GDBR: An Optimal Relation Generalization Algorithm for Knowledge Discovery from Databases

Colin L. Carter and Howard J. Hamilton

Department of Computer Science, University of Regina, Regina, SK, Canada, S4S 0A2

Abstract

We present GDBR, **G**eneralize **D**ata**B**ase **R**elation, an optimal, on-line O(*n*) algorithm for database relation generalization using concept hierarchies. The algorithm is a variant of attribute-oriented induction which generalizes database relations in either O(*n* log *n*) or O(*np*) time, where *n* is the number of input tuples and *p* is the number of tuples in the output relation. We augment the concept hierarchy structures to provide inherent generalization, define an encounter order on input concepts, progressively generalize input concepts as the need arises, and structure the final generalized relation in such a way to allow tuples to be inserted in one step.

1. Introduction

Knowledge discovery from databases (KDD) is "the nontrivial extraction of implicit, previously unknown, and potentially useful information from data[bases]" ([5], p.3). The motivation for KDD stems from the information explosion, or more precisely, the data explosion that our computer oriented society has been experiencing for some time. Tremendous amounts of specific, unanalyzed and therefore largely under-utilized data abound in computer databases worldwide. KDD encompasses a number of automated methods whereby useful information is mined from this rich source of potential information. We present in this paper GDBR (**G**eneralize **D**ata**B**ase **R**elation), an optimal, on-line algorithm that is an enhancement of a well known KDD method.

One general method used in KDD is that of *generalization*, the replacement of data values or concepts with higher level concepts which also include other related lower level concepts in their scope. Thus, the

concepts of *Alberta* or *North Dakota* might be generalized to *Prairies*, a concept which summarizes them and a few other Canadian provinces and American states. In reference to relational databases, generalization operates on a data set extracted by a database query initiated by a user. We call this data set an input *relation* in a loose sense of the term in that it may contain duplicate tuples. The generalization of a relation, therefore, is the replacement of all attribute (column) values found in the data with more general concepts of an equivalent level of generality and the summarization and removal of duplicate tuples (rows). The goal of the relation generalization is to produce a small relation that accurately summarizes the data in the original relation to such a degree that humans can recognize, understand and use the information therein. Relation generalization is a task to which computer automation is ideally suited, especially since relations frequently have hundreds of thousands of tuples.

Attribute-oriented induction is a relatively efficient generalization technique which, given a relation retrieved from a relational database, generalizes its contents on an attribute by attribute basis [2]. It is attribute-oriented in the sense that individual attribute values of a tuple are generalized independently of the values of other attributes in that tuple. Thus it avoids the exponential complexity that results from having to generalize an attribute value while taking into account all combinations of possible values of the tuple's other attributes. The relation generalization portion of attribute-oriented generalization runs in O(*np*) time where *n* is the number of input tuples and *p* is the size of the final generalized relation [6]. In this paper, we use the term *attribute-oriented induction* in a very restricted sense, specifically referring to a previous algorithm described in [6]. The term has been previously used in a broader sense in [2].

A very closely related algorithm, LCHR (**L**earning **CH**aracteristic **R**ules), accomplishes exactly 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]. Although both these algorithms are relatively efficient, they are not optimal.

GDBR is an enhancement of attribute-oriented induction which accomplishes the same result as attribute-oriented induction and LCHR but runs in O(*n*) time. We claim that this is optimal. Section 2 describes

relevant portions of attribute-oriented induction and LCHR as previously published. Section 3 describes GDBR and the basic data structures and methods the algorithm uses. Section 4 presents the algorithm more formally. Section 5 presents a summary of the complexity analysis of attribute-oriented induction, LCHR and GDBR, and Section 6 concludes the paper.

2. Overview of Attribute-Oriented Induction

Attribute-oriented induction takes as input a relation retrieved from a database and generalizes the data guided by a set of user defined or automatically generated concept hierarchies. A *concept hierarchy* is a tree of concepts with leaf nodes corresponding to the actual data values which may be found in the database and higher level nodes being more general concepts created by combining groups of lower level concepts under unique names. An example of a concept hierarchy for a PROVINCE attribute is given in Figure 1. After retrieving the task relevant data from the database, the first step of generalization is to convert the data values to matching leaf concepts from the appropriate concept hierarchy. This can be accomplished by storing all leaf concepts in a list or table and either binary searching the list or hashing values into the table. This initial conversion, however, is not the focus of our algorithm. This paper, like [6], is primarily concerned with the generalization algorithm itself which takes a relation from ungeneralized concepts to appropriately general concepts.

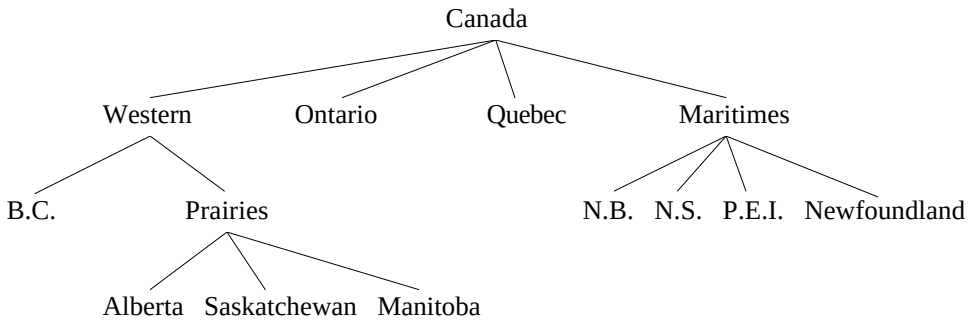

Figure 1: A concept hierarchy for the attribute PROVINCE

The generalization process is limited by a set of user defined *attribute thresholds* which specify the maximum number of distinct attribute values that may exist for each attribute of the final 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, greatly reducing the size of the input relation. The result is called the *prime relation*.

Attribute-oriented induction and LCHR are the same in the first stage of generalization, called *PreGen* in [6]. 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 of sufficient generality.

Attribute-oriented induction and LCHR handle the replacement of ungeneralized concepts and removal of duplicates somewhat differently. LCHR loops through the input relation replacing all data values with generalized concepts, then sorts the resulting relation and removes duplicates with one final pass. Attribute-oriented induction loops through the input relation on a tuple by tuple basis replacing 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 compared with every tuple 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. The final result is a small relation of unique tuples, each of which summarizes the number of tuples that it represents from the input relation.

3. GDBR Overview

GDBR incorporates several enhancements to the data structures and methods of attribute-oriented

induction 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, it defines an encounter order on the concepts in the input relation. 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 the data is read in. While this may require the prime relation to be reorganized a few times, it avoids both the need to sort a generalized relation of size *n*, as in LCHR, and the need to perform *n* linear searches on the prime relation to insert a tuple as in attribute-oriented induction.

3.1 Augmented Concept Hierarchies

For each concept in a concept hierarchy, we define a variable in which a pointer to an array of concepts is stored. When each hierarchy is constructed, we traverse it and provide each node with a *path array*, an array of pointers representing the node's path to the root of the tree. Sibling leaf nodes share a single array since the path to the root from each is the same. In addition, each parent node shares a portion of one of its leaf descendant's arrays, since a parent's path is a subpath of any of its descendants' paths. In this way, the arrays need only to be allocated and constructed for each group of leaf siblings and all non-leaf nodes share these in some way. An augmented tree structure is illustrated in Figure 2. Only one pointer for each group of leaf siblings is shown, though a pointer from each leaf node exists.

In addition to the path arrays, each node is assigned a variable *distance_to_max* which is the difference between the depth of the node and the depth of the deepest leaf node. In Figure 2, 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, and that of Newfoundland is 1.

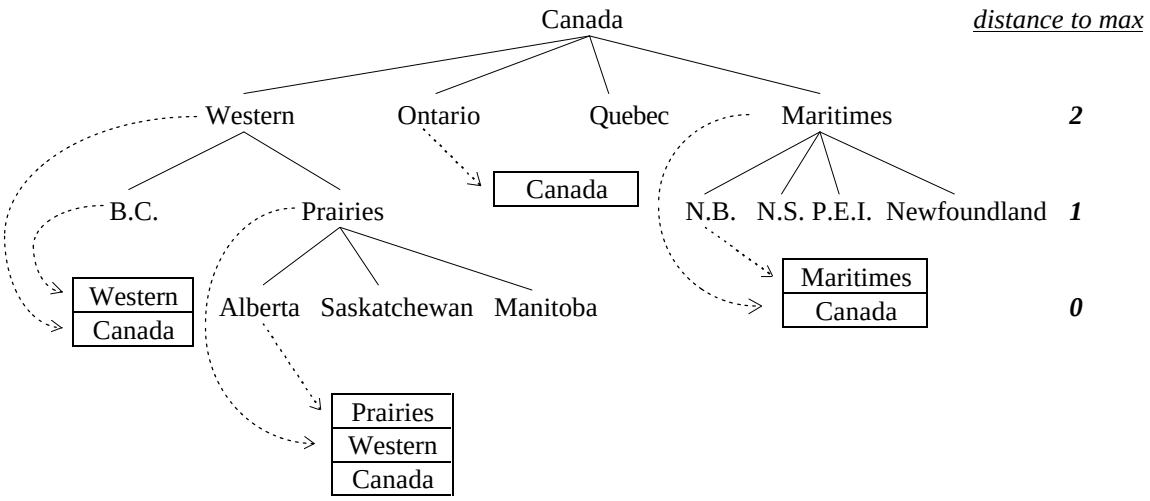

Figure 2. An augmented concept hierarchy for the attribute PROVINCE.

For each attribute in the input relation, the algorithm keeps track of a *generalization level*, an integer representing how many times this attribute has been generalized. The generalization level of each attribute 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, in Figure 2, 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 values are read from the database, they are converted to concepts and stored as such. As the input relation is generalized, the acts of incrementing the generalization level of an attribute and retrieving concept values by the *get_generalized_concept* function inherently generalizes the values already stored without having to replace the concepts themselves. This saves the algorithm from having to scan the input relation a second time to replace leaf values with generalized concepts as both LCHR and attribute-oriented induction do.

3.2 Encounter Ordering

To avoid sorting the generalized relation, we define an encounter ordering on the input tuples. 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 is converted to a concept and the ordinal of that concept is examined. If its value is 0, the *distinct_value_count* variable for that attribute is incremented and the resulting value is assigned to the concept as its ordinal. In essence, therefore, we both define an order on the input data in terms of a concept's first encounter and keep track of how many distinct values have been encountered.

3.3 Progressive Generalization

As tuples are read 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 newly encountered concept 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.

(a)	(b)	(c)	(d)	(e)
Alberta	1. Alberta	1. Prairies	1. Western	1. Western
Saskatchewan	2. Saskatchewan	2. Newfoundland	2. Newfoundland	2. Maritimes
Newfoundland	3. Newfoundland		3. N.B.	
Manitoba				
N.B.				
P.E.I.				

Figure 3: Example of Progressive Generalization

Given the concept hierarchy in Figure 2 and an attribute threshold of 3, Figure 3 illustrates the process of

progressive generalization.

Suppose that the PROVINCE attribute of the input relation initially contains the values shown in Figure 3(a). The generalization level for this attribute is initially set to 0. When generalization begins, Alberta is the first distinct value encountered. The *distinct_value_count* is incremented to 1 and Alberta is assigned this ordinal and stored in the *distinct_values* array. In turn, Saskatchewan is assigned 2 and Newfoundland 3, and both are also stored. The distinct values array now contains the elements in Figure 3(b). When Manitoba is encountered, the *distinct_value_count* is incremented to 4 and the attribute threshold is exceeded. The generalization index is incremented to 1 and the concepts in the distinct value array are scanned for new concepts at that level of generalization. The *get_generalized_concept* access function for Alberta will retrieve the Prairies concept which replaces Alberta and inherits Alberta's ordinal of 1. The access function for Saskatchewan also returns the Prairies concept which has already been encountered under Alberta. Saskatchewan is therefore removed from the *distinct_values* array and the *distinct_value_count* is decremented. Since the generalization level does not exceed the *distance_to_max* field of Newfoundland, the access function returns the same concept. Since Saskatchewan was removed from the distinct values array, however, Newfoundland's ordinal is changed to 2 and it is moved up in the *distinct_values* array. The newly encountered node of Manitoba is included under the Prairies node and so is no longer a candidate for addition to the *distinct_values* array. As shown in Figure 3(c), the number of distinct attribute values is now 2, which falls within the attribute threshold. N.B. is then read from the input, assigned the ordinal 3 and placed in the *distinct_values* array as shown in Figure 3(d). Finally, when P.E.I. is read, the attribute threshold is again exceeded, and more generalization occurs, giving the table shown in Figure 3(e).

3.4 Duplicate Elimination

To efficiently handle duplicate tuples, we construct the prime relation in the form of an *a* dimensional array where *a* is the number of attributes in the input relation. The size of each dimension is determined by the attribute threshold for the matching 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, though not all, of the contents of the prime relation to reflect the changes of some concepts' indices. Since the number of attributes and the attribute thresholds are generally very small, the size of the prime relation is much smaller than the input relation. The prime relation also accurately reflects the data read in immediately after a tuple has been inserted and so upon reading the last tuple, processing is complete. No further scanning, sorting or reorganizing is necessary.

4. GDBR Algorithm

In this section, we present the GDBR algorithm, a preliminary version of which appeared in [4]. We assume that a discovery task has been defined and the database initialized to retrieve relevant input data. We also assume that the tuple arrives from the database retrieval process with its attribute values already converted into leaf concepts from the appropriate concept 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.

Algorithm GDBR: generalize_database_relation - 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_to_attribute_threshold (
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;
    generalized_relation : Relation;

    { dynamically allocate distinct_values array and prime relation }
    distinct_values := allocate_distinct_values_array( attr_count, attr_threshold );
    generalized_relation := allocate_new_relation( attr_count, attr_threshold );
    generalized_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( generalized_relation, i );
                else
                    concept.ordinal := distinct_value_counts[i];
                    distinct_values[concept.ordinal ] := concept;
                end  { if }
            end { if }
        end { for }
        insert_tuple( generalized_relation, tuple );
    end { while }
    return generalized_relation;
end { generalize_to_attribute_threshold }
```

The *allocate_distinct_values_array* and *allocate_new_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 these values. The *get_next_tuple* procedure retrieves a tuple from the database and converts it to a leaf concept from the appropriate concept hierarchy. 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 and 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. Complexity Analyses

In this section, we present the analyses of attribute-oriented induction, LCHR and GDBR. The attribute-oriented analysis is summarized from [6] with some notational adjustments.

5.1 Analysis of Attribute-Oriented Generalization

Theorem 5.1.1 (Han, [6]): *The worst-case time complexity of [attribute-oriented induction as described] 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:

We use the iteration of a loop structure in each algorithm as a unit measure of work. The initial scanning of the input relation *R* to compile statistics about the number of distinct attribute values takes exactly *an* iterations since *R* has *n* tuples, each with *a* attributes and is scanned exactly once.

The statistics gathered in the initial scan are then used in conjunction with the relevant concept hierarchies. Each hierarchy is ascended until the lowest level of generality is attained that causes the number of distinct attribute values encountered to fall within the bounds of the attribute threshold. Suppose that there are a maximum of *v* distinct values for each attribute, *d* levels of each concept hierarchy to be ascended and a constant cost of *c* for each level ascended. The cost of the generalization of the attribute values encountered in *R* would be *avdc*. For large numbers of tuples in the input relation, however, all these values are extremely small, and therefore can be ignored in the overall order of the algorithm.

For attribute-oriented induction, *R* is scanned to replace the ungeneralized values with the generalized

values and then the generalized tuples are immediately inserted into the prime relation *P*. Though the basic loop of this algorithm will only run *n* times, the insertion of tuples into *P* will take an imbedded loop that performs a linear search of *P* to find a matching tuple. If a match is found, statistics are updated in that cell. If not, the new tuple is inserted at the end of *P*. The worst case situation of this step would therefore take *anp* iterations, giving a total for the algorithm of $a(n + np)$. Again, *a* is insignificant in comparison to *n* or *p* and so may be ignored, giving a final result of $O(np)$. ■

Theorem 5.1.2: *The worst-case time complexity of LCHR is O(n log n) where n is the number of tuples in the input relation.*

Proof Sketch:

The first two steps of LCHR are the same as for attribute-oriented induction as described above and are accomplished in *an* iterations. The difference comes in the replacement of ungeneralized concepts with generalized concepts and the elimination of duplicate items. LCHR, scans the input relation once to replace the ungeneralized attributes in *R* with their generalized counterparts in another *an* iterations. It then sorts the relation in $an \log an$ time and removes duplicates with another complete scan in *an* steps. This step therefore takes $a(2n + n \log an)$ and the total work of the algorithm would be $a(3n + n \log an)$. If $a << n$, the algorithm is therefore $O(n \log n)$. ■

Where *p* is greater than log *n*, LCHR would be more efficient for large input. Otherwise, attribute-oriented induction would be better.

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^a. If a distinct attribute threshold, t_i, is used for each attribute a_i, the worst case size of the prime relation will be:

$$p = \prod_{i=1}^{a} t_i .$$

We restrict the value of t_i to the total number of leaf nodes in the ith concept hierarchy. For discrete valued attributes, this represents the number of distinct attribute values that may be encountered in the database data. For continuous valued attributes such as integers or times, this represents the number of ranges of values that the user has 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.

As far as space required by the algorithms, there are two possibilities to consider. If the input relation is read only once, the whole relation will need to be stored in memory between the statistics calculating step and the generalization step. If memory is limited, the input relation could be read twice, the first time from the database and the second from either the database or a temporary file. Our experience 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 inputs. To store the input relation, whether in main memory or on disk will be O(an) for the input relation when storing pointers to distinct attribute values. The prime relation will need a maximum of ap storage cells. 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(an) + O(ap), or O(n) + O(p) since a is small in relation to n.

5.2 Analysis of GDBR

Theorem 5.2.1: *The worst-case time complexity for GDBR is O(n) where n is the number of tuples in the input relation, and where the number of attributes in the input relation, the attribute thresholds for each attribute, and the depths of all concept hierarchies are small.*

Proof Sketch:

Again we define n as the number of input tuples, a 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 *a* iterations to loop through the attributes in each tuple. All operations in these loops are bounded by a small constant except for the *generalize_concepts* and *generalize_relation* procedures.

There are *a 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 *cadt.* Again, these values are extremely small in comparison to *n* and so may be ignored.

The prime relation of size *p* will be adjusted a maximum of *ad* times with a small constant of *c* as a measure of work to adjust each concept. An upper bound for the *generalize_relation* function can therefore be initially set at *cadp.* The total work for the algorithm, therefore, *an* + *cadp*. If *c*, *a* and *d* are small in relation to *n*, GDBR is O(*n*) + O(*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*). ■

To quantify these amounts, we illustrate typical values from our experience on public domain data, data supplied by corporate sponsors and values taken from previous publications on LCHR and attribute-oriented induction. We have found that we typically run generalization tasks on two or at most three attributes since the results of more than this are hard to understand. In addition, attribute thresholds tend to be in the 2 to 15 range. The depth of the concept hierarchies we have used rarely exceeds 4 or 5. Using the maximums of these figures, we find that *adp* equals $3(5)(15^3) = 50{,}625$. While this figure may seem large, we have done learning tasks on data with up to 600,000 tuples and larger tasks are possible. In this case, the 50,625 units of work done in the reorganization stage is dwarfed by the size of *n*. The original attribute-oriented algorithm using the first method would take *n* log *n*, or 11,516,761 units of work. The second method of attribute-oriented generalization would take *np*, or $600{,}000(15^3) = 2{,}025{,}000{,}000$ units in worst case. On average, however, all these figures would be lower.

In addition, GDBR will never take our worst case estimate. This is because the higher the attribute threshold, the larger the prime relation will be, but it will be generalized fewer times since the concept hierarchies will not need to be ascended fully. In addition, when the prime relation is reorganized, only those attributes whose ordinal value have changed will need to be adjusted. The first concept encountered will always keep the ordinal of one and never need moved. The algorithm is also easily modifiable to keep ordinal values in place when possible and fill in gaps in the ordinal progression after the prime relation is minimally reorganized. These factors combine to reduce the *adp* amount in our analysis. For large *n*, the assumption that *adp* << n is justified.

Space requirements for GDBR are also very modest. Each tuple is examined only once and immediately inserted into the prime relation. After this, it is no longer needed and so need not be stored. This saves the O(*an*) space requirement of both LCHR and attribute-oriented induction. Unlike LCHR and attribute-oriented induction, 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, however, the prime relation is very small in comparison to *n*. Other storage requirements by the algorithm are also insignificant in comparison to *n*, the two most notable being an array of length *t* for *distinct_attribute_counts*, and *a distinct_values* arrays with a total of *at* cells which keep track of the encountered concepts at the current level of generalization. The total space needed therefore is O(*p*) which is much less than O(*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 exponentially. Since GDBR allocates this structure to be maximum size, this could put GDBR at a space disadvantage to LCHR or attribute-oriented induction 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.

We have noted that each tuple can be examined, generalized if necessary, stored in the prime relation and discarded after this. We also previously noted in Section 3.4 that the algorithm accurately reflects the current generalized state of the input at any stage. In this regard, GDBR is an *on-line* algorithm.

5.3 Proof of GDBR's Optimality

Theorem 5.3.1: *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 *n*th tuple. For the *n*th 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 *n*th 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. ■

6. Conclusion

We have noted that the GDBR algorithm is O(*n*) where *n* is the number of input tuples and as such is optimal. It's space requirements are also very modest at O(*p*) where *p* is the size of the output prime relation. Finally it is also an on-line algorithm. As such it greatly enhances the potential for automated knowledge discovery from large databases. 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, the more thoroughly we can explore the possibilities available. In addition, where multiple concept hierarchies exist for the same attribute, an algorithm which runs at a very efficient rate will be able to generalize a relation according to these different concept

hierarchies in parallel.

7. Acknowledgments

We wish to thank Jiawei Han for providing us with a prototype implementation of DBLearn [1], a KDD program written by Y. Cai, J. Han, L. Liu, and Y. Huang implementing attribute-oriented induction. We also thank Nick Cercone, Sharon Hamilton, Jiawei Han, and Larry Saxton for valuable discussions. We gratefully acknowledge the support of the Institute for Robotics and Intelligent Systems, the Networks of Centres of Excellence Program of the Government of Canada, the Natural Sciences and Engineering Research Council (NSERC), and the participation of PRECARN Associates, Inc. We are grateful to NSERC and Rogers Cable for providing test databases.

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, *Appl. Math Lett.* Submitted.

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

[6] J. Han, Towards Efficient Induction Mechanism in Database Systems, *Theoret. Comput. Sci.* (Special Issue on Formal Methods in Databases and Software Engineering), (October 1994).

Accepted.

[7] J. Han, Y. Cai and N. Cercone, Knowledge Discovery in Databases: An Attribute-Oriented Approach, *Proceedings of the 18th VLDB Conference*, (Vancouver, British Columbia, 1992) 547-559.

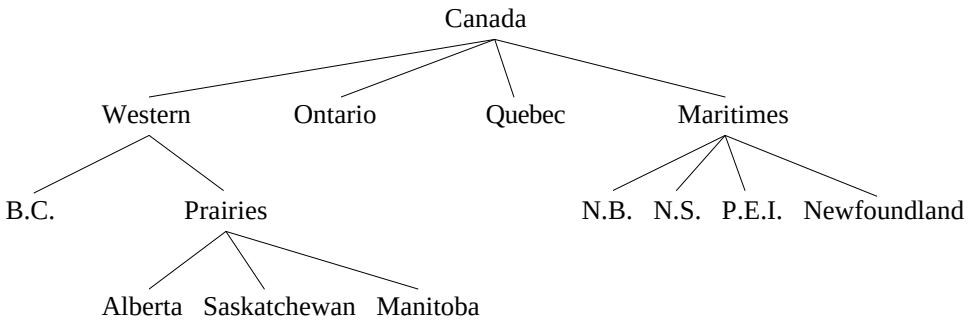

Figure 1: A concept hierarchy for the attribute PROVINCE

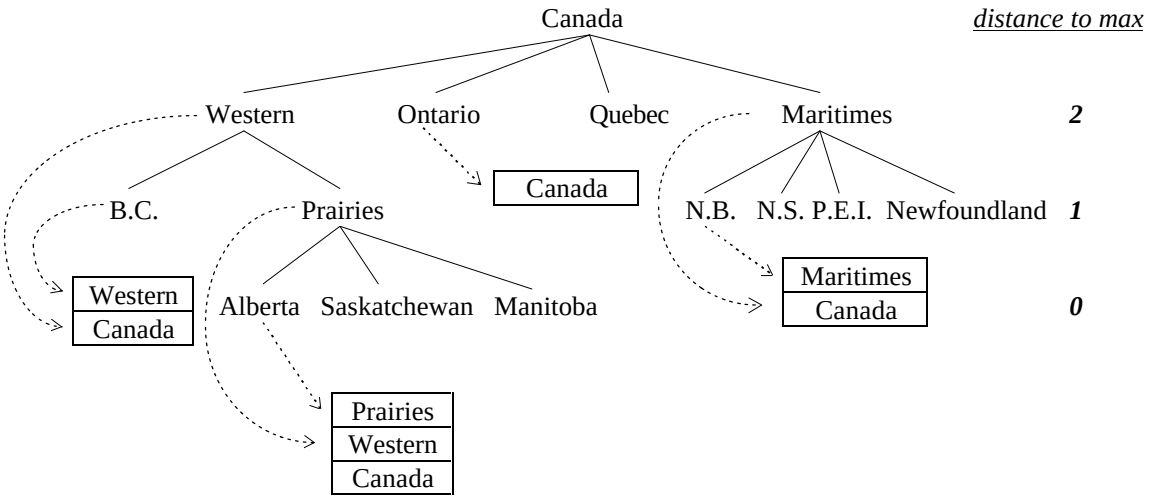

Figure 2. An augmented concept hierarchy for the attribute PROVINCE.