A Comparison of Four Tests for Attribute Dependency in the LEM and LERS Systems for Learning from Examples

Jerzy W. Grzymala-Busse and Sachin Mithal
Department of Computer Science
University of Kansas
Lawrence, KS 66045

Abstract. The paper discusses two programs for learning from examples, LEM (Learning from Examples Module) and LERS (Learning from Examples based on Rough Sets). A few versions of both programs are implemented in Franz Lisp and are running on VAX 11/780. Both programs' main task is to automate knowledge acquisition for expert systems. Hence, they produce rules in the minimal discriminant form. The main problem addressed in the paper is the selection of the best mechanism for determining coverings, the minimal sets of relevant attributes. Four different methods, based on indiscernibility relation, partition, characteristic set and lower boundary are compared. Both theoretical analysis and experimental results of multiple running of many sets of examples, with variable number of examples and with variable number of attributes are taken into account. As a result the partition method is determined to be the most efficient way to compute coverings.

1. Introduction.

The paper discusses learning from examples, the most popular form of similarity-based learning [8, 9]. The idea of learning simple rules with only relevant attributes involved is well known [1 - 7, 10]. The paper addresses the issue of choice of the best mechanism for determining the minimal sets of relevant attributes.

The two systems for learning from examples, LEM and LERS, were designed to automate knowledge acquisition for expert systems. Hence, both systems produce rules in the minimal discriminant form [9]. The first system, LEM (Learning from Examples Module), learns rules from the set of consistent examples, while the second system, LERS (Learning from Examples based on Rough Sets), is able to learn rules from inconsistent examples. Both systems are implemented, in a few different versions, in Franz Lisp and both are running on VAX11/780. In the process of improving LEM and LERS, special attention was paid to the execution time of programs as well as the required space. Earlier versions of LEM and LERS used *characteristic sets* as a tool for finding minimal sets of relevant attributes. The newest versions employ another mechanism, *lower boundaries*, thus accomplishing much greater efficiency. The above two and two additional mechanisms, called partition and indiscernibility relation methods, are described and compared in the paper.

2. Four Methods for Testing Attribute Dependency

The following definitions are used in the sequel. In the process of learning given are *examples* (in other works also called *objects* or *instances*). The set of all examples is called an *universe* and denoted U. Each example is described by n attributes, where n is a positive integer. For each attribute the number of attribute values is finite. For any two different attributes, their value sets are not necessarily mutually exclusive. An example is represented by a vector of the length n of pairs (attribute, attribute value). Two different examples may be represented by the same vector of (attribute, attribute value) pairs. A nonempty subset C of U is called a *concept* (to be learned). In other works the concept is also called a *class*. Members of C are called *positive examples*, while members of U – C are called *negative examples*. In the process of learning, a *representation* of the concept C is sought. In this paper, like in the most of other approaches to learning from examples, a set of if-then rules is such a representation.

Let Q denote the set of all attributes. It is clear from the previous assumptions that the number of elements in Q is n. Let P be a subset of Q. Let k be the number of elements in P. Two examples, x and y, represented by the same vector of length k of pairs (attribute, attribute value), where each attribute is a member of P, are called *P-indiscernible*, and denoted $x \sim_P y$. Obviously, $\sim_P$ is an equivalence relation on U. The equivalence relation $\sim_P$ induces a partition on U, denoted P*. Such a partition P* is the set of all equivalence classes of $\sim_P$, called *blocks*.

In many papers on machine learning from examples the importance of using relevant attributes in representation of the concept has been observed [1 - 7, 10]. Nevertheless, very few methods for selection relevant attributes were given [2, 3]. The problem of selection of relevant attributes is the main problem of this paper. First of all, the formal definition of relevant attribute is needed. The following definition of dependency is borrowed from

relational databases, where it is known under the name of functional dependency [11, p. 213]. Here, the name of functional dependency is abbreviated to dependency because it is the only kind of attribute dependency which is discussed.

Note that for a concept C, the set $C^* = \{C, U - C\}$ is a partition on U. Hence we may think about yet another attribute, different from all attributes from the set Q, and corresponding to the concept C. Such an attribute will be called a *concept attribute* and also denoted C. The concept and concept attribute may be easily distinguished by context in which they are used. The assumption is that the concept attribute is not a member of the attribute set Q. An indiscernibility relation, corresponding to the partition C^* on U will be denoted $\sim_C$. Therefore, a concept C *depends on* a subset P of the set Q of all attributes, denoted $P \rightarrow C$, if and only if

$$\sim_P \subseteq \sim_C .$$

Practically, concept C depends on a subset P of the set Q of all attributes if and only if the description of positive examples by attributes from P is sufficient to recognize the concept of C, i.e. the set of all positive examples is the union of some blocks of relation $\sim_P$ (and then, as a consequence, the set of all negative examples is also the union of some blocks of relation $\sim_P$.

Obviously, the most interesting case is when k, the size of the set P, is the smallest possible. Thus, a new definition is needed: a subset P of the set Q of all attributes is called a *covering* of the concept C if and only if C depends on P and P is minimal. This is equivalent to the following: P is a *covering* of C if and only if C depends on P and no proper subset P' of P exists such that C depends on P'. The notion of a covering of the concept C, for $Q^* = C^*$, is analogous to that of a key of relation scheme [11, p.217].

All algorithms of LEM and LERS are based on looking for coverings, since the task of these programs is finding the minimal discriminant descriptions. The problem is how to recognize the fact that concept C depends on a subset P of the set Q of all attributes. The first method, called an *indiscernibility relation method*, is described above. The second method, called a *partition method*, follows immediately from the first one and is based on the following condition: the concept C depends on a subset P of the set Q of all attributes if and only if

$$P^* \leq C^*,$$

where $C^* = \{C, U - C\}$. Partition P^* is smaller than or equal to partition C^* if and only if each block of P^* is a subset of either C or U – C. The next test is based on the notion of characteristic set. Let $U = \{x_1, x_2, ..., x_m\}$. Let P be a nonempty subset of the attribute set Q. A *characteristic set* P^c of P is the set of all pairs (x_i, x_j), where

(i) $i < j$,

(ii) x_i and x_j are members of the same block of partition P^* on U, and

(iii) $1 \leq i, j \leq m$.

A characteristic set P^c of P is a subset of the indiscerniblity relation $\sim_P$. Thus, the concept C depends on a subset P of the set Q of all attributes if and only if

$$P^c \subseteq C^c,$$

where C^c is defined the same way as P^c, except that $\sim_P$ should be substituted by $\sim_C$. In the first version of LEM the above condition was checked in a slightly different form, namely

$$\bigcap_{p \in P} (\{p\}^c - C^c) = \emptyset.$$

The last method is called *lower boundary method*. In order to introduce it the following definition is necessary. Let X be a subset of U and P be a subset of Q. Then the P-lower boundary of X, denoted $\underline{B}_P(X)$, is defined as

$$X - \{x \in U \mid [x]_P \subseteq X\},$$

where $[x]_P$ is a block of $\sim_P$ containing X. The set $\{x \in U \mid [x]_P \subseteq X\}$ is called a *P-lower approximation of X*. It is shown in [2] that the concept C depends on a subset P of the set Q of all attributes if and only if

$$\underline{B}_P(C) = \emptyset.$$

3. Algorithms

The following are algorithms to compute coverings using the above four methods.

3.1 General Algorithm for Computing Coverings

```
Start algorithm
   input C (the concept attribute);
   input Q (the set of attributes);
   initialize coverings to nil;
   read data from input file and insert them into property
        lists;
   pass the list of attributes to function Comb, which
        returns a list combs, the power set of U;
   for each element P of the list combs do
   begin
        if C depends on P {Check Dependency}
             then if P is minimal
                       then add P to the list coverings
   end {for}
   return the list coverings, which has coverings as its
        elements;
end algorithm.
```

3.2 Algorithms for Checking Dependencies

3.2.1 Algorithm 1 (Indiscernibility Relation Method)

```
Procedure Check Dependency Method1 (P, C)
{P is a set of attributes}
    {data from input file inserted into property lists}
    compute the indiscernibility relation associated with P;
    compute the indiscernibility relation associated with C;
    if the indiscernibility relation of P is a subset of
    indiscernibility relation of C,
    then return "C depends on P"
    else return "C does not depend on P"
end; {Check Dependency Method1}

Procedure Indiscernibility Relation (P, U)
{P is a set of attributes, U is the set of examples}
    for example1 := first example to last example do
      for example2 := first example to last example do
        begin
        for attribute := first of P to last of P do
          begin
          get the value of attribute associated with
          example1;
          get the value of attribute associated with
          example2;
          if the above two values are identical
          then go to next outer loop;
          end; {for attribute loop }
        if the above loop was successful
        then add the pair (example1, example2) to result
        else continue;
        end; {example2 loop}
    return the result, which is a list of pairs of examples,
which succeeded the matching;
end; {procedure Indiscernibility Relation}
```

3.2.2 Algorithm 2 (Partition Method)

```
Procedure Partition Method2 ( P, C)
{P is a set of attributes}
{data from input file inserted into property lists}
    compute the partition for P;
    compute the partition for C;
    If defined subset (partition P, partition C) is true
    then return "C depends on P"
    else return "C does not depend on P"
end; {Partition Method2}

Procedure Defined Subset (A, B)
{A and B are partitions on U}
    for each block of A do
        if there exists a superset block in B
        then return true
        else return false;
end; {procedure Defined Subset}

procedure Partition (P, U)
{P is a set of attributes, U is the set of examples}
    while not empty U do
      begin
      example1 := first of examples;
      for example2 := rest of examples do
        for attribute := first of P to last of P do
          begin
          get the value of attribute associated with
          example1;
          get the value of attribute associated with
          example2;
          if the above two values are identical
          then attribute := next attribute
          else break to next outer loop;
          end; {for attribute loop }
        if the above loop was successful
        then add example2 to the same block as example1
        else form another block with example2 in it;
        remove example2 from U
        end {of example2 loop}
      end; {of while loop}
    return a list, which contains blocks (simple sublists);
end; {procedure Partition}
```

3.2.3 Algorithm 3 (Characteristic Set Method)

```
Procedure Characteristic Set Method3 ( P, C)
{P is a set of attributes}
{data from input file inserted into property lists}
    compute the characteristic set of P;
    compute the characteristic set of C;
    if characteristic set P is a subset of characteristic set C)
    then return "C depends on P"
    else return "C does not depend on P"
end; {Characteristic Set Method3}

Procedure Characteristic Set (P, U)
{P is a set of attributes, U is the set of examples}
    for example1 := first example to last example do
      for example2 := next example to last example do
        begin
        for attribute := first of P to last of P do
          begin
          get the value of attribute associated with
          example1;
          get the value of attribute associated with
          example2;
          if the above two values are identical
          then attribute := next attribute
          else break to next outer loop;
          end; {for attribute loop }
        if the above loop was successful
        then add the pair (example1, example2) to result
        else continue;
        example2 := next example;
        end; {example2 loop}
    return the result, which is a list of pairs of examples,
which succeeded the matching;
end; {procedure Characteristic set}
```

3.2.4 Algorithm 4 (Lower Boundary Method)

```
Procedure Lower Boundary Method4 (P, C)
{P is a set of attributes}
```

```
{data from input file inserted into property lists}
    compute the partition of P;
    compute the partition of C;
    if X is P-definable for all blocks X of partition C*
    then return "C depends on P"
    else return "C does not depend on P"
end; {procedure Lower Boundary Method4}

procedure Partition (P, U)
{P is a set of attributes, U is the set of examples}
    while not empty examples do
      begin
      example1 := first of examples;
      for example2 := rest of examples do
        for attribute := first of Q to last of Q do
          begin
          get the value of attribute associated with
          example1;
          get the value of attribute associated with
          example2;
          if the above two values are identical
          then attribute := next attribute
          else go to next outer loop;
          end; {for attribute loop }
        if the above loop was successful
        then add example2 to the same block as example1
        else form another block with example2 in it;
        remove example2 from U
        end {of example2 loop}
      end; {of while loop}
    return a list, which contains blocks (simple sublists);
end; {procedure Partition}

Procedure P-definable (X, P)
    for block1 := first block of P to last block do
      begin
      find lower approximation (X, P);
      subtract lower approximation from X;
      append this to the result;
      end; {of for loop}
    return the result from above;
end; {Procedure P-definable}

Procedure Lower Approximation (X, P)
      if block1 is a subset of X
      then add block1 to the result
      else next block
    return result;
end; {procedure Lower Approximation}.
```

4. Time and Space Complexity Analysis

In this section time and space complexity analysis for all four methods is cited.

4.1. Time Complexity

Algorithm 1. Here for checking dependency $P \rightarrow C$, the indiscernibility relations are computed for both P and C. In each case, if there are m examples, then m^2 pairs are compared for equality of attributes. Since such a relation is computed for all n attributes, the total number of comparisons in this worst case is m^2n. After computing the indiscernibility relation for both P and C, the subset relationship is checked. This results in m^2 pairs being matched with m^2 pairs, hence m^4 comparisons are executed. This takes up the worst-case function to $m^4 + m^2n$. Similarly, the best-case scenario is when no pairs match each other. That results in $m^2 + m^2$ comparisons.

Algorithm 2. The worst-case is when each example lies in a different block. For this, the upper-bound will be $nm(m-1)/2$. Then, moving on to checking of the subset relation, the worst case is when each block of P is a subset of some block of R. Say there is p blocks of average size q. Here pq=m. The number of comparisons needed for checking the relation that each block of P is a subset of a block in R will need p^2q^2 comparisons. But since pq=m, $p^2q^2 = m^2$. Therefore, the worst-case will result in complexity of $m^2n + m^2$. In the best-case, all the examples are in the same block. Thus, the total number of comparisons is mn + m.

Algorithm 3. This method is similar to the first one, except that it needs fewer comparisons. The redundant pairs which were present in indiscernibility relation are automatically omitted. This results in $(m^2/2)n$ comparisons. For checking the inclusion relation $m^4/4$ comparisons are needed. This totals up to $m^4/4 + (m^2/2)n$ comparisons. The best-case is when no pairs match each other. As in method 1, this results in $m^2 + m^2$ comparisons, which occurs only if all examples are different blocks and hence do not match any other example.

Algorithm 4. In this method blocks are computed first. The worst-case is when each example lies in a different block. Then the lower boundary is computed by checking whether a block is a subset of C (another block). Similar to the subset checking in method 2, $p^2q^2 = m^2$ comparisons is necessary. Then subtraction may take another m^2 steps. The total is $m^2 + m^2 + m^2n$ comparisons. In the best-case, mn steps are required for the partition and m^2 for checking the inclusion relation and subtraction, which totals to $m^2 + mn$.

Table 1 presents time complexity for all four methods.

Table 1

	Time complexity (big oh)	
	Worst-case	Best-case
Algorithm 1	$m^4 + m^2n$	m^2
Algorithm 2	m^2n	mn
Algorithm 3	$m^4 + m^2n$	m^2
Algorithm 4	m^2n	mn

4.2 Space Complexity

The space functions, which values represent the number of memory locations, are determined below.

Algorithm 1. Here for checking dependency P → C, their indiscernibility relations are computed for both P and C. In the worst case, all $2m^2$ pairs are created.

Algorithm 2. In this method instead of pairs, blocks are generated. Their number is 2m.

Algorithm 3. This method, like Algorithm 1, also employs generation of pairs, in this case $2 \cdot m(m-1)/2 = m(m-1)$ pairs.

Algorithm 4. This method, like Algorithm 2, also needs 2m memory locations.

Table 2 presents space complexity for all four methods.

Table 2

	Space complexity (big oh)
Algorithm 1	m^2
Algorithm 2	m
Algorithm 3	m^2
Algorithm 4	m

These theoretical results are supported by the experimental results presented below. Though the space needed by the four programs during the actual runs has not been measured, the algorithm #1 was the first one to fail due to the space shortage. In fact, this one failed for as many as 15 examples and 5 attributes. The second worst was algorithm #3, which quit for 27 examples and 5 attributes. The other two were much more sturdy as far as the space is concerned and were doing well even for 300 examples and 5 attributes.

5. Experiments

Five families of example sets were constructed. For each of the four different algorithms a UNIX shell file was created. The file would invoke the LISP interpreter, get the LISP commands to be run from another file, and redirect the output from the LISP interpreter to an output file. The main reason for using the shell file is that the *time* command was used to clock the run time used by our process. The execution time is measured in seconds. There also is an input file which would load the relevant main program and invoke the LISP function to start computing the coverings. The concept C, an attribute set P, and the data file name are also provided by this file. The commands which otherwise are typed on the Franz-LISP interpreter level were stored in this input file.

5.1 Results for Variable Number of Examples and Constant Number of Attributes

Below three family of data are presented, all with variable number of examples and constant number of attributes. In the first two families of example sets five attributes were used, while the third family had four attributes. Tables 3 - 5 and Figures 1 - 3 present experimental results for the above three families of example sets. Dashes in the tables mean that the program did not even complete the run and quit due to the space limitation.

Table 3

	Execution time (in seconds)			
# of examples	Algm. #1	Algm.# 2	Algm.# 3	Algm.# 4
4	3.3	3.0	3.1	2.9
10	24.6	9.1	18.4	9.6
15	57.0	12.9	46.6	15.3
20	-	17.7	94.4	19.8
30	-	26.9	-	29.8
50	-	46.7	-	50.9
100	-	105.9	-	113.1
150	-	172.9	-	186.2
200	-	244.9	-	276.9
300	-	412.0	-	474.1

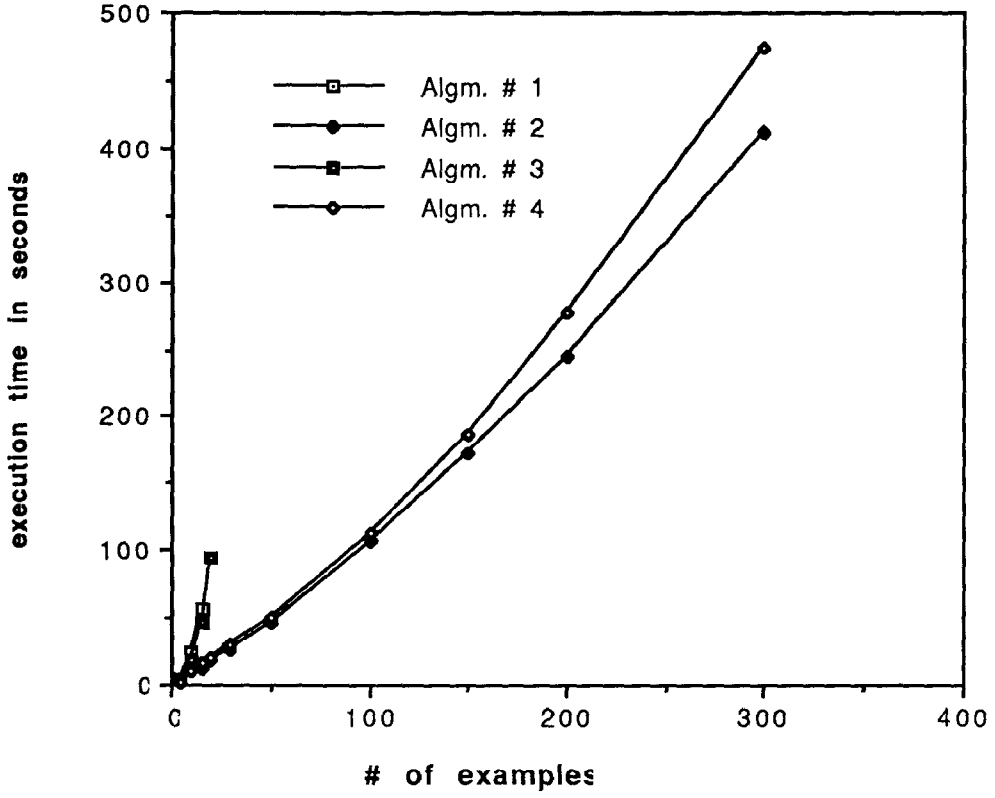

Figure 1. Family 1 of example sets, all with five attributes

Table 4

	Execution time (in seconds)			
# of examples	Algm. #1	Algm.# 2	Algm.# 3	Algm.# 4
2	2.0	1.9	2.1	2.1
4	2.7	2.3	2.4	2.4
6	4.8	3.0	3.7	3.2
8	9.9	4.6	7.3	4.9
10	22.8	8.5	17.1	9.0
12	32.7	10.2	26.4	11.1
15	52.5	12.0	44.9	13.4
19	-	16.1	80.0	18.0
23	-	19.5	131.0	21.3
27	-	22.3	-	25.7

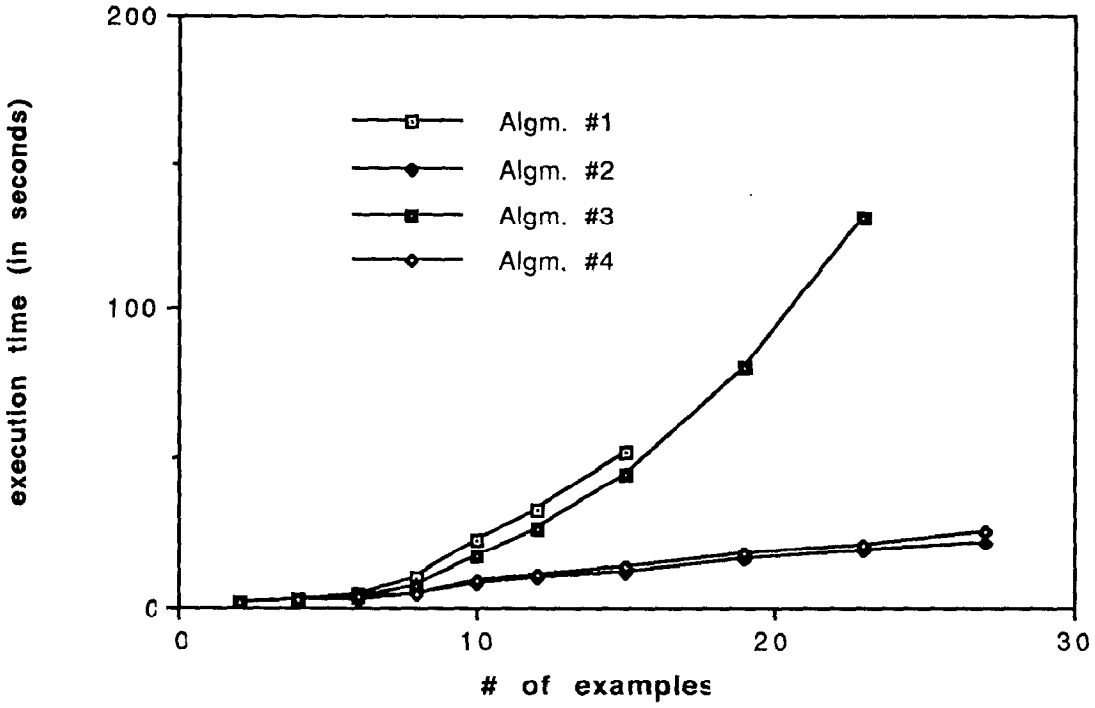

Figure 2. Family 2 of example sets, all with five attributes

Table 5

	Execution time (in seconds)			
# of examples	Algm. #1	Algm.# 2	Algm.# 3	Algm.# 4
5	5.8	3.2	4.4	3.4
10	22.0	6.0	20.3	6.8
15	46.5	9.4	54.0	10.3
20	-	12.3	119.8	14.0
25	-	16.2	-	17.7

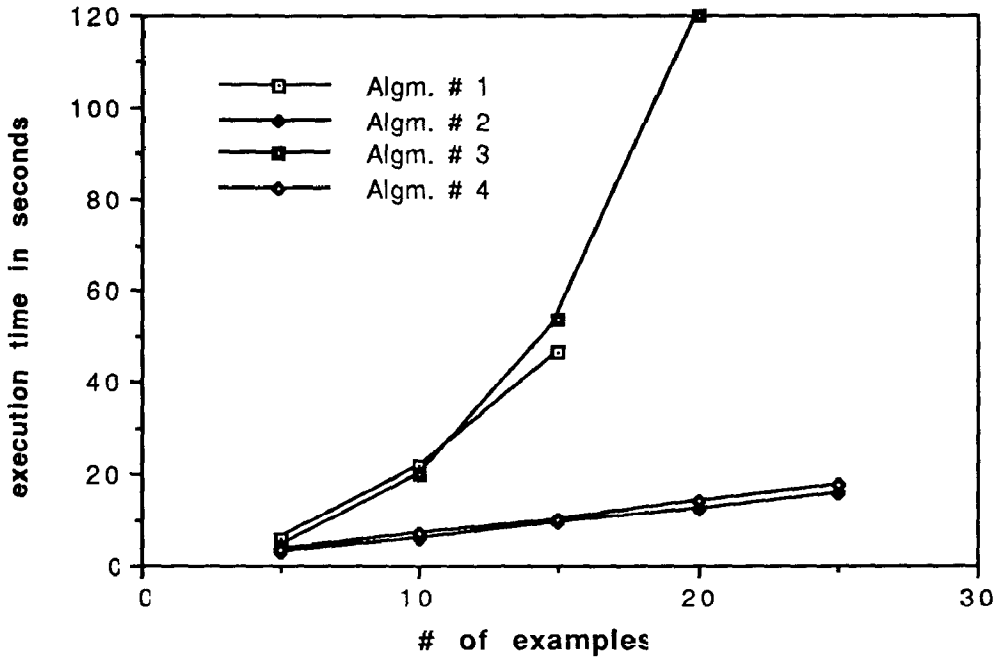

Figure 3. Family 3 of example sets, all with four attributes

5.2 Results for variable number of attributes and constant number of attributes

Below two families of data, all with variable number of attributes and with the same number of ten examples, are presented an Table 6 and 7 and Figures 4 and 5, respectively.

Table 6

	Execution time (in seconds)			
# of attributes	Algm. #1	Algm.# 2	Algm.# 3	Algm.# 4
3	8.9	5.7	5.8	6.1
5	20.7	11.5	13.0	13.6
7	38.6	22.7	26.1	26.5
8	58.0	33.9	39.2	39.2
10	120.1	85.8	93.6	93.2

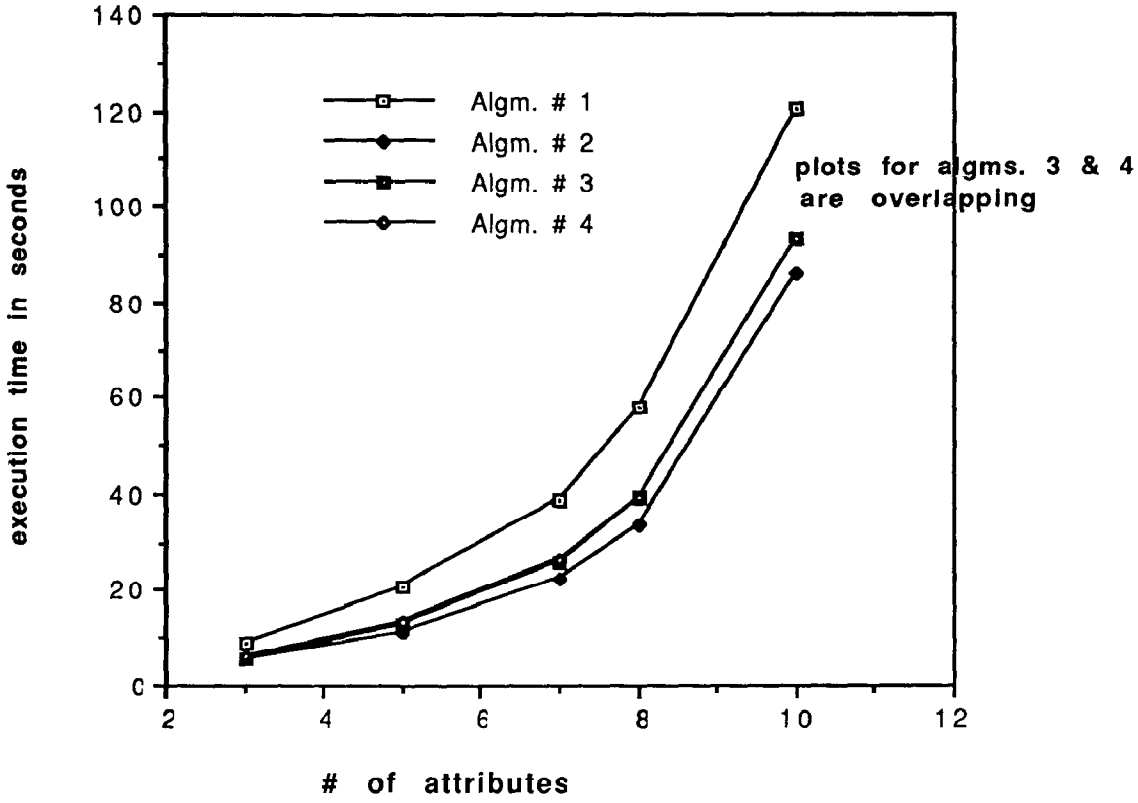

Figure 4. Family 4 of example sets, all with ten examples

Table 7

	Execution time (in seconds)			
# of attributes	Algm. #1	Algm.# 2	Algm.# 3	Algm.# 4
3	6.7	4.5	4.7	4.9
5	16.5	9.8	11.6	10.9
7	34.0	18.9	23.4	21.7
9	69.6	42.5	52.0	49.8
11	153.7	112.6	125.4	123.4

5. Conclusions

The summary of the work accomplished can be described as follows. First, four different ways to compute coverings were implemented in Franz-LISP Opus 38.92 on VAX 11/780. This was done at *csvax* computer which is available at the Computer Science Department of University of Kansas. Several families of example sets were generated and used to compute the coverings using all four programs on the system. Some of the sets of examples were used to compute coverings manually and matched with the ones computed by the programs to make sure that the programs were correct. Some programs were run many times to assess the reliability of the method. The time statistics for running all the data of example sets to compute the coverings using all four programs were collected, tabulated and represented as relevant graphs. Based on all the above work, a conclusion was reached as to which algorithm is most efficient.

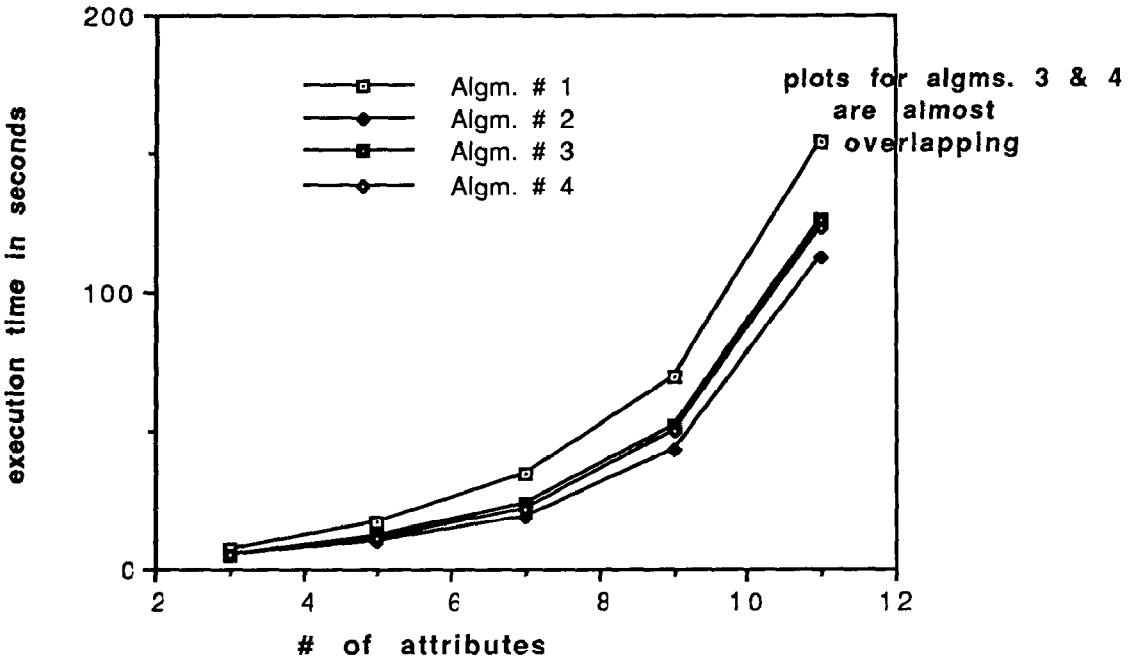

Figure 5. Family 5 of example sets, all with ten examples

A clear winner, which is the Partition Method (algorithm # 2), is closely followed by the Lower Boundary Method (algorithm #4). Trailing far behind is the Characteristic Set Method (algorithm #3). The Indiscernibility Relation Method (algorithm # 1) turned out to be the worst of all algorithms. The respective superiority of each algorithm is established not only in terms of time, but of space as well. The ranking with respect to space complexity is further confirmed by the failure of some algorithms due to space limitations. The two best algorithms (Partition Method and Lower Boundary Method) are very close. In such a case, looking at both algorithms and then cleverly programming parts of them may change the picture. We have avoided taking such clever short-cuts, because that may not be fair to the comparative assessment.

References

[1] D.W. Aha. Incremental, instance-based learning of independent and graded concept descriptions. *Proc. 6th Int. Workshop on Machine Learning, Cornell University, Ithaca, NY, June 26 - 27, 1989*, 387 - 391.

[2] C.-C. Chan, J.W. Grzymala-Busse. Rough-set boundaries as a tool for learning rules from examples. *Proc. 4th Int. Symp. on Methodologies for Intelligent Systems, Charlotte, NC, October 12 - 14, 1989*, 281 - 288.

[3] J. Dean, J.W. Grzymala-Busse. An overview of the learning from examples module LEM1. *Report TR-88-2, Department of Computer Science, University of Kansas, 1988.*

[4] D.H. Fisher. Conceptual clustering, learning from examples, and inference. *Proc. 4th Int. Workshop on Machine Learning, University of California, Irvine, CA, June 22 - 25, 1987*, 38 - 49.

[5] D.H. Fisher, J.C. Schlimmer. Concept simplification and prediction accuracy. *Proc. 5th Int. Conf. on Machine Learning, University of Michigan, Ann Arbor, MI, June 12 - 14, 1988*, 22 - 28.

[6] J.W. Grzymala-Busse. An overview of the LERS1 learning system. *Proc. 2nd Int. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Tullahoma, TN, June 6 - 9, 1989*, 838 - 844.

[7] J.W. Grzymala-Busse, D. Sikora. LERS1 - A system for learning from examples based on rough sets. *Report TR-88-5, Department of Computer Science, University of Kansas, 1988.*

[8] Y. Kodratoff. *Introduction to Machine Learning.* Morgan Kaufmann, 1988.

[9] R.S. Michalski. A theory and methodology of inductive learning. In *Machine Learning. R. S. Michalski, J.G. Carbonell, T.M. Mitchell (eds.)*, Tioga Publ. Co., 1983, 83 - 134.

[10] M.W. van Someren. Using attribute dependencies for rule learning. In *Knowledge Representation and Organization in Machine Learning. K. Morik (ed.). Lecture Notes in AI*, Springer-Verlag, 1989, 192 - 210.

[11] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, 1982.