $A \wedge B \Rightarrow HIV$ won't hold any more either. This shows that once we find an association rule $A \Rightarrow HIV$, we no longer need to search association rules like $A \wedge B \Rightarrow HIV$. We formalize this as the following theorem.

**Simplified Association Rule Set** S is a set of association rules with the form $\{A_i \Rightarrow HIV\}$, where $A_i$ is a literal or item set. If $A_i$ is a subset of $A_j$, we say $A_j$ *includes* $A_i$. We eliminate such $A_i$ which is included by another association rule $A_j$ in S. Thus we can get a subset $S_1$ of S. $S_1$ is a simplified association rule set of S. The *maximum simplified association rule set* $S_m$ is the maximum subset of S in which there is no association rule included by another one. We can easily prove that for any S, such a $S_m$ exists and is unique.

**Theorem 1.** $S_m$ is the most simplified association rule set of $S$. If $R$ is a valid generalized rule set that makes $S_m$ doesn't hold any more, then $R$ will make $S$ not to hold any more either. If interested in the proof detail, please contact for proof appendix.

## 3.2 Mining the most simplified association rule set

### 3.2.1 the IBM's algorithm

Agrawal[ATA93, AS94] et. al. in IBM proposed two algorithms to mine the generalized association rules, given the database and the taxonomy on the attributes. One is the *basic* algorithm, the other one is the improved version called *Cumulate*. Since our medical database is very different from IBM's basket database, we design our own data structure and improve the Cumulate algorithm to fit our purpose of mining the most simplified generalization rules in the medical database.

We first describe the IBM algorithms, then we will show our algorithm and improvements.

**Algorithm Basic** The Basic algorithm goes through the whole database several passes until it can't generate new candidates. In *pass k*, it counts the support for each candidates with size $k$ (contains $k$ items). So for the first pass, it simply counts the frequent 1-itemsets. Then in the following passes, say pass $k$, it first generates a candidate set with candidate size $k+1$ using the candidate set got from last pass, and then go through the database to get the support for a each candidate in it. In order to get the generalized rules, the nodes in the itemsets can come from the leaves or the interior nodes of the taxonomy tree. To check if a tuple supports a candidate $C$, it first adds all the ancestors of each item in $C$, and then check if the extended tuple supports $C$ in the candidate set. The pseudo code is as the following:

```
L1 := frequent 1-itemsets;
k := 2; // k represents the pass number
while (L_{k-1} ≠ 0) do
begin
    C_k := New candidates of size k generated from L_{k-1}.
    forall transactions t ∈ D do
    begin
        Add all ancestors of each item in t to t, removing any duplicates.
        Increment the count of all candidates in C_k that are contained in t.
    end
    L_k := All candidates in C_k with minimum support.
    k := k + 1;
```

end

Answer := $\bigcup_k L_k$;

**Algorithm Cumulate** The Cumulate algorithm is an extension of the Basic algorithm. It mainly has three improvements:

- Instead of adding all ancestors of the items in a transaction t to t, now we only add the ancestors that are in the candidate set to transactions.
- Pre-compute the ancestors for each item.
- Pruning itemsets containing an item and its ancestor.

The pseudo code of the this algorithm is as follows:

.

Compute T′, the set of ancestors of each item, from T. //Optimization 2
$L_1$ := frequent 1-itemsets;
k := 2; // k represents the pass number
while ($L_{k-1} \neq 0$) do
begin
    $C_k$ := New candidates of size k generated from $L_{k-1}$.
    if (k=2) then
        Delete any candidate in $C_2$ that consists of an item and its ancestor. //Optimization 3
        Delete any ancestors in T′ that are not present in any of the candidates in $C_k$. //Optimization 1
    forall transactions t ∈ D do
    begin
        for each item x ∈ t do
            Add all ancestors of x in T′ to t.
            Remove any duplicates from t.
            Increment the count of all candidates in $C_k$ that are contained in t.
    end

Answer := $\bigcup_k L_k$;

**Our improved algorithm:**

```
LH := empty;
L1 := {frequent 1-itemsets};

Extract all of the very high confidence and high support candidates
from L1 and add it to LH.

k  := 2;
while( L(k-1) is not empty ), do
```

```
begin
  C(k) := New candidates of size k generated from L(k-1).
  if( k == 2 ) then
    Delete any candidate in C(k) that consists of an item and its
    ancestor.
  for each tuple t in database, do
    Increment the count of all candidates in C(k) that are contained
    in t
  end
  L(k) := ALl candidates in C(k) with minimum support.
  Extract all of the very high confidence and high support candidates
  from L(k) and add it to LH.
  k    := k + 1;
end

Output all of the non-empty L(k), and LH.
```

One of the differences between the above algorithm and the IBM Cumulate algorithm is that we don't need to extend the tuples in the database by adding to the tuple the ancestors of each item in that tuple. This will save the space as well as the time to match the tuple against the candidate. This benefit comes from our special data structure. That is, we store a template for each node in the taxonomy tree. Any child of this node will have one or several bits set inside the template. So this node will be supported as long as any one of its children appears in the tuple. Since a candidate is simply a list of nodes in the taxonomy trees, with the templates we already take into account the ancestors of each item in the tuple without extending.

The other difference is that if there is a candidate $c$ in $L(k)$ has very high support and confidence, we extract it from $L(k)$ and output it as the association rules. Therefore, in the following pass, we will not have candidates contain $c$ as a subset. The reason we have this improvement is mentioned before.

### 3.3 Database generalization

**Immediate generalization** Given an item $A$ and its immediate parent $B$ in the taxonomy, the immediate generalization rule of $A$ is $A \mapsto B$.

Given a most simplified association rule set $S$, $W$ is the set of the ancestor items of the rules in $S$. Naively we could do an exhaustive search of the subset of $W$. For any subset of $W$, we generate the immediate generalization and verify its validity and compute the information loss. At the end we can find the valid generalization rules with the least information loss.

But obviously this will be too expensive. We develop a greedy algorithm which is very efficient and in general the information loss can be approximate to the optimal solution.

The algorithm works in several passes.

**Pass One** We compute $W_1$, the set of ancestor items of the size-one association rules in $S$. We add the immediate generalization rules of $W_1$ to the generalization rule set $G_1$. We then use $G_1$ to prune $S$, eliminating the association rules which will not hold any more in the generalized database with $G_1$ generalization rule set. Hence we get $S_1$ which doesn't contain any size-one association rule.

**Pass Two** We compute $W_2$, the set of ancestor items of the size-two association rules in $S_1$. For each item $A_i$ in $W_2$, we compute a heuristic function $H(A_i)$. $H$ function will be explained later. Assume $A_m$ has the highest $H$ value. We add the immediate generalization rule of $A_m$ to the generalization rule set $G_1$. We then use $G_1$ to prune $S_1$, eliminating the association rules which will not hold any more in the generalized database with $G_1$ generalization rule set. Then we repeat this pass again until $W_2$ is empty. Hence we get $S_2$ which doesn't contain any size-one and size-two association rules.

**Pass K** Do similar steps as in Pass two for size-k association rules.

**Finally,** We get $G_1$. If $B$ is $A$'s ancestor in the taxonomy, and both A and B's immediate association rules are in $G_1$, then we delete A's immediate association rule from $G_1$. At the end we get the pruned generalized association rule set $G_f$.

**Heuristic function H** The intuition of the heuristic function H is that we should favor an item if it appears in many association rules in $S$. Also we should favor an item more if it appears more in short size association rules in $S$. One possible heuristic function could be $H(A_i) = \sum$ weight($R_j$) where $A_i \in R_j$, $R_j$ is an association rule in $S$. A possible weight function could be $weight(R_j) = \frac{1}{size(R_j)}$. Other heuristic functions with similar properties are possible as well.

We can see that this algorithm finds valid generalization rule set pretty efficiently. With a good heuristic function, on the average case, the information loss is close to the optimal solution.

## 4 Data Generation

Data generation is very important and useful for developing and testing new database algorithms. (Generated data is usually referred to as synthetic data.) First, it's easier to control the statistical characteristics of synthetic data which can predict the correct result theoretically and hence help testing the new algorithms. Second, it's not always easy to get interesting and easy-to-use realistic database. For the same reason, data generation plays an important role in our project as well, especially since it's very difficult to get a medical database containing real-world sensitive data because we would have to go through some clearance process first.

The data generation in this project contains 3 parts: Taxonomy generation, tuple-set generation and association rule generation.

For the *Taxonomy*[1] generation we have two parameters to control the generation of the hierarchy tree: the maximum depth of the hierarchy tree $D_{max}$, and the maximum node fanout $F_{max}$. The entire hierarchy tree stands for one categorical attribute. Each leaf in the hierarchy tree forms one binary attribute and corresponds to one value of the categorical attribute. Figure 1 describes this

[1] A taxonomy is a "is-a hierarchy"

point, please note that only the leaves have an attribute number. We call the group of binary attributes corresponding to one categorical attribute an *attribute group*. The total number of attribute groups $N_{ag}$ is therefore the number of attributes in our medical database. We chose the values $D_{max} = 4$ and $F_{max} = 4$. Figure 2 shows another tree that was generated. On average one attribute group has around 20 leaves. The total number $N_{tot}$ of binary attributes generated varies. *talk about max size, etc.*

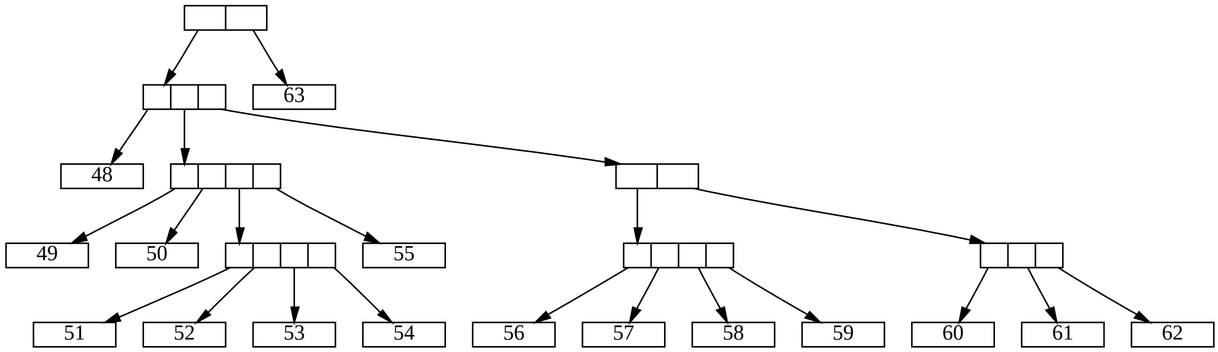


Figure 1: A sample hierarchy tree

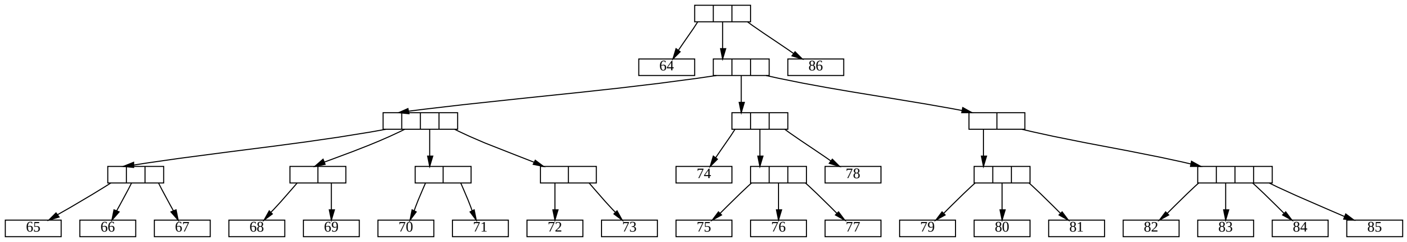


Figure 2: Another sample hierarchy tree

Once we have generated the taxonomy, we can generate the association rules and the tuple sets. The parameters for the association rule are the number of association rules $N_{ar}$ to generate and the maximum number of literals per association rule $L_{max}$. As mentioned earlier, our association rules are of the form $I_1 \wedge I_2 \wedge \ldots \wedge I_n \Rightarrow HIV$ (we designate $n$ as the length of the association rule). Ideally our association rules do not partition the positive tuples into disjunct sets, but any positive tuple can be a match for any subset of rules. In our opinion, this models the reality of a medical database the closest.

Conceptually, the association rules are generated first, along with a desired support and confidence parameter. The tuples are then generated accordingly to these parameters. This sequence however, leads to complicated computations of which elements in the positive and negative tuple set need to match which association rules to satisfy the support and confidence parameter of each rule. We have used the following trick to simplify the generation. The positive and negative tuples are generated first in a totally random manner. We keep the number of true attributes small to model reality closer (i.e. it does not make sense for a person to take 10 out of 100 medications, the actual number of distinct medications taken is usually small). In addition we did not want to introduce unintended associations. If we set a high number of binary attributes to true, the probability is high that the data would contain an unintended, hidden association rule. Given the total number of binary attributes $N_{tot}$, we set $k = N(\sqrt{N_{tot}}, ln(N_{tot}))$ to true, where $N$ is a normal distributed random variable with average $\mu = \sqrt{N_{tot}}$ and $\sigma = ln(N_{tot})$. We were satisfied with the resulting number of true attributes. *discuss how it changes and how many total attributes we targeted*

The next step then was to generate the association rules. Our objective was to generate rules which matched only few of the randomly generated tuples. The intuition behind this scheme is that if an association rule matches many randomly generated tuples, then it is too general. So we chose a threshold $t = 1\%$ and if the number of matching tuples in the randomly generated tuples passed it, the association rule was discarded. We then assigned to every found association rule a support parameter in the interval $S = U(\lfloor \frac{N_{pos}}{4}, N_{pos} \rfloor)$ with $U$ standing for the uniform distribution. This support only determines matching tuples in the positive tuple set. We then generated the confidence parameter, which we set to be higher than 99%. This parameter determined how many tuples in the negative tuple set were to be set. Again we used a uniform distribution to compute the actual number $C = U(\lfloor 0, \frac{S}{99} \rfloor)$.

Knowing the number of desired matches in each the positive and negative tuple set for each association rule, it was a simple task to generate the data. For each association rule, we choose as many tuples as needed in a random manner and set the attributes needed to transform the tuple into a matching tuple. The method used can be described best by looking at an association rule as a set of nodes in our forest of hierarchy trees — each literal in the rule corresponds to exactly one node in the tree. Figure 3 shows a tree with the node numbers printed. The number of binary attributes per literal that were actually set were determined after a uniform distribution $N_{toset} = U(1, N_{leaves})$ where $N_{leaves}$ stands for the number of leaf nodes present in the subtree of the node corresponding to the literal. It was necessary to set multiple binary attributes because our categorical attributes can have a set of values. For example, consider figure 3. If our association rule carried node 138 as an item, a non-empty subset of the seven leaves 140 - 142, 144 - 147 needs to be set. So a random number between 1 and 7 is chosen and random attributes are then set to true.

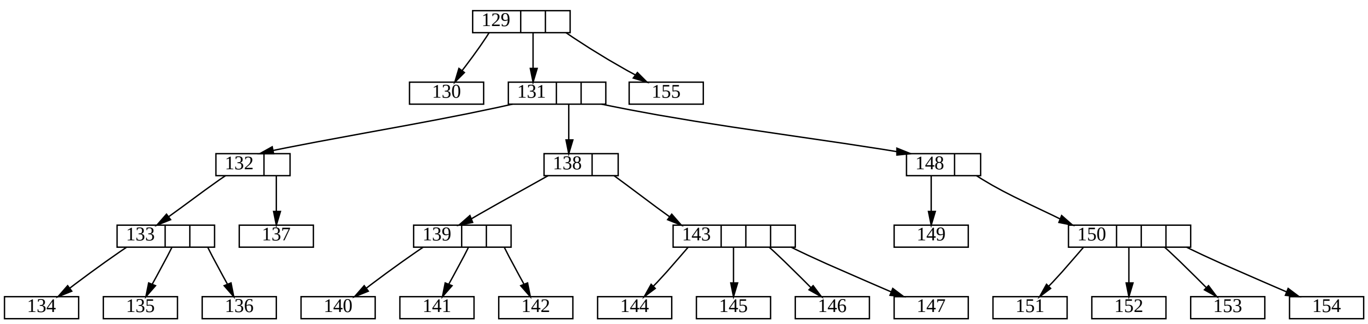


Figure 3: A hierarchy tree including the node number

Here's an example of a short database that was generated. The database after the first randomization looks as follows:

```
<HIV+ database> 10 elements
 0 + 010000000010000010 100010000000100000010000100000000000010 0 000000010000000001000000000 0
 1 + 000000000001001010 000000000000000000000001000000001000 0 000000000000010000000000010 0
 2 + 000000100000000000 000100000001000000000000000000000001001 0 000001000000000100010000000 0
 3 + 000000000100100000 000000100000000000000000010010000001010 0 000000010000000000000100010 0
 4 + 000000001000000000 100010000000000100000000010000000001000 0 000001000000000010000000100 0
 5 + 000000001010000100 0 000000000000000000000000011011001000 0 000001000000000000010000000 0
 6 + 000000000000000100 000001000010000000000000000000000000000 1 010000000000000100000000000 0
 7 + 001000000000000000 000000000000000101001000000000001000000 0 000000001000000010000110100 0
 8 + 000000000110100000 0 000010000000000000000000000000000000 0 000010100000000000100001000 0 0
 9 + 000000100000000000 000000000000000000000000000010000000 10 0 000001001000000000011010100 0 0
<HIV- database> 10 elements
 0 - 010000000000100000 0 101101000000001000000000000000000000 0 000000000000000000100000000 0
 1 - 000101000000000000 000000000000000001000001000000000100010 0 010000100000000000000000001 0
 2 - 010100000000000010 000000000101000000000000010000000000000 0 000000000100001000000000000 0
 3 - 000010010110000100 0 000000000000000000000100000001000000000 0 001000000000000000000000000 0
 4 - 001000000001000100 0 000000000000000000101000000010000100010 0 000000001000000000000000000 0
 5 - 000000100000000000 0 000000000000000001000100000000000010000 0 000000000000000000001000011 01 0
 6 - 001000000000000000 0 000000010000000100001100000000000000000 0 000000000000001000000000000 0 0
 7 - 000100001000000000 0 100010000000101000000000000000000000000 0 000000000000000000001000010 00 0
 8 - 000000010010000000 1 000000000000000000000000000000000000100 0 000000000000000010000000001 0 0
 9 - 000000000010001010 0 000000100000001100000000000000000000000 0 000000000000000000000000000 0 0
```

The tree corresponding to attribute group 0 is shown if figure 4. The data generated looks quite “dense”: many attributes are set to true. With 96 binary attributes, we expect $\sqrt{96} \approx 10$ attributes to be set on average. This is still quite high, which would be against using the square root. But our application is targeted towards a larger application on the order of 1000 attributes — in which case the average number of set attributes would be around 30, a number which we are anticipating.

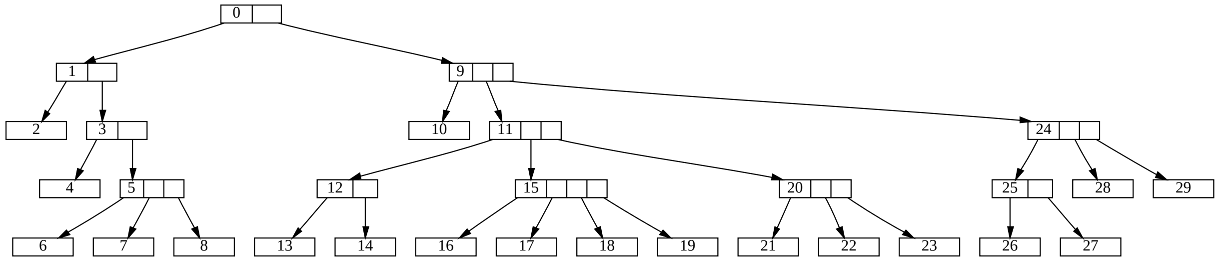


Figure 4: Generated tree for attribute group 0

The association rules were generated in the following steps:

```
Generating association rule # 0
Association rule 0
Candidate with 4 literals
- 011110000000000000 000000000111111111110000000000000000 1 011111111111111110000000000 0
Positive support: 0, Negative support: 0
Support: 3 Confidence: 0

Generating association rule # 1
```

```
Association rule 1
Candidate with 1 literals
- 000000000000000000 0000000000000000011100000000000000000 0 0000000000000000000000000000 0
Positive support: 2, Negative support: 4
Association rule 1
Candidate with 2 literals
- 000001111111111111 0000001110000000000000000000000000000 0 0000000000000000000000000000 0
Positive support: 1, Negative support: 2
Association rule 1
Candidate with 1 literals
- 000000000000000000 0000000000000000000011110000000000000 0 0000000000000000000000000000 0
Positive support: 3, Negative support: 4
Association rule 1
Candidate with 2 literals
- 000000000000000000 0000000000000000000000000000000000000 1 0000000000000000000000000010 0
Positive support: 0, Negative support: 0
Support: 4 Confidence: 0

Generating association rule # 2
Association rule 2
Candidate with 3 literals
- 000000000000000000 0000000000000000000011111111000000000 1 0000000000000000000000000000 1
Positive support: 0, Negative support: 0
Support: 7 Confidence: 0
```

We would like to point out that the association rule number 0 with the pattern `011110000000000000` for attribute group 0 corresponds to node number 3 in the tree shown in figure 4. During the search for association rule number 2, 3 rules were rejected due to a high match count in the negative tuple set.

The next step was to change the positive and negative tuples to match the support and confidence values of our samples. Remember that support in this case is the number of tuples matching the positive tuple set and the confidence is the number of matching tuples in the negative tuple set.

```
<HIV+ database> 10 elements
 0 + 010000000100000010 1000100000010000001000010100000000010 1 0000000100000000010000000000 1
 1 + 000000000001001010 0000000000000000000000011100000001000 1 0000000000000001000000000010 1
 2 + 000100100000000000 0001000000111000000010001001000001001 1 0110110000001000100010000000 1
 3 + 000000001001000000 0000001000000000000000001001000001010 0 0000000100000000000000100010 0
 4 + 001000001000000000 1000100000001010100101010000000001000 1 0111011100000100110000000110 1
 5 + 000000010100001000 0000000000000000000011110111011001000 1 0000010000000000000100000000 1
 6 + 000000000000000100 0000010000100000000000010001000000000 1 0100000000000000100000000010 1
 7 + 011110000000000000 0000000000100011110110000000001000000 1 0000000111000100100000110110 0
 8 + 000000001101000000 0000100000000000000000111011000000000 1 0000101000000000001000010010 1
 9 + 000000100000000000 0000000000000000000000000000100000010 0 0000010010000000000110101000 0
<HIV- database> 10 elements
 0 - 010000000001000000 1011010000000010000000000000000000000 0 0000000000000000001000000000 0
 1 - 000101000000000000 0000000000000000010000010000000100010 0 0100001000000000000000000001 0
 2 - 010100000000000010 0000000001010000000000001000000000000 0 0000000001000010000000000000 0
 3 - 000010010110001000 0000000000000000000010000001000000000 0 0010000000000000000000000000 0
 4 - 001000000010001000 0000000000000000010100000010000100010 0 0000000010000000000000000000 0
```

```
5 - 000000100000000000 00000000000000000100010000000010000 0 0000000000000000000100001101 0
6 - 001000000000000000 00000001000000100001100000000000000 0 0000000000000100000000000000 0
7 - 000100001000000000 10001000000010100000000000000000000 0 0000000000000000000100001000 0
8 - 000000010010000001 00000000000000000000000000000000100 0 0000000000000000100000000010 0
9 - 000000000100010100 00000010000001100000000000000000000 0 0000000000000000000000000000 0
```

We also developed a method to generate a DAG which contains several hierarchy trees. This is very useful because in reality an attribute could belong to different taxonomies.

## 5 Feature data structure

We briefly outline our data structure used in our implementation. Typically we will outline how to represent an item or generalized item, how to represent an association rule and how is the counting done with our specially data structure.

We use the following structure to represent a node in the taxonomy tree. Each node represents an item or generalized item if it is not a leaf.

```
typedef struct node_st
{
    struct node_st *parent;
    int num_child;             /* Number of children, 0 if it is a leaf */
    struct node_st *children;
    int num_leaves;            /* total number of leaves in the subtree */
    int attr_num;              /* is the corresponding attribute number for
                                  a leaf node, otherwise it is used to hold
                                  the least attribute number of all children
                                  leaves */
    int node_num;              /* The unique node id */
    template bv_template;
    int attr_group;            /* stores the id of the tree (attribute group)
                                  the corresponding attribute is in */
    int count_pos;
    int count_neg;
} node;
```

where $template$ is a bit vector, and is defined as:

```
typedef struct
{
    int begin;
    int length;
    int *bv;
```

```
} template;
```

So $template$ is a variable length bit vector.

Consider the following example. In the above taxonomy tree, the template for each node is

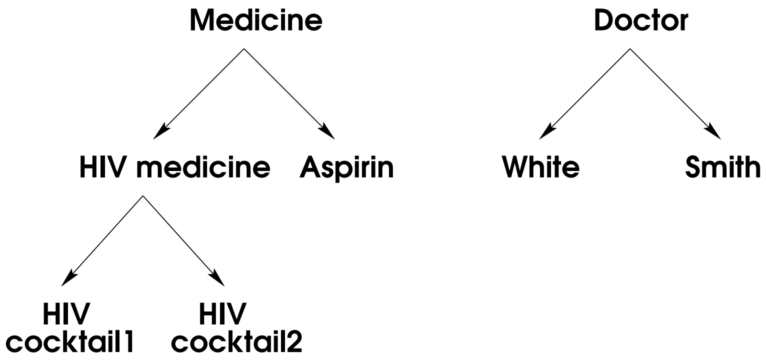


Figure 5: Example of a Taxonomy

(assume length 3 of bit vector for medicine tree, and 2 for doctor tree):

```
Medicine:       111
HIV medicine :  110
HIV cocktail 1: 100
HIV cocktail 2: 010
Aspirin:        001


Doctor: 11
White:  10
Smith:  01
```

Suppose we have rule *medicine = HIV medicine AND doctor = Smith* $\Rightarrow$ *HIV*. The template for this rule is 110 01. A tuple like *tuple 1: medicine = HIV cocktail1, doctor = Smith* has the following template: 100 01, which will match the template of 110 01, since the AND of the template and the tuple is 100 01, which have bit set to 1 in each attribute node of the rule. So the *tuple 1* will support the above template. From this example we can see that we don't need to extend the tuple to count the frequency of the generalized rules, which is done in the IBM algorithms. Also, by using the bit vector, we have a very compact representation for both tuples and candidates, which saves a lot of memeory.

# 6 Experiment Result

We run our experiments on 5 machines: gs5, gs14, gs207 are 266MHz Pentium-II PCs with 64M memory, gs81 is 450MHz Pentium-II PC with 128M memory. Besides the running test program, these machines have low working load.

We designed a set of tests to compare our improved algorithm with algorithm Cumulate. The improved algorithm is expected to improve the performance. For each pass, the candidate sets should be much smaller. Also the improved algorithm is expected to return fewer qualified association rules. This can also be thought of as an automatic rule pruning.

We use our data generation to generate different datasets. The size of these datasets are 200000, 50000, 100000, 500000, and 1000000. The number of attributes vary from 500 to 1000. Our tests are more difficult than some of the IBM testing cases. Although the IBM testing cases have higher number of attributes, the average size of the transactions is 10 and the average size of the frequent itemsets is 4. In our case, the two numbers are higher. In general, the average size of the transactions is the square root of the number of the attributes. We made this decision because it is closer to the real medical databases.

For each test, we output the size of the candidate set for each iteration and the number of rules with high confidence found in every pass. We also output the total time that is used to do the rule mining. The result is shown in the following tables. The basic IBM algorithm described in [SA95] is shown in the first 2 data rows of each table, referred to as "Cumulate". Our improved algorithm "Horizon" is shown in the last two rows of each table and it has a significant speedup of up to 3.5 times. This is due to our "horizon" effect: once a rule has reached the confidence threshold, it is pruned. Therefore all subsequent rules which contain the pruned rule will also be pruned off.

We can see that the running time is strongly dependent on the number of rules searched. A data base which contains a high number of associations will take therefore much longer to process.

# 7 Discussion

## 7.1 Other possible methods for mining the most simplified association rule set

### 7.1.1 ID3

ID3[Qui86] is used to build the decision tree to use as a classifier. But it is possible to use it to find the association rules. A path from the root to the leaf of the tree can be considered as a associated rules. And since in each node (say node $k$) of the decision tree, the number of positive tuples and negative tuples are stored. We can compute the support for the rule got from rule to the current node $k$. The problem with this method is that it may not find all of the association rules. We need to build a decision tree with generalized items. Building a generalized decision tree will be very interesting since we expect it will be better than the classical decision in that it may be more robust to noise. So we can avoid the overfitting problem and save the expensive prune stage.

### 7.1.2 Genetic algorithm

We use Find-S or Least general algorithm to find initial generalized association rules, then use it to generate the initial population. With the two operators of *crossover* and *mutation*, we run the genetic algorithm to generate new association rules. The problem is that genetic algorithm is randomize

| 50000 tuples, 500 attributes, 40 number of attribute groups | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Time |
| Cumulate | 114 | 1270 | 3902 | 5224 | 3769 | 1280 | 146 | 18 | 304 |
| # of rules | 0 | 0 | 196 | 1466 | 2703 | 1252 | 146 | 18 | |
| Horizon | 114 | 1270 | 3706 | 3751 | 1059 | 28 | | | 229 |
| # of rules | 0 | 0 | 196 | 771 | 633 | 8 | | | |

| 50000 tuples, 552 attributes, 40 attribute groups | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Time |
| Cumulate | 113 | 1379 | 5609 | 11793 | 14371 | 11089 | 4617 | 738 | 48 | 1124 |
| # of rules | 0 | 0 | 154 | 1391 | 4851 | 6909 | 3933 | 694 | 48 | |
| Horizon | 113 | 1379 | 5455 | 10400 | 9508 | 4175 | 683 | 44 | | 881 |
| # of rules | 0 | 0 | 154 | 689 | 1542 | 1122 | 281 | 42 | | |

| 10000 tuples, 520 attributes, 40 attribute groups | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Time |
| Cumulate | 111 | 1380 | 5900 | 12911 | 16618 | 12186 | 3904 | 148 | 12 | 263 |
| # of rules | 0 | 12 | 476 | 2971 | 7146 | 7878 | 3255 | 148 | 12 | |
| Horizon | 111 | 1368 | 5423 | 9928 | 9453 | 4299 | 646 | | | 162 |
| # of rules | 0 | 12 | 367 | 1288 | 1968 | 1404 | 371 | | | |

| 500000 tuples, 609 attributes, 50 attribute groups | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Time |
| Cumulate | 137 | 1361 | 4417 | 6792 | 6754 | 3785 | 1064 | 166 | 24 | 4458 |
| # of rules | 0 | 0 | 87 | 1413 | 4008 | 3302 | 1064 | 166 | 24 | |
| Horizon | 137 | 1361 | 4330 | 5379 | 2745 | 483 | | | | 3402 |
| # of rules | 0 | 0 | 87 | 997 | 1670 | 347 | | | | |

Table 1: Tests on gs81

| 50000 tuples, 524 attributes, 40 attribute groups | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Time |
| Cumulate | 114 | 1250 | 4290 | 7153 | 7063 | 4204 | 1125 | 10 | 770 |
| # of rules | 0 | 0 | 80 | 835 | 2298 | 2543 | 1007 | 5 | |
| Horizon | 114 | 1250 | 4210 | 6317 | 4760 | 1655 | 118 | 5 | 653 |
| # of rules | 0 | 0 | 80 | 607 | 1092 | 710 | 138 | 2 | |

| 10000 tuples, 541 attributes, 40 attribute groups | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Time |
| Cumulate | 114 | 1280 | 4631 | 7787 | 7990 | 5126 | 1572 | 120 | 152 |
| # of rules | 0 | 25 | 641 | 3387 | 6430 | 4959 | 1572 | 120 | |
| Horizon | 114 | 1255 | 3990 | 4385 | 1473 | 146 | | | 73 |
| # of rules | 0 | 25 | 414 | 871 | 476 | 39 | | | |

| 10000 tuples, 554 attributes, 40 attribute groups | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Time |
| Cumulate | 138 | 1790 | 8413 | 20581 | 27415 | 20506 | 7849 | 1252 | 96 | 724 |
| # of rules | 0 | 71 | 1394 | 7729 | 17971 | 17917 | 7659 | 1252 | 96 | |
| Horizon | 138 | 1719 | 7016 | 12811 | 9383 | 2574 | 183 | | | 271 |
| # of rules | 0 | 71 | 815 | 2406 | 3004 | 1435 | 225 | | | |

| 100000 tuples, 1272 attributes, 90 attribute groups | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Time |
| Cumulate | 170 | 1672 | 4721 | 6992 | 6638 | 3733 | 1244 | 295 | 48 | 1228 |
| # of rules | 0 | 2 | 647 | 3936 | 5967 | 3714 | 1244 | 295 | 48 | |
| Horizon | 170 | 1670 | 4057 | 3033 | 666 | 12 | | | | 613 |
| # of rules | 0 | 2 | 551 | 1168 | 280 | 15 | | | | |

| 100000 tuples, 1297 attributes, 90 attribute groups | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Time |
| Cumulate | 149 | 1526 | 3646 | 4340 | 3882 | 1885 | 314 | 939 |
| # of rules | 0 | 0 | 137 | 962 | 2318 | 1660 | 310 | |
| Horizon | 149 | 1526 | 3509 | 3375 | 1556 | 222 | 4 | 765 |
| # of rules | 0 | 0 | 137 | 557 | 847 | 242 | 17 | |

Table 2: Tests on gs14

| 50000 tuples, 568 attributes, 40 number of attribute groups | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Time |
| Cumulate | 109 | 1181 | 3031 | 4293 | 4466 | 2517 | 531 | 18 | 458 |
| # of rules | 0 | 0 | 136 | 1120 | 2503 | 2004 | 507 | 18 | |
| Horizon | 109 | 1181 | 2895 | 3173 | 1935 | 499 | 23 | | 336 |
| # of rules | 0 | 0 | 136 | 562 | 636 | 192 | 33 | | |

| 50000 tuples, 556 attributes, 40 attribute groups | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Time |
| Cumulate | 111 | 1410 | 4644 | 6702 | 5468 | 2737 | 678 | 36 | 720 |
| # of rules | 0 | 0 | 175 | 1520 | 3408 | 2664 | 678 | 36 | |
| Horizon | 111 | 1410 | 4469 | 5170 | 2045 | 73 | | | 510 |
| # of rules | 0 | 0 | 175 | 949 | 902 | 59 | | | |

| 100000 tuples, 1187 attributes, 90 attribute groups | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Time |
| Cumulate | 164 | 1586 | 5505 | 11659 | 17021 | 15759 | 9828 | 4536 | 972 | 3877 |
| # of rules | 0 | 0 | 204 | 2784 | 9348 | 13223 | 9600 | 4536 | 972 | |
| Horizon | 164 | 1586 | 5301 | 8851 | 7655 | 2528 | 228 | | | 2094 |
| # of rules | 0 | 0 | 204 | 1917 | 3497 | 2255 | 310 | | | |

| 10000 tuples, 494 attributes, 40 attribute groups | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Time |
| Cumulate | 113 | 1560 | 7555 | 19612 | 30853 | 28797 | 13968 | 2574 | 933 |
| # of rules | 0 | 20 | 1020 | 7276 | 19450 | 23315 | 12837 | 2556 | |
| Horizon | 113 | 1540 | 6532 | 12326 | 11391 | 5481 | 1131 | 18 | 341 |
| # of rules | 0 | 20 | 833 | 3526 | 5714 | 4249 | 1591 | 240 | |

Table 3: Tests on gs5

| 10000 tuples, 631 attributes, 40 attribute groups | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Time |
| Cumulate | 128 | 1619 | 4033 | 4715 | 3783 | 1774 | 384 | 3 | 81 |
| # of rules | 0 | 19 | 442 | 2068 | 2926 | 1681 | 383 | 3 | |
| Horizon | 128 | 1600 | 3591 | 2645 | 782 | 73 | 1 | | 51 |
| # of rules | 0 | 19 | 297 | 594 | 195 | 36 | 0 | | |

| 10000 tuples, 508 attributes, 40 attribute groups | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Time |
| Cumulate | 126 | 1616 | 6174 | 11841 | 13368 | 7706 | 1656 | 263 |
| # of rules | 0 | 37 | 1008 | 5399 | 10089 | 7156 | 1652 | |
| Horizon | 126 | 1579 | 5163 | 6434 | 3276 | 550 | 4 | 115 |
| # of rules | 0 | 37 | 710 | 2031 | 1487 | 368 | 9 | |

| 10000 tuples, 491 attributes, 40 attribute groups | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Time |
| Cumulate | 124 | 1683 | 8232 | 20381 | 28286 | 21507 | 8062 | 1060 | 48 | 798 |
| # of rules | 0 | 31 | 871 | 7329 | 18009 | 18547 | 7729 | 1058 | 48 | |
| Horizon | 124 | 1652 | 7360 | 13041 | 10148 | 2738 | 306 | 2 | | 312 |
| # of rules | 0 | 31 | 703 | 3432 | 3357 | 1607 | 152 | 4 | | |

| 100000 tuples, 1120 attributes, 90 attribute groups | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Time |
| Cumulate | 149 | 1349 | 3576 | 5534 | 4839 | 1983 | 490 | 54 | 986 |
| # of rules | 0 | 0 | 430 | 2247 | 3520 | 1827 | 489 | 54 | |
| Horizon | 149 | 1349 | 3146 | 3280 | 1291 | 143 | | | 623 |
| # of rules | 0 | 0 | 430 | 988 | 460 | 26 | | | |

| 100000 tuples, 1301 attributes, 90 attribute groups | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Time |
| Cumulate | 162 | 1711 | 5671 | 10035 | 11963 | 7980 | 2367 | 174 | 2438 |
| # of rules | 0 | 2 | 431 | 3520 | 8824 | 7506 | 2367 | 174 | |
| Horizon | 162 | 1709 | 5239 | 6505 | 3130 | 472 | | | 1503 |
| # of rules | 0 | 2 | 382 | 1990 | 2800 | 683 | | | |

Table 4: Tests on gs207

| 500000 tuples, 1157 attributes, 90 attribute groups | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Time |
| Cumulate | 142 | 1150 | 1845 | 2109 | 1636 | 670 | 125 | 6 | 1724 |
| # of rules | 0 | 0 | 94 | 814 | 1545 | 670 | 125 | 6 | |
| Horizon | 142 | 1150 | 1751 | 1294 | 90 | | | | 980 |
| # of rules | 0 | 0 | 94 | 398 | 243 | | | | |

| 500000 tuples, 1211 attributes, 90 attribute groups | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Time |
| Cumulate | 173 | 1502 | 2762 | 4553 | 4778 | 2554 | 528 | 12 | 4051 |
| # of rules | 0 | 0 | 71 | 1107 | 2946 | 2341 | 528 | 12 | |
| Horizon | 173 | 1502 | 2691 | 3442 | 1825 | 189 | | | 2929 |
| # of rules | 0 | 0 | 71 | 782 | 981 | 300 | | | |

| 1000000 tuples, 477 attributes, 40 attribute groups | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Itemsetsize | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Time |
| Cumulate | 96 | 1016 | 3187 | 4311 | 2487 | 888 | 146 | 6749 |
| # of rules | 0 | 0 | 5 | 184 | 872 | 698 | 146 | |
| Horizon | 96 | 1016 | 3182 | 4127 | 1615 | 190 | | 6285 |
| # of rules | 0 | 0 | 5 | 156 | 526 | 179 | | |

Table 5: Tests on no.trust

algorithm. So we need to scan the database many passes in order to make it converge. That is expensive. But if we can find a good initial population, then the number of the passes to scan the database may be not many.

## 7.2 Other possible improvements

- For each one-item set, we store pointers to its first and last occurrence. So for each candidate, we can compute the upper-bound of the first occurrence place and the lower-bound of the last occurrence place. The tuples between these bounds are the only ones in the database that we need to scan. This will speed up a lot, especially in the case of sparse database like IBM basket database.
- Adaptive support. Right now all of the algorithms don't mention how to select the threshold of support and confidence to generate the candidates. They all use fix support threshold which is predefined by the users. The problem is that for different databases with different characteristics, users may like to find rules with different supports. And if the users have not proper view of this database, then it will be difficult for him to choose the support thresholds. One possible to deal with this is make the algorithm to select the support threshold for the user by the statics of the supports of each 1-itemset.
- Our program can still be improved in running speed. For instance, we can create a hierarchy of the candidates and first match the candidates which are more general — if the more general candidate does not match a tuple, the more specific one will certainly not match it either. We can also order the attribute groups such that we check for infrequent attributes first to stop the matching as fast as possible. Ideally we could compile code for each candidate and pass the tuple as argument, the candidate would then evaluate a match with the highest speed possible and call the matching function of potential siblings as well. Our candidate generation is currently also inefficient, with a smart data structure we could reduce the complexity from $O(n^2)$ to $O(m)$ where $n$ are the number of candidates in the current candidate set and $m$ are the number of candidates in the next set. Another improvement can be achieved by streamlining our candidate hash function. We currently use the entire bit vector representation to compute the hash code. An improved version would only take the list of items in a candidate and hash it, this list is generally much shorter than the bitvector.

## 7.3 Future Work

We have written the improved algorithm based on IBM's Cumulate algorithm, and test it on the database generated by our data generation algorithms. The future work will be to implement the improvements we propose and test it on the real medical database and other real database like Giant Eagle database.

We are also thinking about how to build a generalized decision tree, which means a generalized item could be a node in the decision tree. This seems an interesting topic since such decision tree should be more robust to noise, thus avoid the overfitting problem or save the prune stage which had to be done in ID3 or other decision tree algorithms.

## 8 Conclusion

We investigate the problem of mining the generalized association rules in the medical database in order to protect the sensitive information when the medical database is released. We developed a faster algorithm based on Cumulate, IBM's frequency counting algorithm, for mining generalized association rules. Our algorithm also automatically prune redundant rules which gives a better result. Firstly by using a bit template for each node in the taxonomy tree, we don't need to extend the tuples in the database to take into account the ancestors of each items in the tuple. Secondly, by extracting the candidates with very high support and confidence at each pass, we highly reduce the sizes of the candidate sets that will be generated in the following pass. Finally, we propose some promising further improvements of the algorithm. We also proposed a fast greedy algorithm on how to generalize databases with minimum information loss.

To test our algorithm, we developed our data generation engine to generate the database according to the medical database model. The experiments over the synthesized datasets show that our algorithm out-performs the existing ones.

## 9 Acknowledgements

We want to thank especially Latanya Sweeney for her help and support. She has lead us to the problem of finding association rules in medical databases. We also thank Christos Faloutsos for his reviews of our progress and his useful feedback.

## References

[AM98] Brigham Anderson and Andrew Moore. Adtrees for fast counting and for fast learning of association rules. 1998.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *In Proceedings of the 20th VLDB Conference*, 1994.

[ATA93] R. Agrawal, T.Imilienski, and A.Swami. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, 1993.

[BMUT97] Sergey Brin, Rajeev Motwani, Jeffrey Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the ACM SIGMOD*, 1997.

[Mit97] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[ML97] Andrew W. Moore and Mary Soon Lee. Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research*, July 1997.

[Qui86] J.R. Quinlan. Induction of decision trees. *Machine learning*, 1:81–106, 1986.

[RS98] Rajeev Rastogi and Kyuseok Shim. Public: a decision tree classifier that integrates building and pruning. In *Proc. 24th VLDB Conference.*, pages 404–415, 1998.

[SA95] Ramakrishman Srikant and Rakesh Agrawal. Mining generalized association rules. In *Proceedings of the 21st VLDB Conference*, 1995.

[SVA97] Ramakrishman Srikant, Quoc Vu, and Rakesh Agrawal. Mining association rules with item constraints. In *Proceeding of AAAI*, 1997.

[Swe98] Latanya Sweeney. Generalization as a technique for producing anonymous data. In *Proceedings of the ACM Principles of Database Systems*, 1998.