Performance Evaluation of Attribute-Oriented Algorithms for Knowledge Discovery from Databases: Extended Report

Colin L. Carter and Howard J. Hamilton
Dept. of Computer Science, University of Regina, Regina, SK, Canada, S4S 0A2
email: {carter, hamilton}@@cs.uregina.ca

Abstract

Practical tools for knowledge discovery from databases must be efficient enough to handle large data sets found in commercial environments. Attribute-oriented induction has proved to be a useful method for knowledge discovery, but two algorithms which implement it, AOI and LCHR, have some limitations due to excessive memory usage. AOI and LCHR generalize database relations in O(np) and O(n log n) time respectively, where n is the number of input tuples and p is the number of tuples in the output relation. Their space requirements are O(n) since they store the whole input relation in memory or on disk. In contrast, we present GDBR, a space efficient, optimal O(n) algorithm for relation generalization. We also present the results of empirical comparisons between GDBR, AOI and LCHR. We have implemented efficient versions of each algorithm and empirically compared them on large commercial data sets. We present the results of these tests, showing that GDBR is consistently faster than AOI and LCHR. GDBR's times increase linearly with increased input size, while times for AOI and LCHR increase non-linearly when memory is exceeded. Through better memory management, however, AOI can be improved to provide some advantageous capabilities.

Keywords: machine learning, knowledge discovery, data mining, attribute-oriented induction

1. Introduction

Knowledge discovery from databases (KDD) is the nontrivial extraction of implicit, previously unknown, and potentially useful information from data [9]. As a branch of machine learning, KDD encompasses a number of automated methods whereby useful information is mined from data stored in databases. When a KDD method is implemented as a practical tool for knowledge discovery in databases, an important requirement is that it be as efficient as possible so that it can handle the large input data sets typically encountered in commercial environments. This paper presents the results of our efforts to compare implementations of three similar KDD algorithms to determine their suitability for application to large scale commercial databases.

Our work has been motivated by the desires of corporate sponsors such as Roger's Cablevision to use our knowledge discovery tools on databases consuming 5 gigabytes of memory and containing millions of records. Discovery tasks that we have explored have usually been geared to relating two or three attributes to each other to see if there is any interesting relationship between them. For example, is there any significant relationship between a customer's basic cable subscription (basic cable services, family channel, movie channel, etc.) and the rating of the pay per view (ppv) movies that the customer rents? This discovery task might lead to the more effective targeting of advertising campaigns for ppv movies on cable stations. Alternatively, is there any relationship between the area in which a customer lives and the types of services subscribed to or the ppv movies rented? We found that certain areas had very high occurrences of one product and very low occurrences of others. Again, this can aid in effective

target marketing campaigns. These types of queries typically related generalizations of only two or three attributes to each other, but access potentially millions of records.

Attribute-oriented induction is a KDD generalization technique which, given a relation retrieved from a relational database, generalizes its contents on an attribute by attribute basis [2]. The goal of relation generalization is to produce a small table of information that accurately summarizes the input data to such a degree that people can easily recognize the interesting patterns that emerge from the generalized data. We have found attribute oriented generalization techniques to produce useful and informative summaries of input data in much less time than standard database techniques commonly used by data managers in commercial environments. Time savings are realized both in terms of defining the generalization paths that will be used on the data and in the actual summarization of data retrieved from the database.

The primary algorithm for attribute-oriented induction, which we call AOI, runs in $O(np)$ time where n is the number of input tuples (database records) and p is the number of tuples in the final generalized relation [10]. A related algorithm, LCHR (**L**earn **CH**aracteristic **R**ule), accomplishes the same result, but differs slightly in its method [3]. LCHR runs in $O(n \log n)$ time where n is the number of input tuples [2]. The space requirement of both AOI and LCHR is $O(n)$ since they both store the whole input relation either in memory or on disk.

Our first research efforts focused on implementing efficient versions of attribute-oriented algorithms so that tools based on them will handle large data sets [7], [8]. We have found, however, that neither AOI or LCHR as published scales well to large input data sets. Although both these algorithms are relatively efficient for moderate input sizes, they are not optimal. In addition, their memory requirements grow with increased input size, causing excessive memory use and resultant performance degradation for large inputs.

We therefore present GDBR, an on-line and space efficient attribute-oriented algorithm which accomplishes the same result as AOI and LCHR but runs in $O(n)$ time, which is optimal. For typical knowledge discovery tasks, GDBR uses a small, constant amount of memory. In Section 2 we describe relevant portions of AOI and LCHR. In Section 3 we describe GDBR and the basic data structures and methods the algorithm uses. In Section 4 we present the algorithm more formally. In Section 5 we summarize the complexity analyses of AOI, LCHR and GDBR. In Section 6 we present empirical test results. In Sections 7, we describe some improvements to AOI which make it more useful and conclude the paper in Section 8. A short form of this paper will appear as [6].

2. Overview of Attribute-Oriented Induction, AOI and LCHR

An attribute-oriented induction algorithm takes as input a relation retrieved from a database and generalizes the data guided by a set of concept hierarchies [11]. A *concept hierarchy* is a tree of concepts arranged hierarchically according to generality. For discrete valued attributes, leaf nodes correspond to actual data values which may be found in the database. For continuous (usually numerical) valued attributes, leaf nodes represent discrete ranges of values. Higher level nodes represent more general concepts created by combining groups of lower level concepts under unique names.

After retrieving task relevant data from a database, the first step of generalization is to convert the data values to matching leaf concepts from the appropriate concept hierarchy. This initial conversion, however, is not the focus of our algorithm. This paper, like [10], is primarily concerned with the generalization algorithm itself which takes a relation from ungeneralized

concepts to appropriately general concepts.

The generalization process is limited by a set of user defined *attribute thresholds* which specify the maximum number of distinct values that may exist for each attribute of the generalized relation. When each attribute of the input relation has been generalized to within the bounds of its threshold, many tuples of the relation are identical to other tuples. A count of each set of identical tuples is then stored in one and the rest are eliminated. The result is called the *prime relation*.

The algorithms AOI and LCHR use the same first stage of generalization, called *PreGen* in [10]. Both make one pass through the input relation to compile statistics about how many distinct values of each attribute the input relation contains. These values are then generalized by ascending the relevant concept hierarchies until the total number of distinct values for each attribute falls within the range of that attribute's threshold. This produces a number of concept pairs where each ungeneralized concept is matched with a higher level concept.

AOI and LCHR handle the replacement of ungeneralized concepts and removal of duplicates somewhat differently. AOI loops through the input relation on a tuple by tuple basis, replaces the value of each attribute with a generalized concept, and immediately inserts the generalized tuple into a dynamically constructed prime relation. The prime relation is initially empty. The first tuple encountered is inserted at the beginning of the relation, and its count variable is initialized to one. Each subsequent tuple read from the input relation is sequentially compared with tuples in the prime relation. If an identical tuple is found, a count variable for that tuple is incremented, and the tuple to be inserted is discarded. If no matching tuple is found, the tuple to be inserted is added to the end of the prime relation.

LCHR loops through the input relation, replaces all data values with generalized concepts, then sorts the resulting relation and removes duplicates with one final pass. The final result of both algorithms is a small relation of unique tuples, each of which summarizes the number of tuples that it represents from the input.

3. GDBR Overview

GDBR incorporates several enhancements to the data structures used by AOI and LCHR and approaches the duplicate removal process in a more efficient way. First, GDBR makes use of an augmented concept hierarchy structure to eliminate the need to replace concepts with more general concepts. Secondly, as it reads tuples from the database and converts these tuples to leaf concepts, it assigns an order to these concepts based on a first encounter principle. Thirdly, the number of distinct concepts encountered for each attribute is tracked as the input relation is read, and as soon as the attribute threshold is exceeded, the algorithm immediately increases the level of generalization. Finally, GDBR uses information about the number of attributes in the input relation and the attribute thresholds of each attribute to structure the prime relation. The encounter orderings of a tuple's component concepts are then used to insert tuples into the prime relation in a single step as each is read. While this may require the prime relation to be reorganized a few times, it avoids both the need to perform n searches of the prime relation to insert input tuples, as in AOI, or the need to sort a relation of size n, as in LCHR.

3.1 Augmented Concept Hierarchies

For each concept in a hierarchy, we construct a *path array,* an array of pointers to concepts which represents the concept's path to the root of the tree. Sibling leaf nodes share a single path array, since the path to the root from each is the same. Each parent node shares a portion of a leaf descendant's array, since a parent's path is a subpath of any of its descendants' paths. Arrays, therefore, are only constructed for each group of leaf siblings. An augmented tree structure for the attribute PROVINCE is shown in Figure 1.

In addition to the path arrays, we assign each concept a *distance_to_max* value which is the difference between its depth and the depth of the deepest leaf concept. In Figure 1, the *distance_to_max* of each level of the tree is marked to the right of the tree. For example, the *distance_to_max* of *Alberta* is 0 since it is at maximum depth, and that of *Quebec* is 2 since its depth is two levels less than the deepest leaf nodes.

For each attribute in the input relation, the algorithm keeps track of a *generalization level*, an integer representing how many times the attribute has been generalized. Each generalization level is initially set to 0. As the need for generalization is detected, the level is incremented.

We define an access function, *get_generalized_concept,* which takes as input any concept and a generalization level and returns a generalized concept at the appropriate level of generality. If the generalization level exceeds the *distance_to_max* of an input concept, the path array is accessed according to the difference of these two values and the concept at that index is returned. Otherwise, the input concept is returned. For example, for the hierarchy shown in Figure 1, the call *get_generalized_concept(B.C., 1)* will return *B.C.* because *B.C.*'s *distance_to_max* does not exceed the generalization level. However, *get_generalized_concept(B.C., 2)* will return *Western*.

When input attribute values are read from the database, they are converted to concepts and stored as such. As the input relation is generalized, the generalization level is incremented. Since concepts are only accessed by the *get_generalized_concept* function, this inherently generalizes

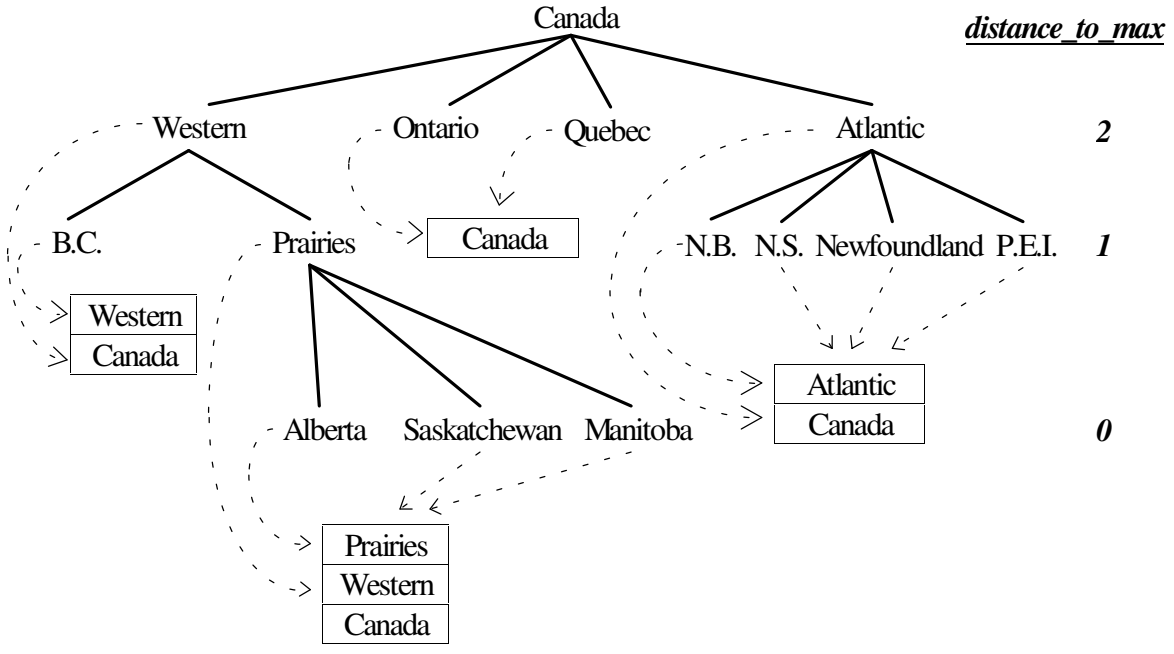

Figure 1. An Augmented Concept Hierarchy for the Attribute PROVINCE

the values already stored without having to replace the concepts themselves. It also saves GDBR from having to scan the input relation a second time to replace leaf values with generalized concepts as both LCHR and AOI do.

3.2 Encounter Ordering

To avoid sorting the generalized relation, we define an encounter ordering on the input concepts. Each node in the concept hierarchy is given an ordinal field which is initialized to 0. For each attribute in the input relation, we also keep a *distinct_value_count,* an integer representing how many distinct concept values have been encountered for that attribute at the current level of generalization. Each *distinct_value_count* is initialized to 0 at the start of a generalization task. When a tuple is read from the database, each attribute value is converted to a concept and the ordinal of that concept is examined. If its value is 0, the *distinct_value_count* variable is incremented and the result is assigned to the concept as its ordinal. In essence, therefore, we both define an order on the input concepts in terms of a concept's first encounter and keep track of how many distinct values have been encountered.

3.3 Progressive Generalization

The GDBR algorithm progressively generalizes the input relation as the data is read. As tuples are retrieved from the database and each new concept is encountered, the *distinct_value_count* variable for that attribute is incremented and compared to the attribute's threshold. If the attribute threshold has not been exceeded, the concept is stored in a *distinct_values* array which keeps a record of all concepts encountered at the current generalization level. When the number of distinct values exceeds the attribute threshold, the concepts in this array are repeatedly generalized until the total number of concepts, including the one newly encountered, again falls within the bounds of the attribute threshold. The *distinct_values* array and *distinct_value_count* are then adjusted to reflect this level of generality.

3.4 Duplicate Elimination

To handle duplicate tuples efficiently, we construct the prime relation as an *m* dimensional array where *m* is the number of attributes in the input relation. The size of each dimension is determined by the attribute threshold for each attribute. Tuples are "inserted" into the prime relation using the ordinal values of the tuple's component concepts as indices into the appropriate dimension of the array. *Inserting* simply means compiling statistics about the inserted tuple, that is, incrementing a counter tracking the number of tuples inserted in that cell and possibly summing values of numerical attributes. Other summarization operations may also be performed.

Since we generalize concepts as soon as the attribute threshold is exceeded and before the tuple is inserted into the prime relation, we will always be able to insert any tuple into the prime relation. This means, however, that when any attribute threshold is exceeded and a *distinct_values* array is adjusted, we may also need to rearrange some of the contents of the prime relation to reflect the changes of some concepts' indices. Immediately after a tuple has been inserted, the prime relation accurately reflects the data read in. Upon reading the last tuple, processing is complete and no further scanning, sorting or reorganizing is necessary.

4. GDBR Algorithm

In this section, we describe the GDBR algorithm (Algorithm 1) which is discussed in more detail in [4] and [5]. We assume that a discovery task has been defined and the database initialized to retrieve input data. We also assume that the tuple arrives from the database retrieval process with its attribute values converted into leaf concepts from the appropriate hierarchies. For simplicity, we also assume that the attribute threshold is the same for every attribute. The algorithm can be easily extended to allow different attribute thresholds for each attribute.

The *allocate_distinct_values_array* and *allocate_prime_relation* procedures dynamically allocate these structures. The prime relation minimally stores a pointer to the *distinct_values* array and an integer count for each possible combination of concept values. The *get_next_tuple*

```
Algorithm: GDBR - Generalizes a Relation to the Attribute Threshold
Input:  attr_count - the number of attributes in the input relation
        attr_threshold - the attribute threshold
        hierarchies - an array of concept hierarchies matching the input relation's attributes
Output: the prime relation
procedure generalize_database_relation (
        attr_count : integer,
        attr_threshold : integer,
        hierarchies : array[attr_count] of ConceptHierarchy )
var
    i : integer;
    distinct_values : array[attr_count, attr_threshold] of ConceptNode;
    distinct_value_counts : array[attr_count] of integer;
    concept : ConceptNode;
    tuple : Tuple;
    prime_relation : Relation;

    { dynamically allocate distinct_values array and prime relation }
    distinct_values := allocate_distinct_values_array( attr_count, attr_threshold );
    prime_relation := allocate_prime_relation( attr_count, attr_threshold );
    prime_relation.distinct_values = distinct_values;
    while ( tuple := get_next_tuple() )
        for i := 1 to attr_count do
            concept := get_generalized_concept( tuple.attribute[i] );
            if ( concept.ordinal = 0 ) then  { concept has not been seen yet }
                distinct_value_counts[i] = distinct_value_counts[i] + 1;
                if ( distinct_value_counts[i] > attr_threshold ) then
                    distinct_value_counts[i] := generalize_concepts( distinct_values[i], concept );
                    generalize_relation( prime_relation, i );
                else
                    concept.ordinal := distinct_value_counts[i];
                    distinct_values[concept.ordinal ] := concept;
                end  { if }
            end { if }
        end { for }
        insert_tuple( prime_relation, tuple );
    end { while }
    return prime_relation;
end { generalize_database_relation }
```

Algorithm 1. GDBR

procedure retrieves a tuple from the database and converts its attribute values to leaf concepts from the appropriate concept hierarchies. The *get_generalized_concept* procedure retrieves a generalized concept at the current level of generality as described in Section 3.1. The *generalize_concepts* procedure takes as input the array of concepts encountered so far and the new concept that caused the attribute threshold to be exceeded. It generalizes the concepts the minimum number of levels necessary to insert the new concept into the array and still be within the attribute threshold. It then returns the number of distinct values in the array after successful adjustment. The *generalize_relation* procedure takes the prime relation and the current attribute index as arguments. It moves any tuples whose concepts have had a change of ordinal to the position the new ordinal determines. The *insert_tuple* procedure inserts the tuple into the prime relation as described in Section 3.4.

5. Time and Space Analyses

In this section, we present the time and space analyses of AOI, LCHR and GDBR. The analysis of AOI is summarized from [10] and discussed in more detail in [5]. The GDBR analysis is also presented in more detail in [5].

5.1 Time Analyses

5.1.1 Time Analysis of AOI

Theorem 1 (Han, [10]): The worst-case time complexity of [AOI] is O(np), where n is the number of tuples of the initial relation R, and p is the number of tuples of the prime relation P.

Proof Sketch:

The O(np) time analysis is derived primarily from the insertion of n generalized tuples into the prime relation of size p. Based on the above theorem and a prototypical implementation of AOI (DBLearn, [1]) which we received from the author, we have concluded that the prime relation is constructed in unsorted order and a linear search of the relation is required to match an inserted tuple with a tuple in the prime relation. The maximum number of comparisons for each insertion is therefore p for p tuples in the prime relation and O(np) for n inserted tuples. ■

We note that the actual value of p can be calculated from the number of attributes in the input relation and the attribute thresholds for each. If a common attribute threshold t is used, the maximum size of the prime relation will be t^m for m attributes. If a distinct attribute threshold, t_i, is used for each attribute m_i, the worst case size of the prime relation will be $p = \prod_{i=1}^{m} t_i$. We restrict the value of t_i to the total number of leaf nodes in the i^{th} concept hierarchy. For discrete valued attributes, this represents the number of distinct attribute values that may be encountered in the data. For continuous valued attributes such as numbers or times, this represents the number of ranges of values that are defined as leaf nodes in the matching concept hierarchy. This restriction on t_i limits the size of p so that it will not grow unreasonably large. In addition, given a fixed number of attributes, p is a bounded value.

5.1.2 Time Analysis of LCHR

Theorem 2: The worst-case time complexity of LCHR is O(n log n) where n is the number of input tuples.

Proof Sketch:

The difference between LCHR and AOI is due to the removal of duplicate tuples and the construction of the prime relation. LCHR replaces each attribute value in the input relation with a generalized value, sorts the result and removes duplicates with another complete pass. Since the lower bound of sorting by comparison of adjacent elements is O(n log n), LCHR is O(n log n). ■

5.1.3 Time Analysis of GDBR

Theorem 3 (From [5])**:** The worst-case time complexity for GDBR is O(n) where n is the number of input tuples, and where the number of attributes, the attribute thresholds, and the depths of all concept hierarchies are small.

Proof Sketch:

We define n as the number of input tuples, m as the number of attributes in the input relation, t as the maximum attribute threshold and d as the depth of the deepest concept hierarchy. The basic loop of the GDBR algorithm runs only n times for an input relation of size n with m iterations to loop through the attributes in each tuple. All operations in these loops are bounded by a small constant measure of work except for the *generalize_concepts* and *generalize_relation* procedures.

There are m *distinct_values* arrays, each holding t concepts. Each array can only be generalized a maximum of d times with a small constant c for the work to generalize each concept. So the total work done by the *generalize_concepts* function is bounded by *cmdt.* Again, these values are extremely small in comparison to n and so may be ignored.

A prime relation of size p is generalized a maximum of md times with a small constant of c as a measure of work to generalize each concept. The upper bound for the *generalize_relation* function is therefore *cmdp.*

The total work for the algorithm, therefore, is $mn + cmdp$. If c, m and d are small in relation to n and p, GDBR is O($n + p$). We have already noted in Section 5.1 that p is a bounded value. In contrast, n is unbounded, so for large n, $p < n$. Overall, therefore, GDBR is O(n). ■

5.1.4 Proof of GDBR's Optimality

Theorem 4: An O(n) algorithm is optimal for relation generalization and therefore GDBR is optimal.

Proof Sketch:

That O(n) is optimal is easily established by a simple adversary argument. We need to show that every tuple in the relation must be examined at least once by the algorithm. We define an adversary that supplies input tuples to GDBR. Assuming for simplicity a common attribute threshold of t, the adversary will supply tuples with differing distinct attribute values for the first t tuples. Subsequently, it will randomly supply any of the attribute values it has already supplied up until the nth tuple. For the nth tuple, it will supply formerly unencountered attribute values for each attribute, causing the attribute threshold to be exceeded for each and the prime relation to be generalized. Should the nth tuple not be examined, the prime relation would contain values of insufficient generality and therefore be an incorrect generalization. Thus the lower bound of relation generalization is O(n) and GDBR is optimal. ■

5.2 Space Analyses

AOI and LCHR may be implemented in two different ways. In the first, all input is stored only in the database, and most processing is done with standard database operations. This is a *disk-based* implementation. Alternatively, the input can be read into main memory, and standard programming techniques can be used to manipulate the data. This is a *memory-based* implementation. While standard database operations are relatively efficient, they inherently rely on access to data stored on disk. In contrast, access to data stored in main memory is much faster. For this reason, we have implemented memory-based versions of AOI and LCHR only.

Both AOI and LCHR require that the data be scanned twice, once for calculating statistics and once for generalization. AOI does not need to store the input in memory for its generalization process, and so can simply read the data twice from the database. Our experience, however, is that the database retrieval time is the largest factor in discovery tasks, so reading from the database twice is a very costly solution, especially for large input. Our version of AOI and the original DBLearn prototype, stores the input in memory to avoid reading the database twice. Since LCHR sorts the generalized relation, it must store the relation in memory unless a much slower external sort is used. Implemented in this way, both AOI and LCHR require $O(n)$ storage for the input relation when storing references to distinct attribute values and $O(p)$ storage for the prime relation. Other storage requirements by the algorithms are few and constant in relation to the size of the input, and so are insignificant. The total requirements therefore are $O(n + p)$, or $O(n)$ since p is small in relation to n. If AOI opts to read data twice, storage requirements will only be $O(p)$, but overall times will increase greatly due to extra database access.

GDBR requires only $O(p)$ space for the prime relation since each tuple is examined only once, immediately inserted into the prime relation and then discarded. This saves the $O(n)$ space requirement of both AOI and LCHR. Unlike AOI and LCHR, however, the prime relation of size p is allocated in a block and therefore is a maximum, worst case size from the start. As noted in Section 5.1.1, however, p is very small in comparison to n.

Should GDBR be used to generalize relations for input to other automated processes and the number of attributes and size of attribute thresholds increase greatly, the size of the prime relation would also expand greatly. Since GDBR allocates this structure to be maximum size, this could put GDBR at a space disadvantage to AOI or LCHR which dynamically build the prime relation as needed. If this were the case, however, the worst case size of the prime relation is still the same for each algorithm, and so the storage requirements would be similar.

6. Empirical Tests of AOI, LCHR and GDBR

We implemented efficient versions of GDBR, AOI and LCHR and ran empirical timing tests for varied input sizes and attribute thresholds. In this section, we present the results of those tests. In Section 6.1 we describe how the tests were structured. In Section 6.2 we present the results of tests that varied input size while keeping attribute thresholds constant. In Section 6.3 we present the results of tests that varied attribute thresholds for a fixed input size.

Our implementation of AOI and LCHR closely followed the algorithms presented in [10] and [3] respectively and originated from a prototypical implementation of DBLEARN [1] supplied to us by J. Han. Every effort was made to make the implementations as fast as possible while remaining consistent with the published algorithms. For example, the DBLEARN prototype was extensively rewritten to enhance memory efficiency and speed [7].

6.1 Testing Methods

We benchmarked the three algorithms to specifically time the generalization process separate from database retrieval and the conversion of database attribute values to leaf concepts. To be able to effectively compare the algorithms, however, some modifications to GDBR were necessary. AOI and LCHR require that the whole relation be read into memory before generalization. The input is read, therefore, and then the actual generalization process timed separately. GDBR, however, is an on-line algorithm. This makes timing just the generalization part impossible since built in timing functions are not precise enough to time the generalization of one tuple each time it is retrieved from the database. We therefore separated the data retrieval portions of GDBR from the actual generalization operations. The input relation was retrieved from the database in blocks of tuples, converted to leaf concepts and stored in memory. This structure was then generalized by GDBR on a tuple by tuple basis as though it were being retrieved from a database.

For GDBR and AOI, we ran each test ten times and computed the average time. This was easily accomplished since neither algorithm is destructive to the input relation, and the same input was used multiple times. LCHR, however, is destructive to the input since each ungeneralized concept is replaced by a generalized concept, and the generalized relation is then sorted in place. To run the test multiple times, the data must be retrieved ten times separately, or the input relation must be duplicated between each generalization. As we will see in Sections 6.2 and 6.3, however, LCHR times were much greater than the GDBR and AOI. In this light, running the test multiple times to get a slightly more precise measurement was in our opinion unnecessary, and LCHR was timed only once for an approximate measure.

We ran the tests on an IBM compatible PC with 32 Megabytes of RAM and an Intel 66 MHz 80486DX2 processor. The operating system was IBM's OS/2 and the database was IBM's DB2/2 relational DBMS. The data was commercial data supplied by a corporate sponsor.

6.2 Tests varying input size

The first set of three tests varied the number of tuples read from the database from 100,000 to 500,000 in 100,000 tuple increments. Only input relations with two or three attributes were generalized since these are typical of the tasks we have been running in actual knowledge discovery sessions. The input relations contained both discrete and continuous (numerical) attributes. In Tests 1 and 3, the numerical attributes were simply generalized without summing their values. In Test 2, the same input relation was used as in Test 1, but a numerical attribute was summed to see the impact of the summation computations. A constant attribute threshold of 4 was used for all attributes.

Test 1 involved an input relation with three attributes, two discrete product codes and a

Table 1. Test 1 Timing Results

	GDBR	*AOI*	*LCHR*
100K	0.53	0.93	6.28
200K	1.07	1.88	13.19
300K	1.62	2.85	20.56
400K	2.13	26.49	54.84
500K	2.73	97.93	678.44

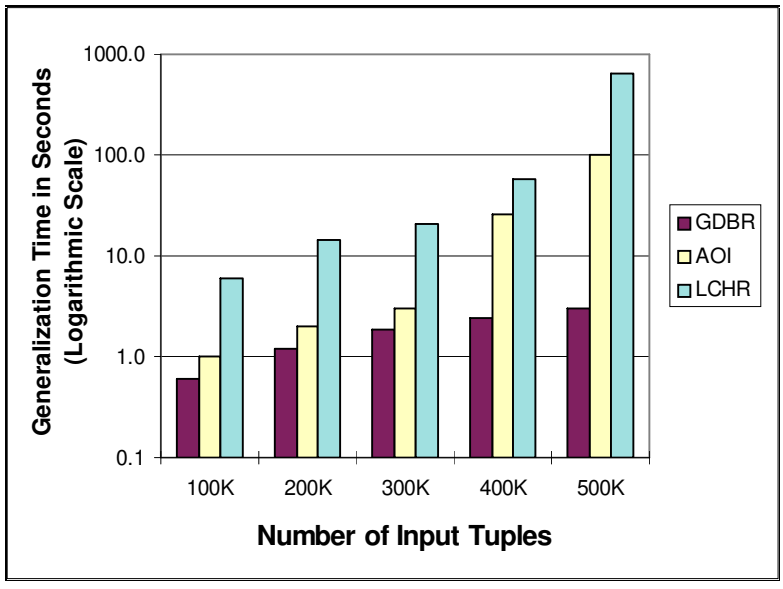

Figure 2. Test 1 Time Results for Generalization of Three Attributes, Varying Input Size

Table 2. Test 2 timing results

	GDBR	*AOI*	*LCHR*
100K	0.59	0.99	6.09
200K	1.18	2.00	14.19
300K	1.84	3.03	20.75
400K	2.37	26.00	56.78
500K	3.01	98.70	645.38

dollar amount. The input relation was generalized without any summation of the dollar amounts. The numerical timing results are presented in Table 1, and a graph of these, shown using a logarithmic scale, in Figure 2. For all timing tables in this section, the algorithms are listed in the top row and the input size in the left hand column. Cells of the table represent generalization times in seconds.

We note first of all that LCHR takes substantially longer than the other two algorithms, taking from 12 to 248 times as long as GDBR and about 7 times as long as AOI. LCHR was slower primarily due to sorting the input relation, which we timed to take approximately 95% of its overall time. Implementationally, the tuples were stored in array of pointers and the C library function *qsort* was used to sort this array. While this function is a general purpose function and may not be as efficient as a sort written specifically for this task, it is still very efficient. Our efforts to implement a faster specific sort were unsuccessful. GDBR ranges from 2 to 36 times faster than AOI.

The times for AOI and LCHR increase in an approximately linear manner until 400,000 or more tuples are input. Then times increase non-linearly when the input sizes exceed memory limitations and disk swapping occurs. The amount of increase will vary somewhat depending on the relative RAM and disk speeds of a given system. This increase, however, emphasizes the primary disadvantage of the unbounded space requirements of AOI and LCHR when large input relations are generalized. On the other hand, GDBR uses a relatively small, constant amount of

Table 3. Test 3 timing results

	GDBR	*AOI*	*LCHR*
100K	0.36	0.51	4.38
200K	0.70	1.02	9.56
300K	1.05	1.54	14.84
400K	1.40	2.04	20.00
500K	1.75	2.53	25.91

memory, independent of input size. We would therefore expect to see the times for GDBR continue to increase in a linear fashion, regardless of input size.

Test 2 used the same input as Test 1 except that it summed the attribute that represents dollar amounts. The results, shown in Table 2, are very similar to Test 1 except that the summation operations added about a constant .06 seconds per 100,000 input tuples for GDBR and AOI. Since LCHR was timed only once for an approximate time, the observed variations in its times can be attributed to the inherent imprecision of one timing result.

Test 3 (Table 3) involved an input relation with two attributes, a product code and a dollar amount. Like Test 1, it did not sum dollar amounts. Since memory was not exceeded by either AOI or LCHR, their time increases are linear and are relatively proportional to those of GDBR. Under no memory limitations, AOI is approximately 1½ times slower than GDBR.

6.3 Tests varying attribute thresholds

The second set of tests were run on fixed sized input relations (250,000 tuples), and varied attribute thresholds. This set was designed to test especially AOI and GDBR since their times are related to the size of the prime relation. The number of tuples in the prime relation is in turn related to the attribute thresholds.

Two tests in this category, Tests 4 and 5, are sufficient to indicate general trends. The results of Test 4 are presented in Table 4 and Figure 3. The left hand column of Table 4 represents the attribute thresholds for the three attributes of the input relation.

The primary observation we draw from Test 4 is that times for GDBR remain relatively constant, LCHR increases only marginally, but AOI increases steadily as attribute thresholds increase. AOI's increase is due to the search of the prime relation when each tuple is inserted. As the attribute thresholds increase, the prime relation grows in size and takes longer to search for each insertion point. From the trends indicated in Table 4, we would expect that AOI's time

Table 4. Test 4 timing results

	GDBR	*AOI*	*LCHR*
1, 1, 1	1.33	1.82	16.06
2, 2, 2	1.35	1.94	16.44
4, 4, 4	1.35	2.37	16.41
8, 8, 8	1.35	2.80	17.09
8, 16, 8	1.35	3.74	17.41
8, 16, 16	1.36	4.68	17.25
8, 32, 16	1.35	7.74	17.41

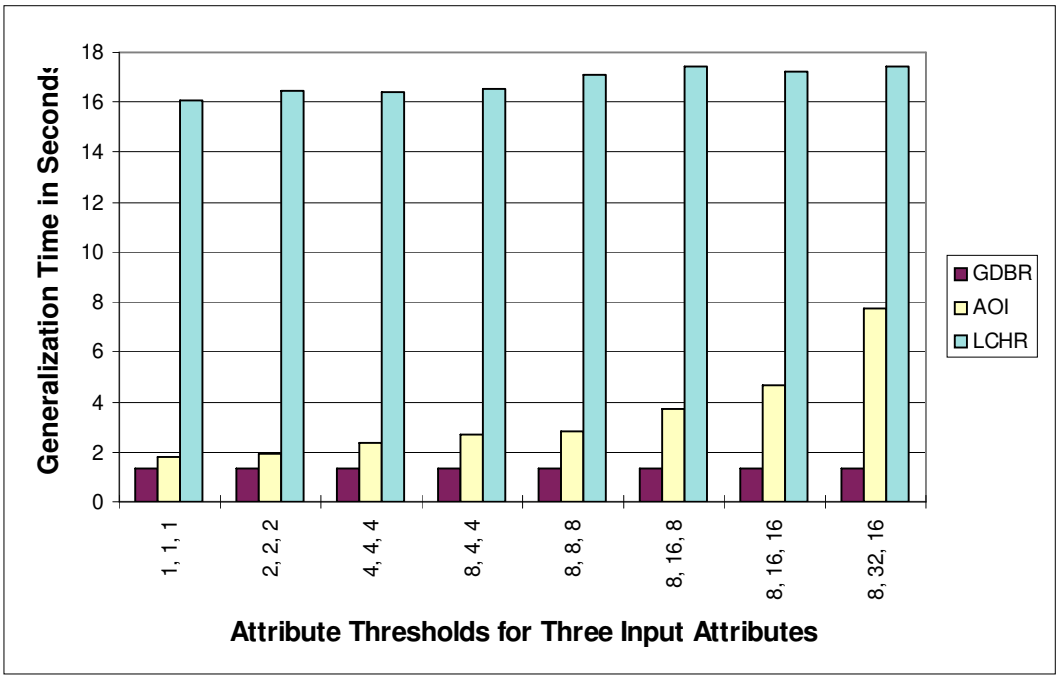

Figure 3. Test 4 Time Results for Generalization of Three Attributes, Varying Attribute Thresholds

would eventually exceed that of LCHR when attribute thresholds get large enough. In current practice, however, this may never happen since attribute thresholds are generally low.

Overall, GDBR ranges from 1.4 to 6 times as fast as AOI and 12 to 13 times as fast as LCHR in these tests. AOI ranges from 9 times faster than LCHR with low attribute thresholds down to only about twice as fast with higher thresholds.

The results of Test 5, which used an input relation with two attributes and did not sum any numerical attributes, are given in Table 5, and parallel the patterns observed in Test 4. Attribute thresholds, however, are somewhat smaller and therefore the increases in AOI are not as dramatic.

In summary, empirical tests on large input relations clearly show that GDBR consistently outperforms both AOI and LCHR. GDBR ranges from being only marginally faster to 36 times faster than AOI, and about 12 to 250 times faster than LCHR. The primary advantages of GDBR are derived from its optimality and its small, constant memory requirements. When large input relations are generalized, the O(n) memory requirements of AOI and LCHR cause memory to be exceeded and disk swapping to begin. This in turn causes a non-linear increase in time, degrading

Table 5. Test 5 timing results

	GDBR	*AOI*	*LCHR*
1, 1	0.88	1.18	11.56
2, 2	0.88	1.22	12.09
4, 4	0.89	1.26	12.19
8, 4	0.90	1.38	12.22
8, 8	0.88	1.40	12.31
8, 16	0.89	1.53	12.53

the performance of both AOI and LCHR. GDBR, however, increases only linearly with increased input size, and so its times remain very small. Increasing attribute thresholds also does not affect the times of GDBR significantly, while the times for AOI clearly increase as thresholds increase.

7. Algorithm Improvements

While the times for LCHR are substantially greater than both AOI and GDBR, the times for AOI are more acceptable when memory limitations are not encountered. The memory excesses of the AOI algorithm can be improved. As described in [10], AOI stores each individual input tuple in ungeneralized form before any generalization begins. The size of the input relation is therefore $O(n)$ for n input tuples. When the input relation grows large, memory is eventually exceeded and disk swapping begins. Many of these tuples, however, are duplicated even at the lowest level of generality. When generalization occurs, all duplicate tuples are combined and a count of the number of tuples contributing to the combined tuple is tracked by a *votes* variable. The AOI algorithm can be modified to store the input relation not as individual input tuples but as unique minimally generalized input tuples. When an input tuple is read from the database, the structure in which the input tuples are stored is searched for a match. If one is not found, the input tuple is inserted as a new tuple with a vote of 1. If a match is found, the vote of the matched tuple is incremented and any summary information is updated. In this way only one storage structure is needed for each unique tuple. The input relation could be stored as an ordered structure so that the search would not be too expensive.

The precise effect of these changes to the run time of AOI is unclear. However, in our experience, we have found that the number of distinct tuples read from a database is much less than the total number of tuples read. As such, AOI would be much less likely to exceed memory limitations and would therefore not suffer the increased time penalty seen in Table 1 and Figure 2.

This change to the AOI algorithm will increase its usefulness in comparison to GDBR. Since GDBR is an on-line algorithm and does not store the input relation, the input must be read again if overgeneralization occurs and the user desires a less general result. This database access can be very time consuming. AOI, however, stores the input in memory. If overgeneralization occurs, the input can simply be regeneralized without another access to the database.

Currently, GDBR and AOI have been tested on input relations with relatively few attributes, usually only two or three. This is primarily because the summaries produced by attribute-oriented generalization become hard to understand when more than three attributes are included in the generalized relation. However, the results of the generalization of more attributes may be useful as input to other machine learning methods, in which case generalizing more attributes may be advantageous. More research is needed to determine the usefulness of the generalization of more attributes and the affect that more attributes would have on the methods presented in this paper.

8. Conclusion

We have noted that the GDBR algorithm is $O(n)$ and as such is optimal. Its space requirements are also constant and very modest at $O(p)$ where p is the size of the output prime relation. These two factors combine to enhance its performance over a wide variety of input conditions. We empirically demonstrated this on relatively large input sets drawn from commercial databases, varying both the attribute thresholds and the input relation size to ensure the validity of the results. While GDBR is fast, however, modifications to the AOI algorithm will cause to have greater flexibility than GDBR, though the effect of these modification on

generalization time may be detrimental.

As fast and memory efficient algorithms, GDBR and an improved AOI will greatly enhance the potential for automated knowledge discovery from existing, large commercial databases. The algorithms provide a suitable basis for designing a software tool for knowledge discovery. Where a number of concept hierarchies exist for a given database, we foresee creating processes which explore the various possible relationships in the database in an automated fashion. The faster the algorithm runs and the more memory efficient it is, the more thoroughly we can explore the possibilities available.

References:

[1] Y. Cai, A Tutorial on the DBLEARN System, School of Computing Science, Simon Fraser University, March, 1990.

[2] Y. Cai, N. Cercone and J. Han, Attribute-Oriented Induction in Relational Databases, in: G. Piatetsky-Shapiro and W. J. Frawley, eds., *Knowledge Discovery in Databases*, AAAI/MIT Press, Menlo Park, CA, 1991, 213-228.

[3] Y. Cai, N. Cercone and J. Han, Learning Characteristic Rules from Relational Databases, *Proceedings of International Symposium of Computational Intelligence '89*, Milano, Italy, September, 1989.

[4] C. L. Carter and H. J. Hamilton, A Fast, On-line Generalization Algorithm for Knowledge Discovery, *Applied Math. Letters,* Accepted.

[5] C. L. Carter and H. J. Hamilton, GDBR: An Optimal Relation Generalization Algorithm for Knowledge Discovery from Databases, *Theoretical Computer Science,* Submitted November, 1994.

[6] C. L. Carter and H. J. Hamilton, Performance Evaluation of Attribute-Oriented Algorithms for Knowledge Discovery from Databases, in *Proceedings of the Seventh IEEE International Conference on Tools with Artificial Intelligence (ICTAI'95),* Washington, DC, November, 1995. Accpted July, 1995.

[7] C. L. Carter and H. J. Hamilton, Performance Improvement in the Implementation of DBLEARN, Tech. Report CS-94-05, Dept. of Computer Science, University of Regina, Regina, SK, January, 1994.

[8] C. L. Carter and H. J. Hamilton, The Software Architecture of DBLEARN, Tech. Report CS-94-04, Dept. of Computer Science, University of Regina, Regina, SK, January, 1994.

[9] W. Frawley, G. Piatetsky-Shapiro and C. Metheus, Knowledge Discovery in Databases: An Overview, *AI Magazine*, *Vol.13, No. 3*, 1992, 57-70.

[10] J. Han, Towards Efficient Induction Mechanism in Database Systems, *Theoretical Computer Science* (Special Issue on Formal Methods in Databases and Software Engineering), October 1994.

[11] X.-H. Hu, Object Aggregation and Cluster Identification: A Knowledge Discovery Approach, *Applied Math Letters*, Vol. 7, No. 4, 1994, 29-34.