Reports of the GMU Machine Learning and Inference Laboratory, MLI 95-3, March 1995.

HOW DID AQ FACE THE EAST-WEST CHALLENGE?

An Analysis of the AQ Family's Performance in the 2nd International Competition of Machine Learning Programs

E. Bloedorn, I. Imam, K. Kaufman, M. Maloof, R.S. Michalski*, and J. Wnek

Center for Machine Learning and Inference
George Mason University
Fairfax, VA 22030
*Also with the Institute of Computer Science, Polish Academy of Sciences
01-237 Warsaw, Poland

ABSTRACT

The "East-West Challenge" is the title of the second international competition of machine learning programs, organized in the Fall 1994 by Donald Michie, Stephen Muggleton, David Page and Ashwin Srinivasan from Oxford University. The goal of the competition was to solve the "TRAINS problems", that is to discover the "simplest" classification rules for train-like structured objects. The rule complexity was judged by a Prolog program that counted the number of various components in the rule expressed in the from of Prolog Horn clauses. There were 65 entries from several countries submitted to the competition. The GMU team's entry was generated by three members of the AQ family of learning programs: AQ-DT, INDUCE and AQ17-HCI. The paper analyses the results obtained by these programs and compares them to those obtained by other learning programs. It also presents ideas for further research that were inspired by the competition. One of these ideas is a challenge to the machine learning community to develop a measure of knowledge complexity that would adequately capture the "cognitive complexity" of knowledge. A preliminary measure of such cognitive complexity, called C-complexity, different from the Prolog-complexity (P-complexity) used in the competition, is briefly discussed.

Acknowledgment

The authors thank Professors Donald Michie, Steve Muggleton, David Page and Ashwin Srinivasan for organizing the "West-East Challenge" competition of machine learning programs, which provided us with a stimulating challenge for our learning programs and inspired new ideas for improving them. The authors also thank Nabil Allkharouf and Ali Hadjarian for their help and suggestions in the efforts to solve problems posed by the competition.

This research was conducted in the Center for Machine Learning and Inference at George Mason University. The Center's research is supported in part by the Advanced Research Projects Agency under Grant No. N00014-91-J-1854, administered by the Office of Naval Research, and Grant No. F49620-92-J-0549, administered by the Air Force Office of Scientific Research, in part by the Office of Naval Research under Grant No. N00014-91-J-1351, and in part by the National Science Foundation under Grants No. IRI-9020266, CDA-9309725 and DMI-9496192.

1. Introduction

Recent years have seen a great proliferation of efforts to apply machine learning methods to practical domains. These efforts have brought a better understanding of the strengths and weaknesses the existing methods, and produced useful insights as to what domains these methods best apply. An important part of these efforts was an international competition of machine learning programs organized by Tom Mitchell, Sebastian Thrun and John Cheng from Carnegie Mellon University in 1991. The competition originated during the 1991 Summer School on Machine Learning at the Priory Corsendonk in Belgium, and in reflection of this, the problems posed in the competition were called "MONKS problems". This was first such competition of Machine Learning programs. About 20 machine learning programs were applied to the posed problems by research teams from the US and various countries in Europe. The research teams that participated in the competitions were from Carnegie Mellon University, George Mason University, Josef Stefan Institute, University of Karlsruhe, University of Zurich, and Vrije Universiteit Brussel. The problems and the results of the competition have been described in (Thrun et al., 1991; see also Wnek and Michalski, 1994a for additional information). The MONKS problems have subsequently become a tested for many other learning programs.

The Cornsendonk competition has demonstrated a significant interest of machine learning researchers in applying their learning programs to the same set of problems in order to develop an insight into their performance and limitations. Following the success of the first competition, the second international competition of machine learning programs, called the East-West Challenge, was organized in 1994. Organizers were Donald Michie, Stephen Muggleton, David Page and Ashwin Srinivasan from Oxford University in England (Michie, et al, 1994). In contrast to the MONKS' problems designed to test programs for learning attributional (or attribute-based) descriptions, the new challenge was designed to test programs for learning relational descriptions. These problems were particularly suitable for programs employing inductive logic programming—a new and very active research subarea in machine learning, particularly in Europe.

The East-West Challenge involved learning simplest rules for classifying TRAINS or train-like structures into "Eastbound" or "Westbound". The trains have a variable number of cars; cars are of different shapes and can carry different loads. A natural characterization of such structures requires a language for representing relational descriptions, such as first order predicate logic or annotated predicate calculus (Michalski, 1983). Inductive logic programming is therefore a particularly suitable approach to such problems. The original TRAINS problem was first proposed by R.S. Michalski over 20 years ago, and was used to test the INDUCE program for learning structural descriptions (Michalski, 1980).

2. Rules of the Competition

The competition included many more trains than in the original TRAINS problem presented in (Michalski, 1980). The new TRAINS problem consisted of three separate learning problems, called Competition 1, 2, and 3, respectively, as described by Michie, et al. (1994):

> **Competition 1**
>
> As in scientific discovery, it is required to conjecture some plausible Law, in this case governing what kinds of trains are Eastbound and what kind are Westbound. Merging the new trains (Figure 2) with Michalski's original ten (Figure 1) the competition organizers applied a freshly conjectured Law to yield class labels for the resultant set of twenty. Can inductive inference recover the new Law, or one as good or better, fitting all 20? The best entry is to be judged on accuracy and simplicity.

The additional ten trains were selected from a randomly generated pool and assigned class labels, all in a way that ensured that the resulting set of 20 was split into East and West subsets by a new Law known as Theory X. The trains generator itself applied attribute constraints suggest by Michalski's original ten train example as follows:

1. A train has two, three or four cars, each of which can either be long or short.

2. A long car can have either two or three axles.

3. A short car cab be rectangular, u-shaped, bucket-shaped, hexagonal, or elliptical, while a long car must be rectangular.

4. A hexagonal or elliptical car is necessarily closed, while any other car can be either open or closed.

5. The roof of a long closed car can be either flat or jagged.

6. The roof of a hexagonal car is necessarily flat, while the roof of an elliptical car is necessarily an arc. Any other short closed car can have either a flat of a peaked roof.

7. If a short car is rectangular then it can also be double-sided.

8. A long car can be empty if it can contain one, two or three replicas of one of the following kinds of load: circle, inverted-triangle, hexagon, rectangle.

9. A short car contains either one or two replicas of the following kinds of load: circle, triangle, rectangle, diamond.

10. No sub-distinctions are drawn among rectangular loads, even though some are drawn square and others more or less oblong. The presumption is that they are drawn just as oblong as they need to be in each case to fill the available container space.

11. In Michalski's original version a possible distinction between hollow and solid wheels was ignored, as is also done here.

Muggleton's Prolog train-generator embodies the above constraints together with certain distributional assumptions concerning values of descriptors, so as to preserve statistical coherence with Michalski's original ten. ... The simplest law Y received that correctly classified the twenty trains of Figures 1 and 2 (see Table 1 for their equivalent Prolog representation) won competition 1. (Michie et al., pp. 2,3)

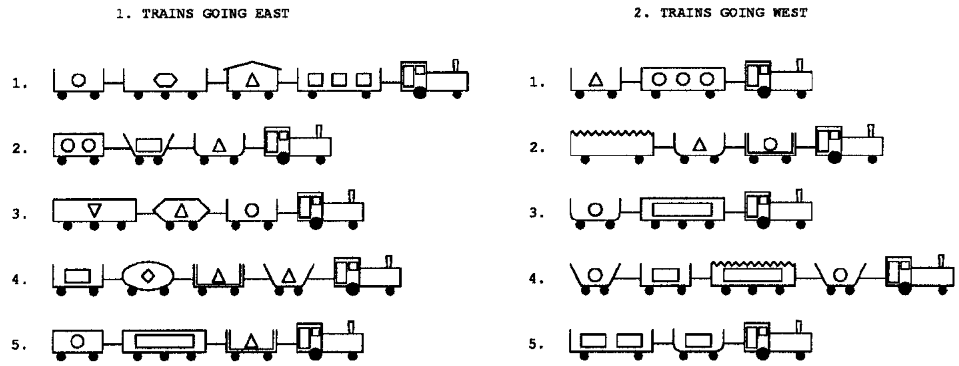


Figure 1. Michalski's original set of trains (Michie et al., 1994).

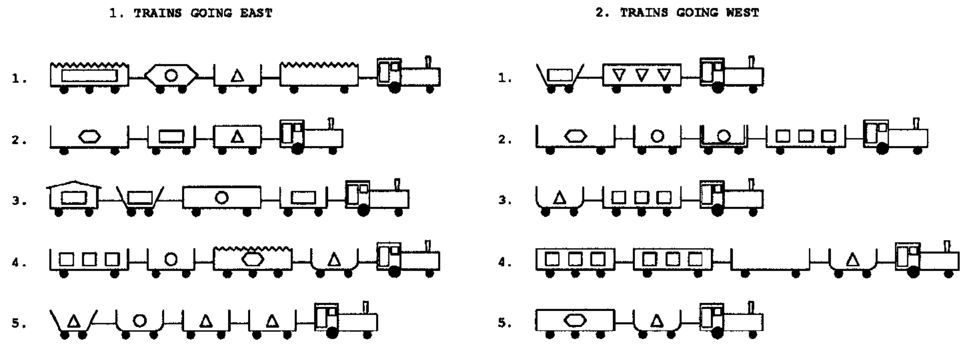


Figure 2. A new set of 10 trains created using Muggleton's train generator (Michie et al. 1994).

Competition 2

The rules in Competition 1 failed to allow for subsymbolic and semi-symbolic forms of inductive analysis, ranging from multivariate non-linear statistical approaches through neural networks and genetic algorithms to paranormal and other intuitive human mental skills. A separate competition was accordingly available for entries in the form of an allocation of Eastbound/Westbound labels to the 100 trains of Figure 3 (and Table 2) unaccompanied by any classifying rule or formula. Since X [the intended rule for Competition 1] was the simplest known to the organizers, they provisionally took it as the oracle for adjucating a subsymbolically derived classifications of the test set of 100. But what if a subsymbolic learner uncovered a classification that was closest to an entry Z either taken from Competition 1 or otherwise unknown to them? Provided that the complexity of the score of Z lay within the bottom quartile of the scores of all such theories, then Z's class labellings would be used for assessment of that subsymbolic learner. ... For this sub-symbolic section the solution that classified, on the above assessment principle, the highest number of new trains would win the second competition. (p. 5)

Competition 3

Returning to theory discovery, a further challenge was proposed, this time based on induction from trains generated and pre-classified entirely randomly (strictly speaking, pseudo-randomly). For this final exercise the cross-ruling shown on Figure 4 partitioned 50 trains into five sets of ten. Arbitrarily assigning "Eastbound" to the trains in the left column and "Westbound" to those in the right column, five separate induction tasks were set up analogous to Michalski's original. ... First place went to the entry with the lowest grand total complexity summed over the five sub-tasks..." (p. 6)

3. Applying AQ Programs to West-East Challenge

3.1 General Comments

The MLI research team applied three programs from the AQ family: AQDT (Michalski & Imam, 1994), INDUCE (Hoff, Michalski & Stepp, 1980) and AQ17-HCI (Wnek & Michalski, 1994b) to the problems of the competition. The programs and results obtained from their application are briefly described in the following sections. The application of each program involved a transformation of the original Prolog representation of the problems into the form required by the program.

Figure 3a. 1-50 out of 100 trains used in Competition 2.

Our early results from applying the programs to the TRAINS problems indicated difficulties with the metric for measuring complexity of the solutions proposed by the organizers. The proposed metric measured the complexity of the learned description expressed as a Prolog program. The complexity score was computed automatically also by a Prolog program. To distinguish this measure of complexity from other measure that we proposed (see below), we called it P-complexity (Prolog-complexity; see Appendix 5). We found that solutions with low P-complexity might have high complexity as evaluated intuitively by a person, and conversely, a solution that seemed simple to a person and easy to express in English, might have high P-complexity.

Figure 3b. 51-100 out of 100 trains used in Competition 2.

In order to rectify this problem, we proposed another measure of complexity, which we called C-complexity (Cognitive-complexity). Details of this metric are provided in Section 3.2. For comparison, results from each of program have been evaluated using both measures of complexity. Section 4 provides a summary of results.

3.2 Measuring Complexity of Knowledge Learned by the Programs

Although the TRAINS domain seems relatively simple, it is not easy for a person to find the simplest solutions. There is a tremendous number of classification rules that are complete and consistent with the given training example set in each problem. (If the data contained noise, the best solutions may not be consitent and/or complete with regard to the input examples (Bergadano

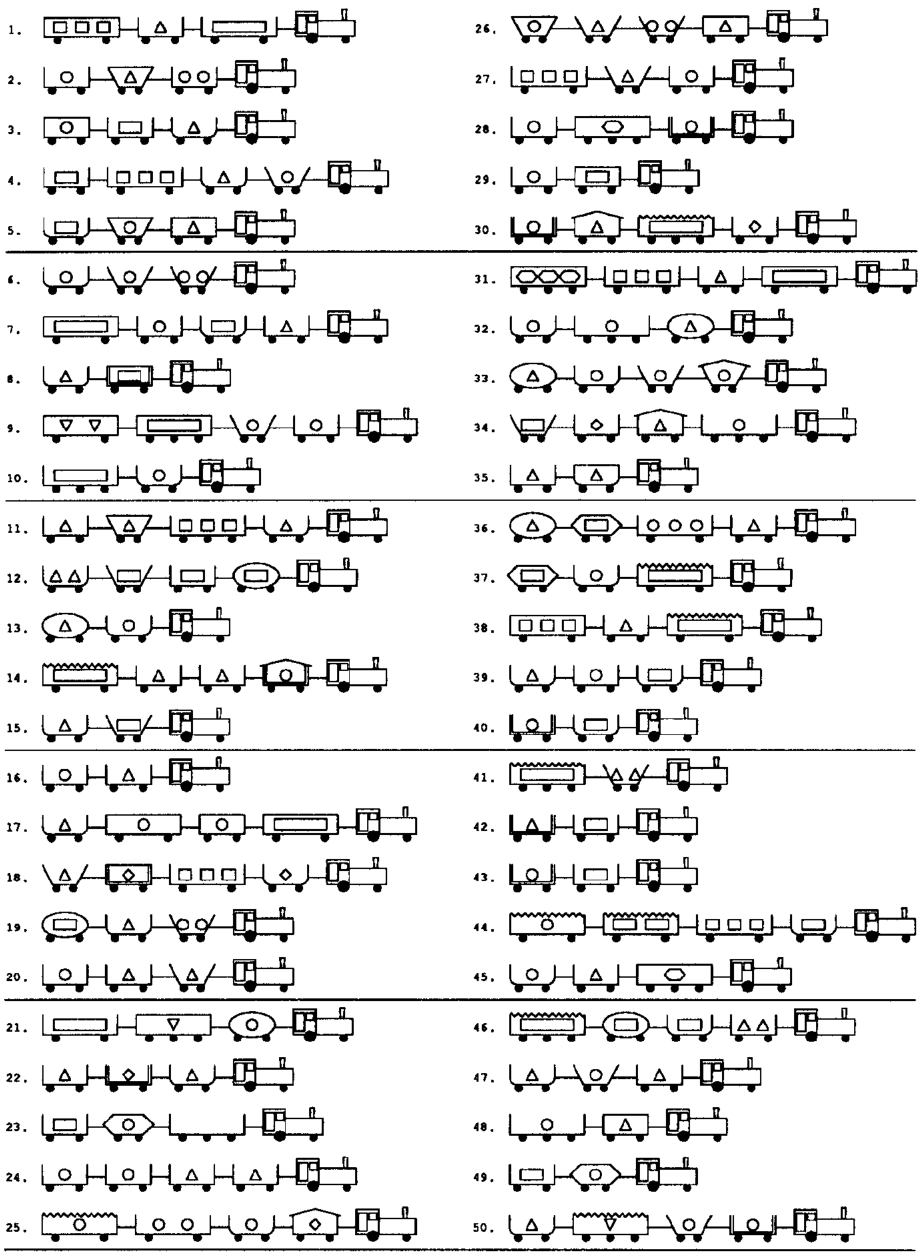


Figure 4. 50 trains grouped into five classes used in Competition 3 (Michie et al., 1994).

et al., 1992). An important issue then is how to decide which of the many logically acceptable solutions is the most desirable. This calls for some metric for evaluating solutions.

The nature of the domain often has an impact on the ranking of rules. Some attributes may be more complex or more costly to measure than others, and rules that do not involve their evaluation would be preferred. In some domains simple, general rules may be more useful, while detailed, specific rules may be better in others. But in designed domains, such as the TRAINS world used in the competition, there are no external cues or implicit goals to guide one's preference criteria. In such cases, human or automated learners typically use preference criteria based on some measure of simplicity of the solutions.

The first criterion is *syntactic simplicity*. Why bring in many conditions in a rule if fewer would suffice? For example, "Eastbound trains have a car with a triangular load" would, all things being equal, be preferred over "Eastbound trains have an even number of cars and two wheels on the rearmost car." The other criterion is *ease of understanding*. Simple, interrelated concepts within the rule will be more readily remembered than a haphazard collection of conditions, even if the latter is shorter. For example, "The third car on an eastbound train is rectangular, single-walled, jagged-topped, and carries a single load" may be preferable to the simpler "A train is eastbound if its first car has three wheels and it has a car with a flat top in front of a double-walled car." We propose that a *conceptual complexity* (C-complexity) metric be based on these two principles that closely mirror typical human preference criteria.

It is clear that it will take a substantial effort and experimentation to capture accurately such a criterion. However, other metrics based upon simplicity and ease of understanding will tend to rank rulesets similarly. A version of C-complexity introduced and used here to evaluate the complexity of learned descriptions is:

> *The number of words it takes to describe the concept concisely, accurately and understandably in correct English.*

We view the measure as a rough approximation of "true" cognitive complexity. The advantage of the measure is its simplicity. Its weakness is that for each description learned by a system there can be many English translations. In order to make the measure more operational and speaker-independent, we propose to select the shortest expression among those provided by *k* (e.g. three) English speakers.

The proposed C-complexity is more closely linked to simplicity than it is to ease of understanding. However, combinations of conditions with high conceptual cohesion may reduce to a shorter form, as did the example above in which the characteristics of the third car could be enumerated without the redundant "The third car is".

In the competition, rules were evaluated using a complexity metric based on the simplicity of the Prolog representation of the rules. This metric, which we call *P-complexity*, would appear to be less of an indicator of conceptual complexity than C-complexity, because some of the easily representable constructs of Prolog such as list representation and predicate recursion do not have simple representations in our mind or in our written language. The elegant Prolog rule "eastbound(A | B) if (closed(A) or has-load(B, triangle)) and (short(A) or eastbound(B))" translates to an English representation something like "A train is eastbound if its first car is closed or one of its other cars has a triangular load, and also either the first car is short or the rest of the train has the eastbound property." Not only is the English lengthy, but people will generally have trouble assimilating the last clause — they have to divide the rest of the train into a first and rest, and then reapply the entire rule, only once if they are lucky.

In any application or competition, the criteria used for ranking the various possibilities are of the utmost importance. In applications in which inductive logic programming will be the sole method of rule generation and application and the efficiency rather than the understandability of the rules is of paramount importance, a preference metric such as P-complexity should be used . However, in most domains, something more like C-complexity is likely to be a better preference criterion.

The competition's first and third problems were judged by P-complexity (assuming complete and consistent rulesets), and hence was oriented toward certain classes of rules. Since the programs described in this paper tend to be oriented toward lower C-complexities, some of the "good" results they obtained would have scored poorly in the competition. Nonetheless, we found many of the P-complex rules "good" and/or "interesting" in their own regard, often due to low C-complexities. Hence this paper will discuss rules that were submitted to the competition and those that were not, those with lower P-complexities and higher C-complexities and vice versa.

The best rules in most of these competitions had P- and/or C-complexities below 20. But sometimes one complexity measure would be low while the other was very high. For instance, the winning entry from Competition 1, “The train has either a closed last car or a triangular load in a car other than the last one. Also, either the last car is short or the train, after the last car is removed, has the above property,” has a P-complexity of 16 and a C-complexity of 40. Conversely, the INDUCE-generate rule, “All cars have two wheels and the second car does not have a rectangular load,” has a P-complexity of 31 and a C-complexity of 15.

4. AQDT-2 System

4.1 Program description

The AQDT-2 system (Michalski & Imam, 1994) learns task-oriented decision structures from decision rules or from examples. This approach was motivated by the need to build a learning and discovery system able not only to generate and store knowledge, but also to use it effectively for decision making. Knowledge can be easily acquired and stored in declarative form; however the form in which knowledge can be most readily used is procedural. A *decision structure* is a directed acyclic graph that specifies an order of tests to be applied to an object (or a situation) to arrive at a decision about that object. The nodes of the structure are assigned individual tests (which may correspond to a single attribute, a function of attributes, or a relation), the branches are assigned possible test outcomes (or ranges of outcomes), and the leaves are assigned one specific decision or a set of candidate decisions (with corresponding probabilities), or an *undetermined* decision. A decision structure reduces to a familiar decision tree when each node is assigned a single attribute and has at most one parent; the branches from each node are assigned single values of that attribute; and leaves are assigned single, definite decisions.

A decision tree/decision structure can be an effective tool for describing a decision process, as long as all the required tests can be performed easily, and the decision-making situations it was designed for remain constant. Problems arise when these assumptions do not hold. For example, in some situations measuring certain attributes may be difficult or costly. In such situations it is desirable to reformulate the decision structure so that the “inexpensive” attributes are evaluated first (by assigning them to the nodes close to the root), and the "expensive" attributes are evaluated only if necessary (they are assigned to the nodes far away from the root). If an attribute cannot be measured at all, it is useful to either modify the structure so that it does not contain that attribute, or, when this is impossible, to indicate alternative candidate decisions and their probabilities. A restructuring may also be desirable if there is a significant change in the frequency of occurrence of different decisions.

The restructuring of a decision structure (or a tree) in order to suit new requirements is usually quite difficult. This is because a decision structure is a procedural knowledge representation, which imposes an evaluation order on the tests. In contrast, no evaluation order is imposed by a declarative representation, such as a set of decision rules. Tests (conditions) of rules can be evaluated in any order. Thus, for a given set of rules, one can usually build a large number of logically equivalent decision structures (trees), which differ in the test ordering. Due to the lack of “order constraints,” a declarative representation (a ruleset) is much easier to modify to adapt to different situations than a procedural one (a decision structure or a tree). On the other hand, to apply decision rules to make a decision, one needs to decide in which order tests are evaluated, and thus, needs a decision structure.

In the AQDT method a test is selected from an available set of tests based on its *utility* (see below) for the given set of decision rules. The test (attribute) utility is a combination of one or more of the following elementary criteria: 1) *disjointness*, which captures the effectiveness of the test in discriminating among decision rules for different decision classes; 2) *importance,* which determines the importance of a test in the rules; 3) *value distribution*, which characterizes the distribution of the test importance over its of values; and 4) *dominance*, which measures the test presence in the rules. These criteria are defined below.

The description of each class is in the form of a ruleset. Assume that this set is the initial *ruleset context*.

Step 1: Evaluate each attribute occurring in the ruleset context using the LEF attribute ranking measure. Select the highest ranked attribute. Let **A** represent this highest-ranked attribute.

Step 2: Create a node of the tree (initially, the root; afterwards, a node attached to a branch), and assign to it the attribute **A**. In standard mode, create as many branches from the node as there are legal values of the attribute **A,** and assign these values to the branches. In compact mode, create as many branches as there are disjoint value sets of this attribute in the decision rules, and assign these sets to the branches.

Step 3: For each branch, associate with it a group of rules from the ruleset context that contain a condition satisfied by the value(s) assigned to this branch. For example, if a branch is assigned values i of attribute **A**, then associate with it all rules containing condition [**A= i** v ...]. If a branch is assigned values i v j, then associate with it all rules containing condition [**A= i** v **j** v ...]. Remove these conditions from the rules. If there are rules in the ruleset context that do not contain attribute **A,** add these rules to all rule groups associated with the branches stemming from the node assigned attribute A. (This step is justified by the consensus law: [x=1] ≡ {[x=1] & [y= a] v [x=1] & [y=b]}, assuming that a and b are the only legal values of y.) All rules associated with the given branch constitute a ruleset context for this branch.

Step 4: If all the rules in a ruleset context for some branch belong to the same class, create a leaf node and assign that class to it. If all branches of the trees have leaf nodes, stop. Otherwise, repeat steps 1 to 4 for each branch that has no leaf.

The AQDT approach allows one to generate a decision structure that avoids or delays evaluating an attribute that is difficult to measure, restructures the values of an attribute based on their importance, and weights each example. Initial research on this approach, and the first system implementation, AQDT-1, is described in (Imam & Michalski, 1993). AQDT-2 generates a goal-oriented decision structure from examples or decision rules learned by either the AQ15c (Wnek et al., 1995) or AQ17 (Bloedorn, et al, 1992) rule learning system, the latter of which has extensive constructive induction capabilities.

4.2 Problem Formulation

Because the examples in the train problems were originally described in terms of Prolog clauses, it was necessary to translate these clauses into representations suitable for the different programs. The AQDT-2 program accepts rules or examples in the form of arrays of attribute-value vectors, and can accept examples with different numbers of attribute-value pairs, so that a train with two cars can be expressed in terms of a smaller set of attributes than a train with three or four cars.

To describe the train problem in a format suitable for AQDT-2, a set of eight (8) attributes was generated that could completely describe any car in the train. To recognize the number (position) of a given car in the train, each of the eight attributes is associated with a two-digit code; the first digit identifies the location of the car and the second identifies the attribute itself. For example, the number 3 in the attribute-name “x32” refers to the third car, and the number 2 refers to the second attribute (as shown below, the car shape).

x*1 = Car_top
0 = open 1 = closed

x*2 = Car_shape
0 = rectangle 1 = hexagon 2 = bucket 3 = u_shaped 4 = ellipse

x*3 = Car_length
0 = short 1 = long

x*4 = Car_frame
0 = not_double 1 = double

x*5 = Car_top_shape
0 = none 1 = peaked 2 = flat 3 = arc 4 = jagged

x*6 = Number_of_wheels
2 = two 3 = three

x*7 = Load_shape
0 = rectangle 1 = hexagon 2 = circle 3 = triangle 4 = utriangle 5 = diamond

x*8 = Number_of_loads
0 = no_loads 1 = one 2 = two 3 = three

Figure 4.1 The set of attributes used in the experiments

Prolog Format
eastbound([c(1,rectangle,short,not_double,flat,2,l(circle,2)),
c(2,bucket,short,not_double,none,2,l(rectangle,1)),
c(3,u_shaped,short,not_double,none,2,l(triangle,1))]).

AQDT-2 Format
eastbound-events
[x11 = 1][x12 = 0][x13 = 0][x14 = 0][x15 = 2][x16 = 2][x17 = 2][x18 = 2]
[x21 = 0][x22 = 2][x23 = 0][x24 = 0][x25 = 0][x26 = 2][x27 = 0][x28 = 1]
[x31 = 0][x32 = 3][x33 = 0][x34 = 0][x35 = 0][x36 = 2][x37 = 3][x38 = 1]

Figure 4.2 An example of Prolog and AQDT-2 descriptions of the same train.

Figure 4.1 shows the attributes and their legal values as they were defined for AQDT-2. The symbol '*' refers to the car number. Figure 4.2 shows an example of a description of one train in Prolog and its corresponding representation in this AQDT-2 format.

4.3 Deriving Decision Structures for the Trains Problem

AQDT-2 has a set of parameters and criteria for generating decision structures optimized for a given task. AQDT-2 allows the user to use different settings to generate different decision structures for any given set of data

AQDT-2 uses a cost criterion to ignore attributes that are of no interest. The cost criterion can be defined for one or more attributes and/or one or more values of a given attribute. AQDT-2 uses weights to define the strength of each example if there is any difference among them. There are many other properties and parameters in AQDT that can adapt the learning process to achieve the required goal. To solve the trains problem, several AQDT-2 runs were made with different settings of the cost criteria.

In the original problem some of the characteristics of a given car in the train are implicitly given higher costs than others. In other words, the use in rules of certain properties of a given car would result in a higher P-complexity than using other properties.

In the search for the simplest decision structure, we designed an algorithm for seeking the simplest and more accurate decision structure. The algorithm uses some heuristics such as:
1) If the attribute at the root of the decision structure has many branches, then increase its cost.
2) If by reducing the cost of some attribute at the second or higher levels of the decision structure, the complexity (number of nodes) of the decision structure decreases, then reduce the cost of that attribute so that it occupies the root.
3) if the predictive accuracy decreases when the costs of one or more attribute are increased, then do not increase their costs or select another attribute to be the root of the decision structure.

4.4 AQDT-2 Results

4.4.1 Competition #1

The AQDT-2 results are presented in four different forms: 1) The decision tree output of AQDT-2, 2) APC (the "Annotated Predicate Calculus" description language) syntax, 3) English, and 4) Prolog. When AQDT-2 generated a set of different decision structures for a given problem, they were combined into a single rule. The P-complexity of most of these rules ranges between 20 and 22; however the reported rules are those with either the best P-complexity or C-complexity.

Solution #1
Decision Tree:

```
x37  = else
|   x34  = 0
|   |   x23  = 0
|   |   |   x12  = 0: eastbound
|   |   |   x12  = 2: westbound
|   |   x23  = 1: westbound
|   x34  = 1: westbound
x37  = 3: eastbound
```

Number of nodes: 4
Number of leaves: 5

APC syntax:
[lshape(car3)=triangle] OR
[cshape(car1)=rectangle][length(car2)=short][double(car3)=false]

English:
"The third car has a triangular load, or the first car is rectangularly shaped, the second car is short, and the third car is not double."

Prolog:

```
eastbound([Car1,Car2,Car3|_]) :-
has_load0(Car3,triangle);
(rectangle(Car1),
short(Car2),
\+ double(Car3)).
```

P-complexity: 20
C-complexity: 26

Solution #2
Decision Tree:

```
x37 = else
|   x12 = 0
|   |   x31 = 0
|   |   |   x22 = 0: westbound
|   |   |   x22 = 1: eastbound
|   |   |   x22 = 3: westbound
|   |   x31 = 1: eastbound
|   x12 = 2: westbound
x37 = 3: eastbound
```

Number of nodes: 4
Number of leaves: 6

APC syntax:
[lshape(car3)=triangle] OR
[cshape(car1)=rectangle]
([top(car3)=closed] OR
[cshape(car2)=hexagon])

English:
"The third car has a triangular load, or the first car is rectangularly shaped, and the third car is closed, or the second car is hexagon-shaped."

Prolog:

```
eastbound([Car1,Car2,Car3|_]) :-
has_load0(Car3,triangle);
(rectangle(Car1),
(closed(Car3);
hexagon(Car2)).
```

P-complexity: 20
C-complexity: 26

Solution #3
Decision Tree:

```
x37 = 0: westbound
x37 = 1: eastbound
x37 = 2
|   x34 = 0: eastbound
|   x34 = 1: westbound
x37 = 3: eastbound
```

Number of nodes: 2
Number of leaves: 5

APC syntax:
[lshape(Car3)=triangle or hexagon] OR
[lshape(Car3)=circle][double(Car3) = false]

English:
"The third car contains a triangular or hexagonal load, or contains a circular load and is not double."

Prolog:

```
eastbound([_,_,Car3|_]) :-
has_load0(Car3,triangle);
has_load0(Car3,hexagon);
(has_load0(Car3,circle),
not(double(Car3))).
```

P-complexity: 22
C-complexity: 18

4.4.2 Competition #2

In this competition, the labeling of the second data set was generated based on the decision structure shown above in Solutions 1 and 2. As this decision structure was in the bottom quartile of oracles received for competition #1, there was a perfect match between this oracle and the class labeling given. However, the judges determined later that prizes for competition #2 be based on the average degree of match between the submitted class labellings and all lowest quartile oracles for competition #1. Below is the official results for the labellings submitted by AQDT-2 for competition #2:

Solution 1: aqdt1
40 eastbound and 60 westbound

Train 1	E	Train 2	E	Train 3	E	Train 4	E	Train 5	E
Train 6	W	Train 7	E	Train 8	W	Train 9	W	Train 10	W
Train 11	E	Train 12	W	Train 13	W	Train 14	E	Train 15	W
Train 16	W	Train 17	W	Train 18	W	Train 19	W	Train 20	E
Train 21	W	Train 22	E	Train 23	E	Train 24	E	Train 25	W
Train 26	W	Train 27	E	Train 28	W	Train 29	W	Train 30	E
Train 31	E	Train 32	E	Train 33	W	Train 34	E	Train 35	W
Train 36	W	Train 37	W	Train 38	E	Train 39	W	Train 40	W
Train 41	W	Train 42	W	Train 43	W	Train 44	W	Train 45	W
Train 46	E	Train 47	E	Train 48	W	Train 49	W	Train 50	W
Train 51	E	Train 52	E	Train 53	W	Train 54	W	Train 55	E
Train 56	W	Train 57	E	Train 58	W	Train 59	E	Train 60	W
Train 61	W	Train 62	W	Train 63	E	Train 64	W	Train 65	W
Train 66	E	Train 67	W	Train 68	E	Train 69	W	Train 70	E
Train 71	E	Train 72	E	Train 73	W	Train 74	W	Train 75	W
Train 76	W	Train 77	E	Train 78	W	Train 79	E	Train 80	W
Train 81	E	Train 82	W	Train 83	W	Train 84	W	Train 85	W
Train 86	E	Train 87	E	Train 88	E	Train 89	E	Train 90	W
Train 91	W	Train 92	W	Train 93	W	Train 94	W	Train 95	W
Train 96	E	Train 97	W	Train 98	W	Train 99	W	Train 100	W

Solution 2: aqdt2
32 eastbound and 68 westbound.

Train 1	W	Train 2	W	Train 3	E	Train 4	E	Train 5	E
Train 6	W	Train 7	W	Train 8	W	Train 9	W	Train 10	W
Train 11	W	Train 12	W	Train 13	W	Train 14	E	Train 15	W
Train 16	W	Train 17	W	Train 18	W	Train 19	W	Train 20	E
Train 21	E	Train 22	E	Train 23	E	Train 24	E	Train 25	W
Train 26	W	Train 27	W	Train 28	W	Train 29	W	Train 30	E
Train 31	E	Train 32	E	Train 33	W	Train 34	E	Train 35	W
Train 36	W	Train 37	W	Train 38	E	Train 39	W	Train 40	W
Train 41	W	Train 42	W	Train 43	W	Train 44	W	Train 45	W

Train 46	W	Train 47	E	Train 48	W	Train 49	W	Train 50	W
Train 51	W	Train 52	E	Train 53	W	Train 54	E	Train 55	W
Train 56	W	Train 57	E	Train 58	W	Train 59	E	Train 60	W
Train 61	W	Train 62	E	Train 63	E	Train 64	W	Train 65	W
Train 66	E	Train 67	E	Train 68	W	Train 69	W	Train 70	W
Train 71	E	Train 72	E	Train 73	W	Train 74	W	Train 75	W
Train 76	W	Train 77	E	Train 78	W	Train 79	E	Train 80	W
Train 81	W	Train 82	E	Train 83	W	Train 84	E	Train 85	W
Train 86	W	Train 87	E	Train 88	E	Train 89	W	Train 90	W
Train 91	W	Train 92	W	Train 93	W	Train 94	W	Train 95	W
Train 96	E	Train 97	W	Train 98	W	Train 99	W	Train 100	W

4.4.3 Competition #3

In the third competition, AQDT-2 was applied to each of the 5 subproblems. The best P-complexity and C-complexity rules found are reported below. The sum of the P-complexities of the AQDT-2 Prolog rules is 85.

Set #1:
Best solution
Decision Tree:

```
x21 = 0
|   x22 = 0: eastbound
|   x22 = 2: westbound
x21 = 1
|   x22 = 0: westbound
|   x22 = 2: eastbound
```

Number of nodes: 3
Number of leaves: 4

APC syntax:
[top(car1)=none][cshape(car1)=rectangle] OR
[top(Car1)#none][cshape(car1)=bucket]

English:
"The second car is open and rectangle, or it is closed and bucket shaped."

Prolog:

```
eastbound([_,Car2|_]) :-
                        (open(Car2),
                         rectangle(Car2));
                        (closed(Car2),
                         bucket(Car2)).
```

P-Complexity: 16
C-Complexity: 14

Set #2
Best solution
Decision Tree:

```
x13 = 0
|   x12 = 0: westbound
|   x12 = 2: westbound
|   x12 = 3
```

```
|  |  x23 = 0: eastbound
|  |  x23 = 1: westbound
|  x12 = 4: westbound
x13 = 1
|  x28 = 1: eastbound
|  x28 = 3: westbound
```

Number of nodes: 6
Number of leaves: 7

APC syntax:
[length(car1)=long][lqty(car2)=1] OR
[cshape(car1)=u_shaped][length(car2)=short]

English:
"The first car is long and the second car has one load, or the first car is u-shaped and the second car is short."

Prolog:

```
eastbound([Car1,Car2|_]) :-
                    (long(Car1),
                     has_load(Car2,1));
                    (u_shaped(Car1),
                     short(Car2)).
```

P-Complexity: 17
C-Complexity: 24

Set #3
Best solution
Decision Tree:

```
x15 = else
|  x22 = 0: westbound
|  x22 = 1: westbound
|  x22 = 2: eastbound
|  x22 = 3
|  |  x12 = 0: westbound
|  |  x12 = 1: westbound
|  |  x12 = 4: eastbound
x15 = 4: eastbound
```

Number of nodes: 3
Number of leaves: 7

APC syntax:
[top(car1)=jagged] OR
[cshape(car2)=bucket] OR
[cshape(car1)=ellipse][cshape(car2)=u_shaped]

English:
"The top of the first car is jagged, or the second car is bucket, or the first car is elliptical and the second car is u-shaped."

Prolog:

```
eastbound([Car1,Car2|_]) :-
                        arg(5,Car1,jagged);
                        bucket(Car2);
                        (u_shaped(Car2),
                         ellipse(Car1)).
```

P-Complexity: 19
C-Complexity: 26

Set #4
Best solution
Decision Tree:

```
x14  = 0
|   x13  = 0
|   |   x21  = 0
|   |   |   x12  = 0: eastbound
|   |   |   x12  = 3: westbound
|   |   |   x12  = 4: eastbound
|   |   x21  = 1: eastbound
|   x13  = 1: westbound
x14  = 1: westbound
```

Number of nodes: 4
Number of leaves: 6

APC syntax:
[length(car1)=long] OR
[double(car1)=true] OR
[cshape(car1)=u_shaped][top(car2)=none]

English:
"The first car is long or double, or the first car is u-shaped and the second car is open."

Prolog:

```
westbound([Car1,Car2|_]) :-
                        long(Car1);
                        double(Car1);
                        (u_shaped(Car1),
                         open(Car2)).
```

P-Complexity: 17
C-Complexity: 21

Set #5
Best solution
Decision Tree:

```
x12  = 0
|   x32  = 0: eastbound
|   x32  = 3
|   |   x22  = 0: eastbound
|   |   x22  = 4: westbound
|   x32  = 4: eastbound
x12  = 3: westbound
```

Number of nodes: 3

Number of leaves: 5

APC syntax:
[cshape(car1)=rectangle][cshape(car2)#ellipse] OR
[cshape(car1)=rectangle][cshape(car3)#u_shaped]

English:
"The first car is rectangular and the second car is not elliptical, or the first car is rectangular and is not u-shaped."

Prolog:
```
eastbound([Car1,Car2,Car3|_]) :-
                rectangle(Car1),
                (\+ (u_shaped(Car3),
                    ellipse(Car2))).
```

P-Complexity: 16
C-Complexity: 22

---

5. INDUCE

5.1 Program Description

INDUCE is an empirical induction program capable of learning first-order concepts (Michalski 1980; Hoff et al. 1986). Examples and concepts are represented in VL_2, which is a first-order version of Variable-Valued Logic (Hoff et al. 1986). Consequently, INDUCE can learn universally and existentially quantified concepts. Available operators include disjunction, conjunction, and internal disjunction. INDUCE allows two types of background knowledge in the form of logical rules (L-rules) and arithmetic rules (A-rules). These mechanisms facilitate the construction of generalization hierarchies and the expression of structural and arithmetic relationships that often simplify learned concepts. In addition, INDUCE can express concepts using meta-selectors, which can be described as higher-order expressions. Universal quantification is one example of a meta-selector. Another example might be a feature that counts the number of cars that have some property.

A sketch of the INDUCE algorithm is as follows:
1. Select a class to cover. A-rules, L-rules, and meta-selectors are applied to the input examples.
2. Star Generation. Select one of the positive examples to be the *seed*, and generate a *star* for this event. Star generation in INDUCE is a two phase process:
   a. Find a consistent generalization of the seed. This process is guided by user-defined parameters and cost functions.
   b. Convert the consistent generalization from VL_2 rules to VL_1 rules, and use the AQ algorithm to extend references. This process is guided by user-defined parameters and cost functions.
   c. Convert the resultant VL_1 rules back into VL_2 rules.
3. Select the best generalization according to the Lexical Evaluation Function (LEF).

See Hoff et al. (1986) for more details of the INDUCE algorithm.

5.2 Problem Representation
Representing Muggleton's trains in VL2 was a straightforward transformation, although in a pure sense, Muggleton's Prolog representation of Michalski's trains was not in the same spirit as the original INDUCE representation (see Section 5.5). Figure 5.1 shows an example of one of Muggleton's trains. For this competition, Figure 5.2 shows the equivalent INDUCE

representation of the train appearing in Figure 5.1. The translation is intuitive and direct; however; notice that in Figure 5.2, the decision class eastbound is represented by assigning the decision variable *d* the value 1.

```
eastbound([      c(1,rectangle,short,not_double,flat,2,l(circle,2)),
                 c(2,bucket,short,not_double,none,2,l(rectangle,1)),
                 c(3,u_shaped,short,not_double,none,2,l(triangle,1))]).
```

Figure 5.1. Prolog representation of a train.

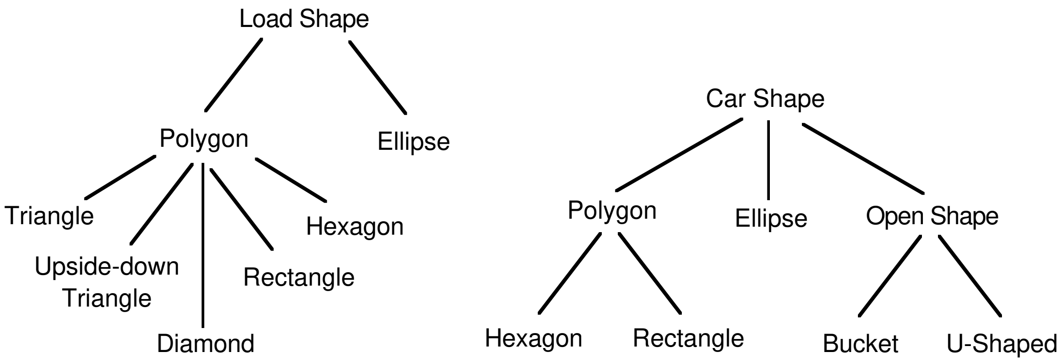
```
[infront(car1,car2)=1][infront(car2,car3)=1]
[pos(car1)=1][cshape(car1)=rectangle][ln(car1)=short][double(car1)=false]
[top(car1)=flat][nwls(car1)=2][lshape(car1)=circle][lqty(car1)=2]
[pos(car2)=2][cshape(car2)=bucket][ln(car2)=short][double(car2)=false]
[top(car2)=none][nwls(car2)=2][lshape(car2)=rectangle][lqty(car2)=1]
[pos(car3)=3][cshape(car3)=u_shaped][ln(car3)=short][double(car3)=false]
[top(car3)=none][nwls(car3)=2][lshape(car3)=triangle][lqty(car3)=1]
=>[d=1].
```

Figure 5.2. INDUCE representation of a train.

5.3 Problem Solving Methodology

Over the six week period that the Center was involved with this competition, many variations were attempted. These included using background knowledge (e.g., a generalization hierarchy of load shapes), derived attributes (e.g., whether the number of loads in the train is odd), various combinations of parameter settings, and removing training examples. Unfortunately, the use of background knowledge did not prove useful for this competition and the use of derived attributes, while producing interesting rules, produced rules that were too costly since Prolog predicates to compute the derived attributes had to be included in computing the P-Complexity.

Early in Competition 1, three types of background knowledge were used: (1) a generalization hierarchy for load shapes (see Figure 5.3a), (2) a generalization hierarchy for car shapes (see Figure 5.3b), and (3) logical transformation rules for car shapes similar to those used in the original trains data set (see Figure 5.3c).



a) Load shape generalization hierarchy

b) Car shape generalization hierarchy

```
[cshape(car1)=rectangle] & [top(car1)=none] & [double(car1)=true]
=>[ctype(car1)=doubleOpenRectangle]
```

c) Example of a car shape transformation rule.

Figure 5.3. Background knowledge used by INDUCE

Several derived attributes were also tried. Table 5.1 lists the derived attributes used for Competition 1. It was hoped that derived attributes were not necessary to win the competition. If this was the case, then the competition would be reduced to someone finding the right piece of background knowledge or the right derived attribute and giving it to a learning program and not a test of the power of the learning algorithm used.

Total Number of Cars
Total Number of Loads
Number of Even Loads
Number of Even Cars
Number of Odd Loads
Number of Odd Cars
Cars Having the Same Load Shape
Cars Having the Same Car Shape
Cars Having the Same Number of Wheels
Cars Having the Same Top
Cars Having the Same Length
Cars Having the Same Load Quantity
The First Car
The Last Car

Table 5.1. Derived attributes used for Competition 1.

INDUCE has a large set of parameters, so any attempt to exhaustively attempt to optimize these parameters is virtually impossible. When running INDUCE, a good approach is to start with a base set of parameter settings and then make modifications to single parameters and see which of these modifications yields the best result using some quality criteria (such as P-complexity), and then use the parameters that produced better results. Essentially, this is a hill-climbing technique through the parameter space.

Initially, the parameters are set to their defaults. The LEF for the VL and AQ phases of generalization were set to maximize coverage of all positive examples, minimize the number of selectors, and minimize coverage negative examples. All tolerances for these parameters were set to 0, which caused the parameters to be considered equally, and strictly in the order that they appeared. At this point, rules were induced for both decision classes 1 (eastbound) and 2 (westbound).

Next, either the coverage parameters were re-ordered or the positive example coverage were changed to maximize coverage of new positive examples. Once the coverage parameters were determined, the affect of meta-selectors was investigated by turning all meta-selectors off. In some cases, the complexity of the induced concepts increased when no meta-selectors were active.

If meta-selectors did indeed simplify learned concepts, then star generation parameters and meta-selectors were set to large numbers (usually 10). If at this point, results were still unsatisfactory, then minimizing the cost of functions and variables would be investigated.

Of course, determining what values and variables to cost and by how much is also a complex procedure. Several schemes were employed. One scheme was to weight predicates based on their P-complexity score. For example, the car shape predicate would have a weight of 3, since any use of the car shape predicate in a concept expressed in Prolog would evaluate to a P-complexity of 3. The square of a predicate's P-complexity, as well as higher powers of a predicate's P-complexity, were also used to weight predicates. Another scheme used, which was more interactive, was to examine a concept and determine which selectors had a high P-complexity and weight the functions and variables of this expensive selector in an effort to eliminate it from further consideration by

INDUCE. On occasion, we had a concept in mind, for example, one induced by AQDT. So a final costing scheme would be to cost those functions and variables appearing in the AQDT solution in an effort to encourage INDUCE to find the same concept. Unfortunately, these costing methods did not work and the reason for this is discussed further in Section 5.5.

5.4 Results

The INDUCE results are shown in four different forms: 1) INDUCE output, 2) APC syntax (the "Annotated Predicate Calculus" description language), 3) English, and 4) Prolog.

5.4.1 Competition 1 Results

Competition 1 Results

Induce:

```
[cshape(car1)=bucket,hexagon][pos(car1)=2] OR
[lshape(car1)=hexagon,triangle][pos(car1)=3]
```

APC:

```
[cshape(car2)=bucket or hexagon] OR
[lshape(car3)=hexagon or triangle]
```

English:

The third car contains a triangular or hexagonal load, or the second car is either hexagon or bucket shaped.

Prolog:

```
eastbound([_,Car2,Car3|_]) :-
  bucket(Car2);
  hexagon(Car2);
  has_load0(Car3,hexagon);
  has_load0(Car3,triangle).

p-Complexity: 22
c-Complexity: 19
```

5.4.2 Competition 3 Results

Competition 3 Results-Set 1 Rules:

Induce:

```
[cshape(car1)=ushaped] OR
[num-diff-pos=3][nwls(car1)#3][top(car1)#none]
```

APC:

```
[cshape(car1)=ushaped] OR
[num-diff-pos=3][nwls(car1)#3][top(car1)#none]
```

English:

There is a u-shaped car or the train has three cars with some closed car having two wheels.

Prolog:

```
eastbound(Train) :-
  has_car(Train,Car),
  (u_shaped(Car);
   (len1(Train,3),
    closed(Car),
     arg(6,Car,2))).
```

p-Complexity: 18
c-Complexity: 18

Competition 3 Results-Set 2 Rules:

Induce:

[lshape(car1)#triangle][num-car(top=peaked)=0][num-diff-cshape=2]

APC:

[lshape(car1)#triangle][num-car(top=peaked)=0][num-diff-cshape=2]

English:

There are no peaked topped cars, some car has no triangle load and the train has cars of two different car shapes.

Prolog:

```
eastbound(Train) :-
  \+ has_load1(Train,triangle),
  \+ (has_car(Train,C),arg(5,C,peaked)),
  findall(C,(has_car(Train,C1),
             has_car(Train,C2),
              arg(2,C1,S1),
              arg(2,C2,S2),
             S1 \== S2),[_,_]).
```

p-Complexity: 39
c-Complexity: 22

Competition 3 Results-Set 3 Rules:

Induce:

[top(car1)=peaked] OR
[cshape(car1)#crect,ushaped][num-car(cshape=hexagon)=0]

APC:

[top(car1)=peaked] OR
[cshape(car1)#crect or ushaped][num-car(cshape=hexagon)=0]

English:

There is a peaked-topped car, or there is a non-rectangle or u-shaped car and there are no hexagonal cars.

Prolog:

```
eastbound(Train) :-
  has_car(Train,Car),
  (arg(5,Car,peaked);
   \+ (rectangle(Car);u_shaped(Car)),
   \+ (has_car(Train,C),hexagon(C))).
```

p-Complexity: 23
c-Complexity: 20

Competition 3 Results-Set 4 Rules:

Induce:

[forall-car(nwls=2)][lshape(car1)#rectangle][pos(car1)=2]

APC:

```
[forall-car(nwls=2)][lshape(car2)#rectangle]
```

English:
All cars have two wheels and the second car does not have a rectangular load.

Prolog:

```
eastbound(Train) :-
  findall(C,(has_car(Train,C),arg(6,C,2)),List),
   len1(List,N),
   len1(Train,N),
  append([_,Car],_,Train),
  \+ has_load0(Car,rectangle).

 p-Complexity: 31
 c-Complexity: 15
```

Competition 3 Results-Set 5 Rules:

Induce:

```
[lshape(car1)#rectangle,triangle][num-car(cshape=bucket)=0]
[num-car(pos=3)#0]
```

APC:

```
[lshape(car1)#rectangle or triangle][num-car(cshape=bucket)=0]
[num-car(pos=3)#0]
```

English:
There is a third car, no bucket cars, and the load shape of some car is not rectangular or triangular.

```
 Prolog:
eastbound(Train) :-
  has_car(Train,Car),
  \+ (has_load0(Car,rectangle);has_load0(Car,triangle)),
  \+ (has_car(Train,C),bucket(C)),
  append([_,_,_],_,Train).

 p-Complexity: 30
 c-Complexity: 20
```

5.5 Discussion

5.5.1 Problems with Muggleton's Prolog Representation

In a pure sense, Muggleton's Prolog representation of Michalski's trains is not in the same spirit as the trains were originally represented in INDUCE. This argument is subtle, but valid. Michalski's original INDUCE representation for the trains is a clausal representation. That is, each attribute of a car correspondents to a VL_2 predicate. Trains were then represented as a conjunction of all VL_2 predicates related to a given train.

Muggleton also uses a conjunctive representation, but the conjuncts in his representation are cars. Individual characteristics of cars are represented as arguments to the term representing the train's cars. While this distinction is subtle, Muggleton's representation gives rise to certain unsavory artifacts not present in Michalski's original representation. For instance, several of the trains are pictured with no loads. However, in the Prolog representation, while the load quantity might be 0, the load shape would actually be defined as a rectangle or circle. Clearly, this is inconsistent. And in some cases, this artifact caused the inductive learner to learn valid concepts with respect to the training examples, but these concepts would be judged incorrect when translated into Prolog using the given Prolog predicates since the predicate that checked load shape also checked load quantity to ensure that it was greater than zero. In short, the trains generated by Muggleton's train generator were not semantically correct.

This situation would not arise in Michalski's original INDUCE representation of the trains. If a car was not carrying a load, then no load shape predicate would appear in the clause. The deficiency in Muggleton's representation could be patched by adding the symbol none when load shapes were zero. Nevertheless, Muggleton still represents each car as a term in a conjunctive expression. Several arguments could be made that Muggleton's representation is clausal and any confusion is a result of syntactic aberrations, but the term *clause* has a very precise definition in both VL_2 and Prolog. Consequently, in order for Muggleton's trains to be clausal in the sense that Michalski's original trains were clausal, trains should be represented in Prolog as Figure 5.4 illustrates.

```
eastbound    :-
             bucket(car1),
             short(car1),
             …
             has_load(car4,1).
```

Figure 5.4. Clausal representation of a train in Prolog.

5.5.2 Selectivity of Cost Functions

The selectivity of INDUCE cost functions proved inadequate for precisely focusing attention in this problem. One of the features of AQDT that helped focus attention to specific attributes of specific cars was a highly precise cost function. In AQDT, individual attributes of a specific car (e.g., the top of the third car) can be assigned costs. Conversely with INDUCE, this selectivity is not as fine-grained. Costing functions can be set for a specific car (e.g., the third car) and for a specific attribute that ranges over all cars (e.g., the tops of cars), but we cannot cost the top of the third car. It would also be useful to be able to cost the values that individual attributes can take. For instance, there is often a need to make circular load shapes in the third car expensive.

5.5.3 Logical Errors in INDUCE and ASTRA

In spite of this lack of selectivity of the cost function, logical errors exist in INDUCE and ASTRA, a knowledge discovery shell that incorporates the INDUCE program. The following discussion applies primarily to ASTRA since it was used predominantly in the Trains Competition.

Appendix A.1 contains an ASTRA run demonstrating problems with the costing functions. This is using the second set of trains from the third competition in which ASTRA induced a selector stating that the number of different shaped cars in a train was either 2 or 4 (see bold selector). The second parameter screen shows an attempt to remove this selector by high costing both the num-diff function and the cshape attribute and minimizing function cost in the VL_1 phase of execution. In spite of setting the cost for both of these parameters at 10000, ASTRA induced the same rule.

Appendix A.2 contains an ASTRA run demonstrating problems with the *mxpe* parameter, which is a parameter to maximize coverage of positive examples. Given the parameter settings appearing in the screen dump, ASTRA core dumps in the procedure phase2prep, which presumably is a preparation procedure for phase 2 or AQ generalization.

Appendix A.3 clearly demonstrates problems with the forall meta-function. Metamax is set to a high value of 30, which brings in numerous meta-functions, including the forall meta function.

This meta-function basically tries to capture relationships that apply to all train cars. In this particular example, ASTRA included the meta-function forall-car(lshape=triangle)=false, which states that for all cars in a train going east bound, no cars have a triangular load shape. However by inspecting the individual events from event set 1, we see that the first car of the third rule clearly is carrying a triangular load.

Appendix A.4 demonstrates problems with ASTRA's event coverage. This particular run is with the Competition 1 trains. The first rule states that the second car is a bucket-shaped car. This rule is reported to cover event 3. In reality however, the second car of event 3 is a hexagonal car. The rule that covers the reported examples is as follows and was included in the submission:

[cshape(car1)=bucket,hexagon][pos(car1)=2]

The problem could be something as simple as ASTRA not printing out the second internal disjunct. The preceding diatribe on ASTRA's problems is intended to be constructive. ASTRA is an excellent program that is much easier to use for problem-solving than INDUCE because of the level of interactivity ASTRA permits. Consequently, in addition to addressing the above problems, the following are additional recommendations for enhancing ASTRA:

1. Add functions to load parameter sets from a file. INDUCE has this function, but ASTRA does not. During some runs of ASTRA as many as 10 or so parameters were changed. Remembering and changing these parameters when beginning a new day of experiments or when ASTRA crashed is tedious. Being able to load parameters from a file and save parameters to a file, while preserving the level of interactivity for changing parameters, would be a useful and simple feature to add to ASTRA.

2. Replace the old AQ engine in ASTRA with a more modern version. Currently, we believe that ASTRA and INDUCE contain an AQ11 engine and some experiments have identified erroneous results produced by this engine. Incorporating the AQ15c engine into INDUCE and ASTRA might solve some of the existing problems.

3. Port ASTRA and INDUCE to C. Both programs are currently in Pascal (INDUCE also has a LISP implementation). If any future work with these systems is planned, they should be ported to C and modernized.

6. AQ17-HCI

6.1 The Method

The AQ17-HCI system implements the AQ-HCI method for combining an inductive rule learning algorithm with a hypothesis-driven constructive induction (HCI) procedure for iteratively transforming a representation space (Wnek & Michalski, 1994b). In each iteration, the method changes the representation space by adding new attributes determined on the basis of detected patterns, and removing insufficiently relevant attributes. The quality of the hypothesis generated in each iteration is evaluated by applying the hypothesis to a subset of training examples. The set of training examples prepared for a given iteration is split into the primary set (the P set), which is used for generating hypotheses, and the secondary set (the S set), which is used for evaluating the prediction accuracy of the generated hypotheses. Figure 6.1 presents a diagram illustrating the method.

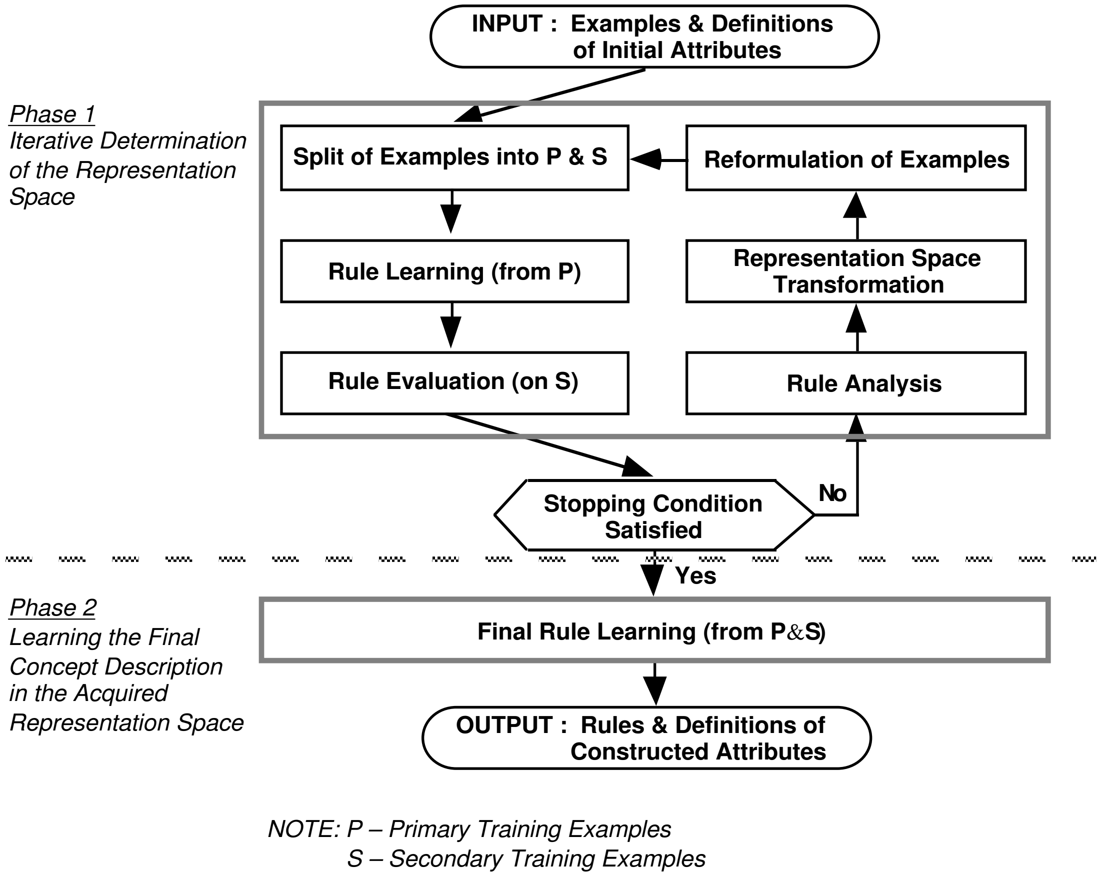


Figure 6.1. The method for Hypothesis-driven Constructive Induction.

The input consists of training examples of one or more concepts, and background knowledge about the attributes used in the examples (which specifies their types and legal value sets). For the sake of simplicity, let us assume that the input consists of positive examples, E^+ and negative examples, E^-, of only one concept. If there are several concepts to learn, examples of each concept are taken as positive examples of that concept, and the set-theoretical union of examples of other concepts is taken as negative examples of that concept.

The method consists of two phases. *Phase 1* determines the representation space by a process of iterative refinement. In each iteration, the method prepares training examples, creates rules, evaluates their performance, modifies the representation space, and then projects the training examples into the new space. This phase is executed until the *Stopping Condition* is satisfied. This condition requires that the prediction accuracy of the learned concept descriptions exceeds a predefined threshold, or there is no improvement of the accuracy over the previous iteration. *Phase 2* determines final concept descriptions in the acquired representation space from the complete set of training examples. The output consists of concept descriptions, and definitions of attributes constructed in Phase 1.

The training set defining problem 1 in the competition was very small. It consisted of 5 positive and 5 negative examples only. Therefore, the system was run without splitting the training examples into P and S sets. All examples were put in the P set, and the S set remained empty. The stopping criterion was formulated accordingly, with addition of the following constraint reflecting the goal of the competition of learning descriptions with minimal complexity. The learning task for AQ17-HCI could be paraphrased in the following way:

> *Beginning with a one-attribute representation space, search for minimal descriptions that cover 100% of training examples. If no such descriptions are found, then add one more attribute into the representation space (up to maximum of 3).*

6.2 Problem Representation

AQ17-HCI was run on an input file similar to AQDT-2 (see Section 4.2). The only modification concerned examples with fewer than four cars. AQ17-HCI, like its predecessor AQ15c, allows fixed-length example definitions only. Since there were eight attributes describing each car, and the maximum number of cars was four, therefore, each train was described using 32 attributes. In shorter trains, the attribute-values of non-existing cars were set to an additional "not applicable" ("na") value. Figure 6.2 shows an example of a shorter train, and its AQ representation.

6.3 Problem Solving Methodology

The original representation space consisted of 48 attributes. Given this space and the set of training examples, the AQ17-HCI system searched for a minimal representation subspace in which the examples could be expressed without ambiguity. There were no single attributes found that would give 100% unambiguous coverage. The system then searched among combinations of two attributes. This search resulted in finding 4 pairs of attributes that describe the input examples in a distinct way (Figure 5.3). Each pair included the c3_LoadShape attribute, and one of c1_TopShape, c1_LoadShape, c2_Shape, and c3_Frame. The whole process of searching for these subspaces together with generation of the four rules took less than 1 second on a Sun Sparcstation-2. From among these rules, the simplest one was selected manually.

Prolog representation

```
eastbound([   c(1,rectangle,short,not_double,flat,2,l(circle,2)),
              c(2,bucket,short,not_double,none,2,l(rectangle,1)),
              c(3,u_shaped,short,not_double,none,2,l(triangle,1))]).
```

AQ17-HCI representation
eastbound-events

c1_Top	c1_Shape	c1_Length	c1_Frame	c1_TopShape	c1_NoWheels	c1_LoadShape	c1_NoLoads
c2_Top	c2_Shape	c2_Length	c2_Frame	c2_TopShape	c2_NoWheels	c2_LoadShape	c2_NoLoads
c3_Top	c3_Shape	c3_Length	c3_Frame	c3_TopShape	c3_NoWheels	c3_LoadShape	c3_NoLoads
c4_Top	c4_Shape	c4_Length	c4_Frame	c4_TopShape	c4_NoWheels	c4_LoadShape	c4_NoLoads
closed	rectangle	short	not_double	flat	2	circle	2
open	bucket	short	not_double	none	2	rectangle	1
open	u_shape	short	not_double	none	2	triangle	1
na	na	na	na	na	na	na	na

Figure 6.2. An example of Prolog and AQ17-HCI descriptions of a three-car train

---

```
  parameters
  run   mode   ambig   trim   wts   maxstar  echo      criteria  verbose
   1    ic     neg     mini   cpx    1       p          default     1
```

Active atts: 5 23 Unambig: 100%

```
  eastbound-outhypo
    #   cpx
    1   [c3_LoadShape=hexagon,triangle]   (t:8, u:8)
    2   [c1_TopShape=peaked,flat] [c3_LoadShape=circle]   (t:2, u:2)

  westbound-outhypo
    #   cpx
    1   [c3_LoadShape=rectangle]   (t:8, u:2)
    2   [c1_TopShape=none,jagged] [c3_LoadShape=rectangle,circle]   (t:8, u:2)
```

Active atts: 7 23 Unambig: 100%

```
  eastbound-outhypo
    #   cpx
    1   [c3_LoadShape=hexagon,triangle]   (t:8, u:8)
    2   [c1_LoadShape=rectangle,utriangle] [c3_LoadShape=circle]   (t:2, u:2)

  westbound-outhypo
    #   cpx
    1   [c1_LoadShape=rectangle,hexagon,circle,triangle] [c3_LoadShape=rectangle]  (t:8, u:6)
    2   [c1_LoadShape=hexagon,circle] [c3_LoadShape=circle]   (t:4, u:2)
```

Active atts: 10 23 Unambig: 100%

```
  eastbound-outhypo
    #   cpx
    1   [c3_LoadShape=hexagon,triangle]   (t:8, u:6)
    2   [c2_Shape=hexagon,bucket]   (t:4, u:2)

  westbound-outhypo
    #   cpx
    1   [c2_Shape=rectangle,u_shaped] [c3_LoadShape=rectangle,circle]   (t:10, u:10)
```

Active atts: 20 23 Unambig: 100%

```
  eastbound-outhypo
    #   cpx
    1   [c3_Frame=not_double] [c3_LoadShape=hexagon,circle,triangle]   (t:8, u:3)
    2   [c3_LoadShape=triangle]   (t:7, u:2)

  westbound-outhypo
    #   cpx
    1   [c3_LoadShape=not_double]   (t:8, u:2)
    2   [c3_Frame=double] [c3_LoadShape=circle]   (t:8, u:2)

System time:      0.750 seconds
```

Figure 6.3. Output produced by AQ17-HCI.

6.4 Results

The AQ17-HCI results are stated in four different forms: 1) AQ17-HCI output, 2) APC syntax (the "Annotated Predicate Calculus" description language), 3) English, and 4) Prolog.

6.4.1. Competition 1 Results

AQ17-HCI:
1 [c3_LoadShape=hexagon,triangle] (t:8, u:6)
2 [c2_Shape=hexagon,bucket] (t:4, u:2)

APC:
[lshape(car3)=hexagon or triangle] OR
[cshape(car2)=hexagon or bucket]

English:
The third car contains a hexagonal or triangular load, or the second car is either hexagon or bucket shaped.

Prolog:
```
eastbound([_, Car2, Car3|_]) :-
  has_load0(Car3, hexagon);
  has_load0(Car3, triangle);
  hexagon(Car2);
  bucket(Car2).
```

P-Complexity**:** 22
C-Complexity**:** 19

6.4.2 Competition 2 Results

The rule obtained for competition 1 was tested against various oracles that scored less or equal 20. Below are results from testing.

Entry	Oracle	Accuracy	Contestant
AQ17-hci	aqdt1	57%	MLIC - AQ17-HCI
	aqdt2	54%	
	inglis	53%	
	mpage	51%	
	pfahr1	53%	
	pfahr2	58%	
	turney	53%	
	weka	53%	
	x	52%	

7. AQ Family Results Versus Those Obtained by Other Programs

The AQ family programs used do not have the ability to involve recursion in the created descriptions. Therefore they were handicapped with regard to the inductive logic programming algorithms that strongly emphasize recursion, and to the Prolog complexity measures that were used in the competition. Nevertheless, the results obtained compare quite favorably with those obtained by many inductive logic programs.

East-West Challenge: Results of Competition 1

Legend:

Entry	is a codeword used by us for scoring purposes
Coverage	is a pair of numbers E/W denoting number of East/West trains covered by the theory
Size	is the complexity score of the theory submitted, as calculated by Ashwin Srinivasan's Prolog program (complex.pl)

Contestant is the name of the contestant

Notes:
1. Entry names prefixed by "comput" were entries received by the British magazine "Computing", in response to Donald Michie's article published on August 4.

2. Multiple entries by the same contestant are suffixed by a number to indicate entry number.

3. Winner is decided on the basis of lowest complexity score.

Entry	Coverage	Size	Contestant
pfahr2	10/0	16	Bernhard Pfahringer
inglis	10/0	19	Stuart Inglis
pfahr1	10/0	19	Bernhard Pfahringer
turney	10/0	19	Peter D Turney
weka	10/0	19	WEKA ML Project
aqdt1	10/0	20	MLIC - AQDT-2
aqdt2	10/0	20	MLIC - AQDT-2

-----------Bottom quartile ends here----------

Entry	Coverage	Size	Contestant
akay	10/0	22	Andrew Kay
aq17hci	10/0	22	MLIC - AQ17-HCI
gamb1	10/0	22	Dragan Gamberger
mli	10/0	22	MLIC - INDUCE
quin	10/0	22	Ross Quinlan
rudy	10/0	23	Rudy Setiono
comput2	10/0	24	Richard Lawrence
comput1	10/0	25	Nicholas Knowles
comput9	10/0	25	T M Bradshaw
comput10	10/0	25	Alan D Cox
comput11	10/0	25	Tony Yule
pfahr3	10/0	27	Bernhard Pfahringer
comput13	10/0	27	Stephane Deom
comput14	10/0	27	Stephane Deom
comput15	10/0	27	R M Yaxley
comput18	10/0	27	Jane Flanders
comput22	10/0	27	Ian Thirkettle
comput24	10/0	27	P Smith
comput25	10/0	27	D P Sayers
comput27	10/0	27	Nick Henfry
comput12	10/0	28	John Brown
comput28	10/0	29	Nick Henfry
comput3	10/0	30	Peter Guy
comput21	10/0	30	A R Archer
vogt	10/0	31	Chris Vogt
comput6	10/0	32	Andrew Davies
comput8	10/0	32	Sue Wood
comput23	10/0	32	David Nelson
mcdon	10/0	38	mcdonald@edu.kestrel
comput4	10/0	42	Bernard Lucas
gamb2	10/0	42	Dragan Gamberger
comput46	0/10	43	R D Scott Westbound rule

-----------Too complex----------

hart	_	_	hart@uk.ac.ox.vax
comput16	_	_	G E Tyack
comput20	_	_	Demetrios Papacharalambous
comput5	_	_	Judy BroadwaY
comput29	_	_	S Roy
comput30	_	_	Melvyn Maltz
comput31	_	_	Mark Henry
comput32	_	_	Kevin Ferriday
comput33	_	_	J Gibbons
comput34	_	_	Donald Mcleod
comput35	_	_	R Millar
comput37	_	_	M White
comput38	_	_	Gianni Pischedda
comput39	_	_	Chris Derry
comput40	_	_	Hans Wrang
comput41	_	_	Peter Young
comput42	_	_	Chris Bergman
comput43	_	_	Peter Young
comput44	_	_	Tim Binney
comput45	_	_	Ian Barker
comput47	_	_	Frank Smith

-----------Inconsistent----------

comput17	_	_	Jane Moch just fits 10 trains
comput19	_	_	Demetrios Papacharalambous
comput26	_	_	Simon Towner
comput29	_	_	Marcus Sean Rebel
comput36	_	_	Dennis Collie

Prolog encoding and English translation of theories in bottom quartile

---

Bernhard Pfahringer (pfahr2)

English:
The train has either a closed last car or a triangular load in a car other than the last one. Also, either the last car is short or the train, after the last car is removed, has the above property.

Prolog:
```
eastbound([Car|Cars]) :-
      (closed(Car);has_load1(Cars, triangle)),
      (short(Car);eastbound(Cars)).
```

P-Complexity**:** 16
C-Complexity**:** 28

Michie-Page effort inspired by pfahr2

English:
There is a short car that either is closed or is somewhere behind a car with a triangular load.

Prolog:
```
eastbound([Car|Cars]):-
       short(Car),
      (closed(Car);has_load1(Cars,triangle));
```

```
        eastbound(Cars).
```

P-Complexity**:** 16
C-Complexity**:** 13

Bernhard Pfahringer (pfahr1)
Stuart Inglis (inglis)
Peter D Turney (turney)
WEKA ML Project (weka)

English:
There are at least three cars, and the second and third cars from the end do not contain the same shape load.

Prolog:

```
eastbound([Car1,Car2,Car3|_]) :-
        has_load0(Car1,_),
        has_load0(Car3,Load),
        not(has_load0(Car2,Load)).
```

P-Complexity**:** 19
C-Complexity**:** 15

Theory X (x)

English:
There is eather a short, closed car, or a car with a circular load somewhere behind a car with a triangular load.

Prolog:

```
eastbound([Car|Cars]):-
        (short(Car), closed(Car));
        (has_load0(Car,circle), has_load1(Cars,triangle));
        eastbound(Cars).
```

P-Complexity**:** 19
C-Complexity**:** 16

MLIC - AQDT-1 (aqdt1)

English:
The third [from the end] car contains a triangular load, or it is not double, while the last car is rectangular and the car inbetween is short.

Prolog:

```
eastbound([Car1,Car2,Car3|_])  :-
            has_load0(Car3,triangle);
            (rectangle(Car1),
            short(Car2),
            not(double(Car3))).
```

P-Complexity**:** 20
C-Complexity**:** 20

MLIC - AQDT-2 (aqdt2)

English:
The third car [from the end] has a triangular load, the second [to last] car is hexagon-shaped, or the last car is rectangularly shaped and the third [from the end] car is closed.

Prolog:

```
eastbound([Car1,Car2,Car3|_]) :-
        has_load0(Car3,triangle);
        rectangle(Car1),
        (closed(Car3);
        hexagon(Car2)).
```

P-Complexity**:** 20
C-Complexity**:** 25

==

East-West Challenge: Results of Competition 2

Legend:

Entry	is a codeword used by us for scoring purposes
Oracle	is a codeword used to denote theories known to us to be within the bottom quartile of complexity scores
Accuracy	is the accuracy of the subsymbolic entry, taking as oracle the theory in the Oracle column
Contestant	is the name of the contestant

Notes:
1. Accuracy is measured as follows. For each oracle/entry pair the following 2x2 table is calculated:

```
                     Oracle class
                  |    East   |   West
            ------------------------------------------
            East  |    N1     |   N2
Entry             |           |
class       ------------------------------------------
            West  |    N3     |   N4
                  |           |
            ------------------------------------------
```

Accuracy = (N1 + N4) / (N1 + N2 + N3 + N4)

2. We know of 9 theories in the bottom quartile of complexity scores relevant to Competition 1. Ordered by complexity score, these are:

Theory	Size	Details
mpage	16	Michie-Page inspired by Theory pfahr2
pfahr2	16	Bernhard Pfahringer
inglis	19	Stuart Inglis
pfahr1	19	Bernhard Pfahringer
turney	19	Peter D Turney
weka	19	WEKA ML Project
x	19	Theory X
aqdt1	20	MLIC - AQDT
aqdt2	20	MLIC - AQDT

3. Winner is decided on the basis of the highest accuracy using any of the theories listed above as oracles. If two or more entries have the same accuracy, then a tie-break rule compares accuracies when using the winner of Competition 1, (Theory pfahr2) as oracle.

Entry	Oracle	Accuracy	Contestant
aq17hci	aqdt1	57%	MLIC - AQ17-HCI
	aqdt2	54%	
	inglis	53%	
	mpage	51%	
	pfahr1	53%	
	pfahr2	58%	
	turney	53%	
	weka	53%	
	x	52%	
aqdt1	aqdt1	100%	MLIC - AQDT-1
	aqdt2	81%	
	inglis	69%	
	mpage	54%	
	pfahr1	69%	
	pfahr2	61%	
	turney	69%	
	weka	69%	
	x	47%	
aqdt2	aqdt1	81%	MLIC - AQDT-2
	aqdt2	100%	
	inglis	60%	
	mpage	57%	
	pfahr1	60%	
	pfahr2	60%	
	turney	60%	
	weka	60%	
	x	52%	
hart	aqdt1	53%	G R Hart
	aqdt2	52%	
	inglis	60%	
	mpage	51%	
	pfahr1	60%	
	pfahr2	56%	
	turney	60%	
	weka	60%	
	x	48%	
imam	aqdt1	69%	Ibrahim F Imam
	aqdt2	66%	
	inglis	70%	
	mpage	63%	
	pfahr1	70%	
	pfahr2	60%	
	turney	70%	
	weka	70%	
	x	58%	
quin	aqdt1	60%	Ross Quinlan
	aqdt2	65%	
	inglis	45%	
	mpage	46%	
	pfahr1	45%	
	pfahr2	55%	

	turney	45%	
	weka	45%	
	x	51%	
turney	aqdt1	43%	Peter D Turney
	aqdt2	48%	
	inglis	52%	
	mpage	85%	
	pfahr1	52%	
	pfahr2	72%	
	turney	100%	
	weka	52%	
	x	90%	

==

East-West Challenge: Results of Competition 3

Legend:

Entry — is a codeword used by us for scoring purposes

Coverage — is a pair of numbers E/W denoting number of East/West trains covered by the theories for each of the 5 subtasks

Size — is the complexity score of the theories submitted, as calculated by Ashwin Srinivasan's Prolog program (complex.pl)

Contestant — is the name of the contestant

Notes:

Winner decided on the basis of lowest total complexity score for the five subtasks.

Entry	Coverage	Size	Contestant
pfahr	5/0	15	Bernhard Pfahringer
	5/0	16	
	5/0	15	
	5/0	14	
	5/0	13	
		74	
dm2	5/0	26	Donald Michie
	0/5	15	(Westbound rule)
	5/0	29	
	0/5	22	(Westbound rule)
	0/5	22	(Westbound rule)
		114	
mli	5/0	18	MLIC - INDUCE
	5/0	56	
	5/0	23	
	5/0	31	
	5/0	30	
		158	
	--------------Below this are inconsistent------------		
aqdt	5/0	16	MLIC - AQDT-2

	5/1	17	has_load(Car,N) wrongly interp as exactly N
	5/0	19	
	0/5	17	Westbound rule
	5/0	16	
		85	
turney	5/0	18	Peter D Turney
	0/2	15	assumes generative infront/3
	0/5	16	Westbound rule
	5/0	17	
	0/5	16	Westbound rule
		82	

Prolog encoding of winning entry (pfahr)

Subtask 1:

English:
The last car is rectangular and closed, or the last car is short and the rest of the train does not have this property.

Prolog:
```
eastbound([A|B]) :-
        (   rectangle(A),
            closed(A)
        ;   short(A),
            not(eastbound(B))
        ).
```

P-Complexity: 15
C-Complexity: 17

Subtask 2:

English:
Either the train does not have a car carrying a triangular load, or the train has a car carrying a rectangular load, and one of its cars is u-shaped.

Prolog:
```
eastbound(A) :-
        (   not(has_load1(A,triangle))
        ;   has_load1(A, rectangle),
            has_car(A, B),
            u_shaped(B)
        ).
```

P-Complexity: 16
C-Complexity: 21

Subtask 3:

English:
Either the train does not have a car carrying a triangular load, or one of its cars is bucket-shaped.

Prolog:
```
eastbound([_|A]) :-
```

```
    (   not(has_load1(A,rectangle))
    ;   has_car(A, B),
        bucket(B)
    ).
```

P-Complexity: 15
C-Complexity: 14

Subtask 4:

English:
The last car is short and not double, and no other car carries a hexagonal load.

Prolog:
```
eastbound([A|B]) :-
    short(A),
    not(double(A)),
    not(has_load1(B,hexagon)).
```

P-Complexity: 14
C-Complexity: 16

Subtask 5:

English:
The third car from the engine is rectangularly shaped.

Prolog:
```
eastbound([A|B]) :-
    rectangle(A),
    (   len1(B, 2)
    ;   eastbound(B)
    ).
```

P-Complexity: 13
C-Complexity: 8

References

Bergadano, F., Matwin, S., Michalski, R.S. and Zhang, J., "Learning Two-tiered Descriptions of Flexible Concepts: The POSEIDON System," *Machine Learning*, Vol. 8, No. 1, pp. 5-43, 1992.

Bloedorn, E., Wnek, J., Michalski, R.S. and Kaufman, K., "AQ17: A Multistrategy Learning System: The Method and User's Guide," *Reports of Machine Learning and Inference Laboratory*, MLI-93-12, Center for Artificial Intelligence, George Mason University, 1993.

Bloedorn, E. and Michalski, R. S., "Data Driven Constructive Induction in AQ17-PRE: A Method and Experiments," *Proceedings of the Third International Conference on Tools for AI,* San Jose, California, November 9-14, 1991a.

Bloedorn, E. and Michalski, R.S., "Constructive Induction from Data in AQ17-DCI: Further Experiments," *Reports of the Machine Learning and Inference Laboratory,* MLI 91-12, Center for Artificial Intelligence, George Mason University, Fairfax, VA, December, 1991b.

Hoff, W., Michalski, R.S. and Stepp, R.E., “INDUCE 3: a program for learning structural descriptions from examples,” *Technical Report*, TR-UIUCDDS-F-86-960, Department of Computer Science, University of Illinois, Urbana, 1986.

Imam, I.F. and Michalski, R.S., “Should Decision Trees be Learned from Examples or from Decision Rules?”, *Lecture Notes in Artificial Intelligence (689)*, Komorowski, J. and Ras, Z.W. (Eds.), pp. 395-404, from the *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems*, ISMIS-93, Trondheim, Norway, June 15-18, Spring Verlag, 1993a.

Michalski, R. S., “Pattern Recognition as Rule-guided Inference,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI, Vol.2, pp. 349–261, 1980.

Michalski, R.S. “A Theory and Methodology of Inductive Learning,” *Artificial Intelligence*, Vol. 20, pp. 111-116, 1983.

Michalski, R.S., Mozetic, I., Hong, J. and Lavrac, N., “The Multi-Purpose Incremental Learning System AQ15 and Its Testing Application to Three Medical Domains,” *Proceedings of AAAI-86*, pp. 1041-1045, Philadelphia, PA, 1986.

Michalski, R.S. and Imam, I.F., “Learning Problem-Optimized Decision Trees from Decision Rules: The AQDT-2 System”, *Lecture Notes in Artificial Intelligence*, Spring Verlag, from the *8th International Symposium on Methodologies for Intelligent Systems, ISMIS*, Charlotte, North Carolina, October 16-19, 1994.

Thrun, S.B., Bala, J., Bloedorn, E., Bratko, I., Cestnik, B., Cheng, J., De Jong, K.A., Dzeroski, S., Fahlman, S.E., Hamann, R., Kaufman, K., Keller, S., Kononenko, I., Kreuziger, J., Michalski, R.S., Mitchell, T., Pachowicz, P., Vafaie, H., Van de Velde, W., Wenzel, W., Wnek, J. and Zhang, J., "The MONK's problems: A Performance Comparison of Different Learning Algorithms," *Computer Science Reports*, CMU-CS-91-197, Carnegie Mellon University, Pittsburgh, PA, December l991.

Wnek, J. and Michalski, R.S., "Comparing Symbolic and Subsymbolic Learning: Three Studies," in *Machine Learning: A Multistrategy Approach*, Vol. 4., R.S. Michalski and G. Tecuci (Eds.), Morgan Kaufmann, San Mateo, CA, 1994a.

Wnek, J. and Michalski, R.S., "Hypothesis-driven Constructive Induction in AQ17-HCI: A Method and Experiments," *Machine Learning*, Vol. 14, No. 2, pp. 139-168, 1994b.

Wnek, J., Kaufman, K., Bloedorn, E. and Michalski, R.S., "Selective Induction Learning System AQ15c: The method and user's guide," *Reports of Machine Learning and Inference Center*, MLI 95-04, Center for MLI, George Mason University, 1995.

APPENDIX A: Sample Runs Demonstrating ASTRA's Logical Errors

Appendix A.1: Problems with the Cost Function

```
File: set2.astra from competition 3
________________________ ALL PARAMETERS SUMMARY ______________________________
 g1 =  2 (grmaxrules)   |-------------------------|TRACES|FUNCTION  ftype  fcost
 g2 =  2 (alter)        |  e1 =  2   (exmaxrules) | t1=i | NAME      (f1)   (f2)
 g3 =  4 (nconsist)     |  e2 =false (exstrip)    | t2=i | num-p's    lin      1
 g4 =  1 (regenstar)    |  e3 = 20   (excutoff)   | t3=i | num-part   lin      1
 gn =  4 (grnumcrit)    |  en =  2   (exnumcrit)  | t4=i | num-diff   lin      1
gc(1)=mnfc  gt(1)=   0  | ec(1)=mxpa  et(1)=   0  | t5=i |
gc(2)=mxpa  gt(2)=   0  | ec(2)=mnsl  et(2)=   0  | t6=i |
gc(3)=mnsl  gt(3)=   0  | ec(3)=      et(3)=      | t7=i |
gc(4)=mnna  gt(4)=   0  | ec(4)=      et(4)=      | t8=i |
gc(5)=      gt(5)=      | ec(5)=      et(5)=      | t9=i |
gc(6)=      gt(6)=      | ec(6)=      et(6)=      |STOPS |
gc(7)=      gt(7)=      | ec(7)=      et(7)=      | s1=i |
-----------------------|-------------------------| s2=i |
 r1 =    4  (metamax)   |  l1 = 5  (lrmax)        | s3=i |
 r2 = false (endpoint)  |  ln = 4  (lrnumcrit)    | s4=i |
 r3 = false (equal)     | lc(1)=mxpa  lt(1)=   0  | s5=i |
-----------------------| lc(2)=mnna  lt(2)=   0  | s6=i |
 c1 = disc  (gentype)   | lc(3)=mxls  lt(3)=   0  | s7=i |
 c2 =  100  (mincover)  | lc(4)=mnrs  lt(4)=   0  | s8=i |
 c3 =    0  (maxback)   | lc(5)=      lt(5)=      | s9=i |

THE FOLLOWING RULE COVERS SET  1:

This rule covers event(s) 5 4 3 2 1  (0 new) :
   Rule 42:
[lshape(car1)#triangle][num-car(top=peaked)=0][num-diff-cshape=2,4]
           Costs: 1. (mnfc) =   3
                  2. (mxpa) =   5
                  3. (mnsl) =   3
                  4. (mnna) =   0

The selected meta-selectors are:
 ms  type          function          value  poscov negcov
  1  num-diff  cshape                    2       5      2
  2  num-p's   cshape    = ellipse       0       5      3
  3  num-p's   top       = arc           0       5      3
  4  num-p's   top       = peaked        0       5      3

________________________ ALL PARAMETERS SUMMARY ______________________________
 g1 =  2 (grmaxrules)   |-------------------------|TRACES|FUNCTION  ftype  fcost
 g2 =  2 (alter)        |  e1 =  2   (exmaxrules) | t1=i | NAME      (f1)   (f2)
 g3 =  4 (nconsist)     |  e2 =false (exstrip)    | t2=i | num-p's    lin      1
 g4 =  1 (regenstar)    |  e3 = 20   (excutoff)   | t3=i | num-part   lin      1
 gn =  4 (grnumcrit)    |  en =  2   (exnumcrit)  | t4=i | num-diff lin 10000
gc(1)=mnfc  gt(1)=   0  | ec(1)=mxpa  et(1)=   0  | t5=i | cshape   nom 10000
gc(2)=mxpa  gt(2)=   0  | ec(2)=mnsl  et(2)=   0  | t6=i |
gc(3)=mnsl  gt(3)=   0  | ec(3)=      et(3)=      | t7=i |
gc(4)=mnna  gt(4)=   0  | ec(4)=      et(4)=      | t8=i |
gc(5)=      gt(5)=      | ec(5)=      et(5)=      | t9=i |
gc(6)=      gt(6)=      | ec(6)=      et(6)=      |STOPS |
gc(7)=      gt(7)=      | ec(7)=      et(7)=      | s1=i |
-----------------------|-------------------------| s2=i |
 r1 =    4  (metamax)   |  l1 = 5  (lrmax)        | s3=i |
 r2 = false (endpoint)  |  ln = 4  (lrnumcrit)    | s4=i |
 r3 = false (equal)     | lc(1)=mxpa  lt(1)=   0  | s5=i |
-----------------------| lc(2)=mnna  lt(2)=   0  | s6=i |
 c1 = disc  (gentype)   | lc(3)=mxls  lt(3)=   0  | s7=i |
 c2 =  100  (mincover)  | lc(4)=mnrs  lt(4)=   0  | s8=i |
 c3 =    0  (maxback)   | lc(5)=      lt(5)=      | s9=i |
```

```
THE FOLLOWING RULE COVERS SET  1:

This rule covers event(s) 5 4 3 2 1  (0 new) :
   Rule 58:
[lshape(car1)#triangle][num-car(top=peaked)=0][num-diff-cshape=2,4]
           Costs: 1. (mnfc) = 10002
                  2. (mxpa) =   5
                  3. (mnsl) =   3
                  4. (mnna) =   0

The selected meta-selectors are:
 ms  type          function          value  poscov negcov
  1  num-diff  cshape                    2       5      2
  2  num-p's   cshape    = ellipse       0       5      3
  3  num-p's   top       = arc           0       5      3
  4  num-p's   top       = peaked        0       5      3
```

Appendix A.2: Problems with the mxpe Parameter

```
file: set3.astra
________________________  ALL PARAMETERS SUMMARY ______________________________
 g1 =  2 (grmaxrules)   |-------------------------|TRACES|FUNCTION  ftype  fcost
 g2 =  2 (alter)        |  e1 =  2    (exmaxrules) | t1=i | NAME      (f1)   (f2)
 g3 =  4 (nconsist)     |  e2 =false (exstrip)     | t2=i | num-p's    lin      1
 g4 =  1 (regenstar)    |  e3 = 20   (excutoff)    | t3=i | num-part   lin      1
 gn =  4 (grnumcrit)    |  en =  3   (exnumcrit)   | t4=i | num-diff   lin      1
gc(1)=mnna  gt(1)=   0 | ec(1)=mxpe  et(1)=   0 | t5=i |
gc(2)=mxpn  gt(2)=   0 | ec(2)=mnsl  et(2)=   0 | t6=i |
gc(3)=mnsl  gt(3)=   0 | ec(3)=mnvc  et(3)=   0 | t7=i |
gc(4)=mnfc  gt(4)=   0 | ec(4)=      et(4)=      | t8=i |
gc(5)=      gt(5)=     | ec(5)=      et(5)=      | t9=i |
gc(6)=      gt(6)=     | ec(6)=      et(6)=      |STOPS |
gc(7)=      gt(7)=     | ec(7)=      et(7)=      | s1=i |
-----------------------|-------------------------| s2=i |
 r1 =    4  (metamax)  |  l1 = 5  (lrmax)        | s3=i |
 r2 = false (endpoint) |  ln = 4  (lrnumcrit)    | s4=i |
 r3 = false (equal)    | lc(1)=mxpa  lt(1)=   0  | s5=i |
-----------------------| lc(2)=mnna  lt(2)=   0  | s6=i |
 c1 = disc  (gentype)  | lc(3)=mxls  lt(3)=   0  | s7=i |
 c2 =  100  (mincover) | lc(4)=mnrs  lt(4)=   0  | s8=i |
 c3 =    0  (maxback)  | lc(5)=      lt(5)=      | s9=i |
```

Core dumps in phase2prep().

Appendix A.3: Problems with the forall Meta-Function

```
_______________________  ALL PARAMETERS SUMMARY _____________________________
 g1 =  2 (grmaxrules)   |-------------------------|TRACES|FUNCTION  ftype  fcost
 g2 =  2 (alter)        |  e1 =  2   (exmaxrules) | t1=i |  NAME     (f1)   (f2)
 g3 =  4 (nconsist)     |  e2 =false (exstrip)    | t2=i |  num-p's   lin      1
 g4 =  1 (regenstar)    |  e3 = 20   (excutoff)   | t3=i |  num-part  lin      1
 gn =  4 (grnumcrit)    |  en =  3   (exnumcrit)  | t4=i |  num-diff  lin      1
gc(1)=mnna  gt(1)=   0  | ec(1)=mxpn  et(1)=   0  | t5=i |
gc(2)=mxpn  gt(2)=   0  | ec(2)=mnsl  et(2)=   0  | t6=i |
gc(3)=mnsl  gt(3)=   0  | ec(3)=mnvc  et(3)=   0  | t7=i |
gc(4)=mnfc  gt(4)=   0  | ec(4)=      et(4)=      | t8=i |
gc(5)=      gt(5)=      | ec(5)=      et(5)=      | t9=i |
gc(6)=      gt(6)=      | ec(6)=      et(6)=      |STOPS |
gc(7)=      gt(7)=      | ec(7)=      et(7)=      | s1=i |
------------------------|-------------------------| s2=i |
 r1 =   30  (metamax)   |  l1 = 5  (lrmax)        | s3=i |
 r2 = false (endpoint)  |  ln = 4  (lrnumcrit)    | s4=i |
 r3 = false (equal)     | lc(1)=mxpa  lt(1)=   0  | s5=i |
------------------------| lc(2)=mnna  lt(2)=   0  | s6=i |
 c1 = disc  (gentype)   | lc(3)=mxls  lt(3)=   0  | s7=i |
 c2 =  100  (mincover)  | lc(4)=mnrs  lt(4)=   0  | s8=i |
 c3 =    0  (maxback)   | lc(5)=      lt(5)=      | s9=i |

THE FOLLOWING RULE COVERS SET  1:

This rule covers event(s) 5 4 3 2 1  (5 new) :
   Rule 102:
[forall-car(lshape=triangle)=false][num-car(lshape=hexagon)=0]
[num-car(lshape=diamond)=0][num-car(top=arc)=0]
           Costs: 1. (mnna) =   0
                  2. (mxpn) =   5
                  3. (mnsl) =   4
                  4. (mnfc) =   4

#ERROR  This car obviously has a triangle load:
   Rule 3: Event set: 1
[cshape(car1)=ushaped][cshape(car2)=crect][double(car1)][double(car2)]
[forall-car(pos=3)=false][forall-car(top=flat)=false]
[forall-car(top=arc)=false][forall-car(top=peaked)=false]
[forall-car(pos=4)=false][forall-car(double=false)][forall-car(nwls=3)=false]
[forall-car(ln=long)=false][forall-car(lqty=2)=false][forall-car(lqty=3)=false]
[forall-car(lshape=rectangle)=false][forall-car(cshape=ushaped)=false]
[forall-car(lshape=utriangle)=false][forall-car(lshape=hexagon)=false]
[forall-car(lshape=diamond)=false][forall-car(cshape=bucket)=false]
[forall-car(cshape=crect)=false][forall-car(lshape=triangle)=false]
[forall-car(cshape=ellipse)=false][forall-car(pos=2)=false][infront(car1,car2)]
[ln(car1)=short][ln(car2)=short][lqty(car1)=1][lqty(car2)=1]
[lshape(car1)=triangle][lshape(car2)=rectangle][num-car(cshape=ellipse)=0]
[num-car(lshape=diamond)=0][num-car(lqty=3)=0][num-car(lshape=hexagon)=0]
[num-car(pos=2)=1][num-car(pos=1)=1][num-car(top=arc)=0][num-car(top=peaked)=0]
[num-diff-cshape=2][num-diff-double=1][nwls(car1)=2][nwls(car2)=2][pos(car1)=1]
[pos(car2)=2][top(car1)=none][top(car2)=flat] => [d=1]
```

Appendix A.4: Problems with Event Coverage

```
________________________  ALL PARAMETERS SUMMARY ____________________________
 g1 = 10 (grmaxrules)   |-------------------------|TRACES|FUNCTION  ftype  fcost
 g2 = 10 (alter)        |  e1 = 10    (exmaxrules) | t1=i | NAME      (f1)   (f2)
 g3 = 10 (nconsist)     |  e2 = true (exstrip)     | t2=i | num-p's    lin      1
 g4 = 10 (regenstar)    |  e3 = 20    (excutoff)   | t3=i | num-part   lin      1
 gn =  3 (grnumcrit)    |  en =  3    (exnumcrit)  | t4=i | num-diff   lin      1
gc(1)=mxpn  gt(1)=   0 | ec(1)=mxpn  et(1)=   0  | t5=i |
gc(2)=mnsl  gt(2)=   0 | ec(2)=mnsl  et(2)=   0  | t6=i |
gc(3)=mnna  gt(3)=   0 | ec(3)=mnna  et(3)=   0  | t7=i |
gc(4)=      gt(4)=     | ec(4)=      et(4)=      | t8=i |
gc(5)=      gt(5)=     | ec(5)=      et(5)=      | t9=i |
gc(6)=      gt(6)=     | ec(6)=      et(6)=      |STOPS |
gc(7)=      gt(7)=     | ec(7)=      et(7)=      | s1=i |
-----------------------|-------------------------| s2=i |
 r1 =    0  (metamax)   |  l1 = 5  (lrmax)         | s3=i |
 r2 = false (endpoint)  |  ln = 4  (lrnumcrit)     | s4=i |
 r3 = false (equal)     | lc(1)=mxpa  lt(1)=   0  | s5=i |
-----------------------| lc(2)=mnna  lt(2)=   0  | s6=i |
 c1 = disc  (gentype)   | lc(3)=mxls  lt(3)=   0  | s7=i |
 c2 =  100  (mincover)  | lc(4)=mnrs  lt(4)=   0  | s8=i |
 c3 =    0  (maxback)   | lc(5)=      lt(5)=      | s9=i |

THE FOLLOWING RULES COVER SET  1:

This rule covers event(s) 8 6 3 2  (2 new) :
   Rule 7116:
[cshape(car1)=bucket][pos(car1)=2]
           Costs: 1. (mxpn) =   2
                  2. (mnsl) =   2
                  3. (mnna) =   0
#
In reality the above rule does not cover event 3, since the car shape for that car is
actually hexagon.
#

This rule covers event(s) 10 9 7 6 5 4 2 1  (8 new) :
   Rule 6780:
[lshape(car1)=hexagon,triangle][pos(car1)=3]
           Costs: 1. (mxpn) =   8
                  2. (mnsl) =   2
                  3. (mnna) =   0
```

Appendix A.5: Prolog Complexity Tester

```
%%%%%  Ashwin Srinivasan's Prolog code
%%%%%  for measuring theory complexity.


% dynamic statements are only for compiled Prologs
% (not Clocksin and Mellish standard)

:- dynamic counts/2.

% count clauses, literals and terms in file FileName
count(FileName):-
        reset_counts,
        see(FileName),
        count_clauses,
        seen,
        print_theory_counts.

count_clauses:-
        repeat,
        read(Clause),
        count_clause(Clause),
        Clause = end_of_file,
        !.

count_clause(end_of_file):- !.
count_clause((Head:-Body)):-
        !,
        inc(clauses,1),
        get_litterm_count((Head,Body)).
count_clause(UnitClause):-
        inc(clauses,1),
        get_litterm_count(UnitClause).


get_litterm_count((LitTerm;LitTerms)):-
        !,
        inc(litterms,1),              % for ';'/2
        get_litterm_count(LitTerm),
        get_litterm_count(LitTerms).
get_litterm_count((LitTerm,LitTerms)):-    % no charge for ','/2
        !,
        get_litterm_count(LitTerm),
        get_litterm_count(LitTerms).
get_litterm_count(LitTerm):-
        inc(litterms,1),              % for Lit
        functor(LitTerm,Name,Arity),
        get_arg_count(LitTerm,Arity,0,T0),
        inc(litterms,T0).

get_arg_count(_,0,LT,LT).
get_arg_count(Expr,Arg,T,LitTerms):-
        arg(Arg,Expr,Term),
        var(Term), !,
        Arg0 is Arg - 1,
        T1 is T + 1,
        get_arg_count(Expr,Arg0,T1,LitTerms).
get_arg_count(Expr,Arg,T,LitTerms):-
        arg(Arg,Expr,LitTerm),
        functor(LitTerm,LitTermName,LitTermArity),
        inc_term_count(LitTermName/LitTermArity,T,T1),
        get_arg_count(LitTerm,LitTermArity,T1,T2),
        Arg0 is Arg - 1,
```

```
        get_arg_count(Expr,Arg0,T2,LitTerms).

inc_term_count(',',/2,T,T):-         % no charge for ','/2
        !.
inc_term_count(_,T,T1):-
        T1 is T + 1.

reset_counts:-
        retractall(counts(_,_)),
        asserta(counts(clauses,0)),
        asserta(counts(litterms,0)).

print_theory_counts:-
        counts(clauses,C),
        write('clauses:'), write(C), nl,
        counts(litterms,LT),
        write('lits+terms:'), write(LT), nl, nl,
        Total is C + LT,
        write('total:'), write(Total), nl.


inc(Parse,N):-
        retract(counts(Parse,N1)),
        N0 is N1 + N,
        asserta(counts(Parse,N0)).
```

APPLYING AQ TO THE EAST-WEST CHALLENGE

For a possbile journal paper:

HOW DID AQ FACE WEST-EAST CHALLENGE?

Results and Lessons from the the 2nd International Competition of Machine Learning Programs