INDUCTIVE LEARNING SYSTEM
AQ15c:
The Method and User's Guide

Janusz Wnek, Kenneth Kaufman
Eric Bloedorn, Ryszard S. Michalski

MLI 95-4

March 1995

INDUCTIVE LEARNING SYSTEM AQ15c:
The Method and User's Guide

Janusz Wnek
Ken Kaufman
Eric Bloedorn
Ryszard S. Michalski

Machine Learning and Inference Laboratory
George Mason University
Fairfax, VA 22030

October 1995

INDUCTIVE LEARNING SYSTEM AQ15c: The Method and User's Guide

Abstract

AQ15c is a system for acquiring decision or classification rules from examples and counterexamples and/or from previously learned decision rules. When learning rules, AQ15c uses 1) background knowledge in the form of rules (input hypotheses), 2) the definition of descriptors and their types and 3) a rule preference criterion that evaluates competing candidate hypotheses. Each training example characterizes an object, and its class-label specifies the correct decision associated with that object. The generated decision rules are expressed as symbolic descriptions involving relations between objects' attribute values. Rule generation is guided by a user-defined rule-preference criterion. The user-defined criterion ranks the importance and tolerance of a number of measures of rule quality including rule complexity, cost and coverage. AQ15c is a C language re-implementation of AQ15 (Hong, Mozetic, Michalski, 1986) written in Pascal. This version can handle larger datasets, is more robust and portable. Versions of AQ15c have so far been compiled for the Sun Solaris (1.1), IBM-compatible (DOS 6.0), and Apple (MacOS 7.5) platforms. In addition the testing facilities have been expanded to include three different measures of match.

Key words: Concept learning, Inductive inference, Learning from examples, Constructive induction

Acknowledgements

The authors thank Jim Ribeiro and Witold Szczepanik for their help with the ANSI C implementation. Witold also contributed to Appendix describing the new data structures for AQ15c.

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

Table of Contents

1. Introduction

1.1 Concept Learning by Induction

Among the fundamental characteristics of intelligent behavior are the abilities to pursue goals and to plan future actions. To exhibit these characteristics, an intelligent system - human or machine - must be able to classify objects, behaviors as equivalent for achieving given goals, and some others as differing. For example, to satisfy hunger, an animal must be able to classify some objects as edible, despite the great variety of their forms and despite the changes they undergo in the environment. Thus, an *intelligent system* must be able to form concepts, that is, classes of entities united by some principle. Such a principle might be a common use or goal, the same role in a structure forming a theory about something, or just similar perceptual characteristics. In order to use the concepts, the system must also develop efficient methods for recognizing concept membership of any given entity. The question then is how concept recognition methods are developed.

The study and computer modeling of processes by which an intelligent system acquires, refines and differentiates concepts is the subject matter of *concept learning* (Michalski, 1986). In research on concept learning, the term concept is usually viewed in a more narrow sense, namely, as an equivalence class of entities, such that it can be comprehensibly described by a small set of statements. The description must be sufficient for distinguishing this concept from other concepts. Individual entities in this equivalence class are called *instances* of the concept.

In any learning process a learner applies the knowledge already possessed to information obtained from a source, e.g., a teacher, in order to derive new useful knowledge. This new knowledge is then stored for subsequent use. Learning a new concept can proceed in a number of ways, reflecting the type of inference the learner performs on the information supplied. For example, one may learn the concept of a *friend* by being given a description of him/her, by generalizing examples of specific people, by constructing this concept in the process of observing and analyzing different types of people, or by some other method. The type of inference performed by the learner on the information supplied defines the *strategy* of concept learning, and constitutes a useful criterion for classifying learning processes.

One basic concept learning strategy is *learning by induction*. In this strategy, the learner acquires a concept by drawing inductive inferences from supplied facts or observations. Depending on what is provided and what is known to the learner, two different forms of this strategy can be distinguished: learning from examples and learning from observation and discovery. In *learning from examples*, the learner induces a concept description by generalizing from provided examples and counter-examples of the concept. It is assumed that the concept is already known to the teacher. The task for the learner is to determine a general concept description by analyzing individual concept examples.

1.2 Concept Representation

A concept learning task is strongly dependent on the concept representation space and the representational formalism. A *concept representation space* is the set of all descriptors used in describing the concept. A *representational formalism* defines ways (syntax) of constructing descriptions from descriptors. An example of a concept representation space is the *feature space*, in which descriptors are attributes with predefined sets of values. Considering attributes as dimensions spanning a multidimensional space, concept instances map to points in this space. An example of a representational formalism is predicate calculus with its set of logical operators.

AQ15c is a program that learns concept descriptions from examples. It uses the Variable-Valued Logic system VL1 as its representational formalism which defines representation spaces in terms of

attribute sets (the attributes may have multiple, discrete values). The representation space and the set of concepts for learning have to be defined by a teacher.

Quinlan (1993) lists some of the elements that are required by systems such as AQ15c and Quinlan's C4.5 in order to perform learning from examples. These include:

- *Attribute-value descriptions*, in which each example is described in terms of a set of attributes that is identical throught the example set.
- *Predefined classes*, by which the goal concepts are known and the training examples can be classified prior to learning.
- *Discrete classes*, implying that for any possible example, an oracle can determine with certainty whether or not it belongs to a particular class.
- *Sufficient data* to allow the learning mechanism to detect significant patterns and to discount much of what is likely noise.
- *"Logical" classification models*, in which output knowledge may describe concepts in terms of sets of attributes and values.

Another important aspect of intelligent learning systems is a set of *preference criteria* that can be evaluated and compared, such that the learner can choose wisely from among the many concept descriptions that are equally consistent with the input examples. AQ15c is equipped both with a set of default criteria that should ensure satisfactory performance in many learning tasks and the capacity for the user to enter different sets of criteria custom-designed for a given application.

1.3 AQ15c Implementation

AQ15c acquires decision or classification rules from examples and counter-examples and/or from previously learned decision rules. When learning rules, AQ15c uses 1) background knowledge in the form of rules (input hypotheses), 2) the definition of descriptors and their types and 3) a rule preference criterion that evaluates competing candidate hypotheses. Each training example characterizes an object, and its class-label specifies the correct decision associated with that object. The generated decision rules are expressed as symbolic descriptions involving relations between objects' attribute values. The program performs a heuristic search through a space of logical expressions, until it finds a decision rule that best satisfies the preference criterion while covering all positive examples, but no negative examples. The program implements the STAR method of inductive learning (Michalski, Larson 1983). It is based on the AQ algorithm for solving the general covering problem (Michalski, 1969).

The AQ15c program is an immediate descendent of AQ15 (Hong, Mozetic, Michalski, 1986). The program AQ15 was written in Pascal as an enhancement of the GEM (Generalization of Examples by Machine) program written by Bob Stepp and Mike Stauffer. GEM was written from scratch on the Cyber 175; it was not a modified version of the AQ7-AQ11 series of programs (Michalski 1969; Michalski and Larson 1975, 1978, 1983).

AQ15c includes a number of new features that extend the functionality of previous versions. One change from previous versions is the language in which it is written. AQ15c has been ported to ANSI-C. The most important advantage of this change is that it reduces the need for limiting, system-defined data structures. In particular, the Pascal set structure, which under Sun Pascal was

limited to a cardinality of 58, was very restrictive as it forced upon the programmers a complex way of handling sets, and limited attributes in the data sets to 58 values. The ANSI-C version has no preset limitations on the number of variables, the number of values per variable, or the number of classes. In addition, the ANSI-C version is on average six times faster than the previous implementation, and provides better error diagnostics.

Another important result of the port from Pascal to ANSI-C was the ability to easily port the program to different platforms. The current AQ15c version has been ported to Sun Solaris, IBM-compatible (DOS 6.0), and Apple (MacOS 7.5) platforms. More platforms can be supported in the future with no change in the underlying source code because of the use of ANSI-C.

One new constraint in AQ15c is the requirement that the declarations of types and variables precede input examples and rules. This requirement is similar to those found in some programming languages where variables must be declared before they are used. The general structure and order of the input file is as follows:

- Parameters controlling the learning process (parameters, criteria-table)
- Definitions of knowledge types (domaintypes, names tables)
- Attribute declarations (variables table)
- Background knowledge in the form of rules (inhypo tables)
- Training data (events tables)
- Testing data (tevents tables)

1.4 Example: Conceptual Design of Wind Bracings

Let us consider a problem from the area of conceptual design of wind bracings in steel skeleton structures of tall buildings. The objective is to find decision rules which would assist a designer during the conceptual design stage. These rules represent the relationships among attributes describing the design requirements to be met, the possible structural design decisions, and an assumed quality criterion (in our case - the unit steel weight). The quality criterion is considered as a target concept. Decision rules sought are expected to be design rules which would show how various structural design decisions taken under different combinations of design requirements would result in one of four values of the quality criterion. Therefore, four categories of design rules are sought, associated with the quality criterion.

The representation space consists of seven multivalued attributes: number of stories, bay length, wind intensity factor, type of joints, number of braced bays, number of vertical trusses, and number of horizontal trusses. The first three attributes are used to define design requirements, while the remaining attributes are used to characterize structural design decisions available. Classification of design examples into the four categories of designs was done according to the relative unit steel weight. Accordingly, designs related to "low weight" are called "recommendation rules," those related to the category "medium weight" were called "standard rules," and those related to the category "high weight" were called "avoidance rules." All rules related to the category "infeasible" were called "infeasibility rules" since they represent relationships among attributes which occur in the case when it is impossible to design a wind bracing of a given type under assumed design conditions. More details of the design problem considered and the representation space used are provided in (Arciszewski et al. 1994).

Figure 1 shows an example of an input file to AQ15c for the wind bracing file. The first thing to notice is that the input is organized in a series of *tables*, each of which describe an aspect of the problem for the program. The first table shown, the **parameters** table, instructs AQ15c in how to go about the learning process, how to choose among different plausible conclusions, and what information to report to the user on the completion of the run. The **domaintypes** and **variables** tables inform the program about the set of attributes to be considered, while the **names** tables

define the different values for each of the attributes. Finally, the **events** tables list the training examples from which AQ15c will induce its hypotheses.

```
parameters
  run   mode   ambig   trim   wts   maxstar  echo   criteria  verbose
   1    ic     neg     mini   cpx    10      pdnv   default     1

domaintypes
    type   levels   name
    lin       5     stories
    nom       2     bayLength
    nom       2     windIntensity
    nom       3     typeOfJoints
    lin       3     numberOfBays
    lin       4     numberOfTrusses

stories-names
    value   name
      0     6
      1     12
      2     18
      3     24
      4     30

bayLength-names
    value   name
      0     20
      1     30

windIntensity-names
    value   name
      0     07
      1     11

typeOfJoints-names
    value   name
      0     rigid
      1     hinged
      2     mixed

numberOfBays-names
    value   name
      0     1
      1     2
      2     3

numberOfTrusses-names
    value   name
      0     0
      1     1
      2     2
      3     3

variables
    #      name
    1    NS.stories
    2    BL.bayLength
```

```
    3   WI.windIntensity
    4   JO.typeOfJoints
    5   BA.numberOfBays
    6   VT.numberOfTrusses
    7   HT.numberOfTrusses


Avoid-events
 NS BL WI JO      BA VT HT
  6 30 11  mixed 2  0  2
  6 30 11  mixed 2  0  1
  6 20 07  mixed 2  0  1
  6 20 11  rigid 1  0  0
  6 30 11  rigid 1  0  0
...

Standard-events
 NS BL WI JO      BA VT HT
 12 20 07  rigid 3  0  0
 30 30 11  rigid 3  0  0
 18 30 07  mixed 2  0  3
  6 30 11 hinged 1  1  2
 12 20 07  mixed 1  0  2
...

Recommended-events
 NS BL WI JO      BA VT HT
 18 30 07 hinged 1  1  2
 12 30 07 hinged 1  1  2
 12 30 11 hinged 1  1  2
  6 30 11 hinged 2  2  0
 24 30 11 hinged 1  1  3
...

Infeasible-events
 NS BL WI  JO     BA VT HT
 30 30 11  mixed 1  0  1
 30 30 07  mixed 1  0  3
 30 30 11  mixed 1  0  3
 30 30 07  mixed 1  0  1
 30 30 07  rigid 1  0  0
...
```

Figure 1. AQ15c input file for the Wind Bracing problem

Figure 2 shows a portion of the output generated by running AQ15c on the input file shown in Figure 1. The **outhypo** tables show the rulesets learned for each of the four decision classes. The rules for recommended designs, for example, state that either (1) the number of stories should be between 12 and 30 and the number of trusses should be between 1 and 3 or (2) the number of bays should be between 2 and 3 while the number of trusses is between 1 and 3.

The numbers after each rule inform the user about how many of the input examples that particular rule describes. For example, the first recommended rule applies to 106 input examples of the recommended class and applies uniquely (i.e., no other recommended rule applies to them) to 49

of the input examples. Similarly, 77 examples of recommended design are satisfied by the second rule for that class, 20 uniquely.

One of the columns in the input **parameters** table, **trim**, directs AQ15c as to how general or specific the chosen output rule should be. The "mini" value shown in Figure 1 instructs the program to output rules as short and simple as possible; Figure 2's output was generated with this setting. Compare those rules to those shown in Figure 3, in which the only change in the input was to change the trim value to "spec" — a request to select very detailed rules with as many specific conditions as possible. Yet both rule sets are equally consistent with respect to the input data. One of AQ15c's strengths is its ability to respond to the requests of the user representing the requirements of different learning problems.

Detailed explanations and instructions on setting up input tables and parameters are given in Section 4.2.

```
Avoid-outhypo
#    cpx
1    [NS=6] [JO=rigid,mixed]    (t:15, u:15)


Standard-outhypo
#    cpx
1    [NS=12..24] [VT=0]    (t:103, u:103)
2    [NS=30] [BA=2..3] [VT=0]    (t:11, u:11)
3    [NS=6] [BA=1] [VT=1]    (t:8, u:8)


Recommended-outhypo
#    cpx
1    [NS=12..30] [VT=1..3]    (t:106, u:49)
2    [BA=2..3] [VT=1..3]    (t:77, u:20)


Infeasible-outhypo
#    cpx
1    [NS=30] [JO=rigid,mixed] [BA=1]    (t:8, u:8)


Learning system time:       0.267 seconds
Learning user time:         0.00  seconds
```

Figure 2. AQ15c output generated from the input file shown in Figure 1

```
parameters
  run    mode    ambig    trim    wts    maxstar    echo    criteria    verbose
   1     ic      neg      spec    cpx     10        p       default       1


  Avoid-outhypo
    #    cpx
    1    [NS=6] [JO=rigid,mixed] [BA=1..2] [VT=0]    (t:15, u:15)
```

```
Standard-outhypo
  #    cpx
  1    [NS=12..24] [JO=rigid,mixed] [VT=0]    (t:103, u:103)
  2    [NS=30] [BL=30] [JO=rigid,mixed] [BA=2..3] [VT=0]    (t:11, u:11)
  3    [NS=6] [BL=30] [JO=hinged] [BA=1] [VT=1]    (t:8, u:8)


Recommended-outhypo
  #    cpx
  1    [NS=12..30] [JO=rigid,hinged] [VT=1..3]    (t:106, u:106)
  2    [NS=6] [JO=hinged] [BA=2..3] [VT=1..3]    (t:20, u:20)


Infeasible-outhypo
  #    cpx
  1    [NS=30] [BL=30] [JO=rigid,mixed] [BA=1] [VT=0]    (t:8, u:8)



Learning system time:       0.283 seconds
Learning user time:         0.00  seconds
```

Figure 3. AQ15 coutput from the Wind Bracing problem using maximally specific rules

2. Knowledge Representation

AQ15c uses the VL_1 (Variable-valued Logic system 1) and APC (Annotated Predicate Calculus) (Michalski, 1975, 1983) representational formalisms.

Training examples are given to AQ15c in the form of *events*, which assign values to the domain's variables. Each *decision class* (or class, for short) in the training set may be assigned a set of events, which form the set of *positive examples* of the class. During the learning of rules, the events from all other classes are considered *negative examples*. When rules for another class are being generated, the positive and negative example labels are changed accordingly. For each class the algorithm chooses the best decision rule set (according to user-defined criteria) is produced that is complete and consistent with respect to the input events. A *complete* rule set is one that covers (describes) all of the positive examples. A *consistent* rule does not cover any negative examples. The user may provide initial decision rules to the program. These rules are treated as initial hypotheses. Intermediate results during the search for a cover are called *candidate hypotheses* or *partial covers.*

Each decision rule is described by one or more conditions (also known as selectors), all of which must be met for the rule to apply. A *condition* is a relational statement and is defined as:

[TERM RELATION REFERENCE]

where:
TERM is an attribute
RELATION is one of the following symbols: <, <=, =, <>, >=, >
REFERENCE is a value, a range of values, or an internal disjunction of values.

Conditions state that the attribute in TERM takes one of the values defined in REFERENCE. Examples of conditions (selectors) are shown below:

[color = red, white, blue]
[width = 5]
[temperature = 20...25, 50..60]

A *rule* (also called a complex) is a conjunction of conditions. The following are examples of rules in AQ15c notation:

[color = red, white, blue] [stripes = 13] [stars = 1..50]
[width = 12] [color = red,blue]

A *cover* (or a hypothesis) is a disjunction of rules that together describe all of the positive examples and none of the negative ones. The following is an example of a two rule cover:

[color = red, white, blue] [stripes = 13] [stars = 50] v
[color = red, white, blue] [stripes = 3] [stars = 1]

A cover is satisfied if any of its rules are satisfied, while a rule is satisfied if all of its conditions are satisfied. A condition is satisfied if the term takes one of the values in the reference. The cover shown in the above example can be interpreted as follows: An object is a flag if:

1) Its color ir red, white, or blue, and it has 13 stripes and 50 stars on it, or
2) Its color is red, white, or blue, and it has 3 stripes and 1 star on it.

3. Learning Algorithms

Figure 4 is a flowchart of the AQ15c program. Figure 5 is a flowchart of the STAR generation component of the algorithm. An illustration of the AQ algorithm using the diagrammatic visualization system DIAV is given in Wnek (1995).

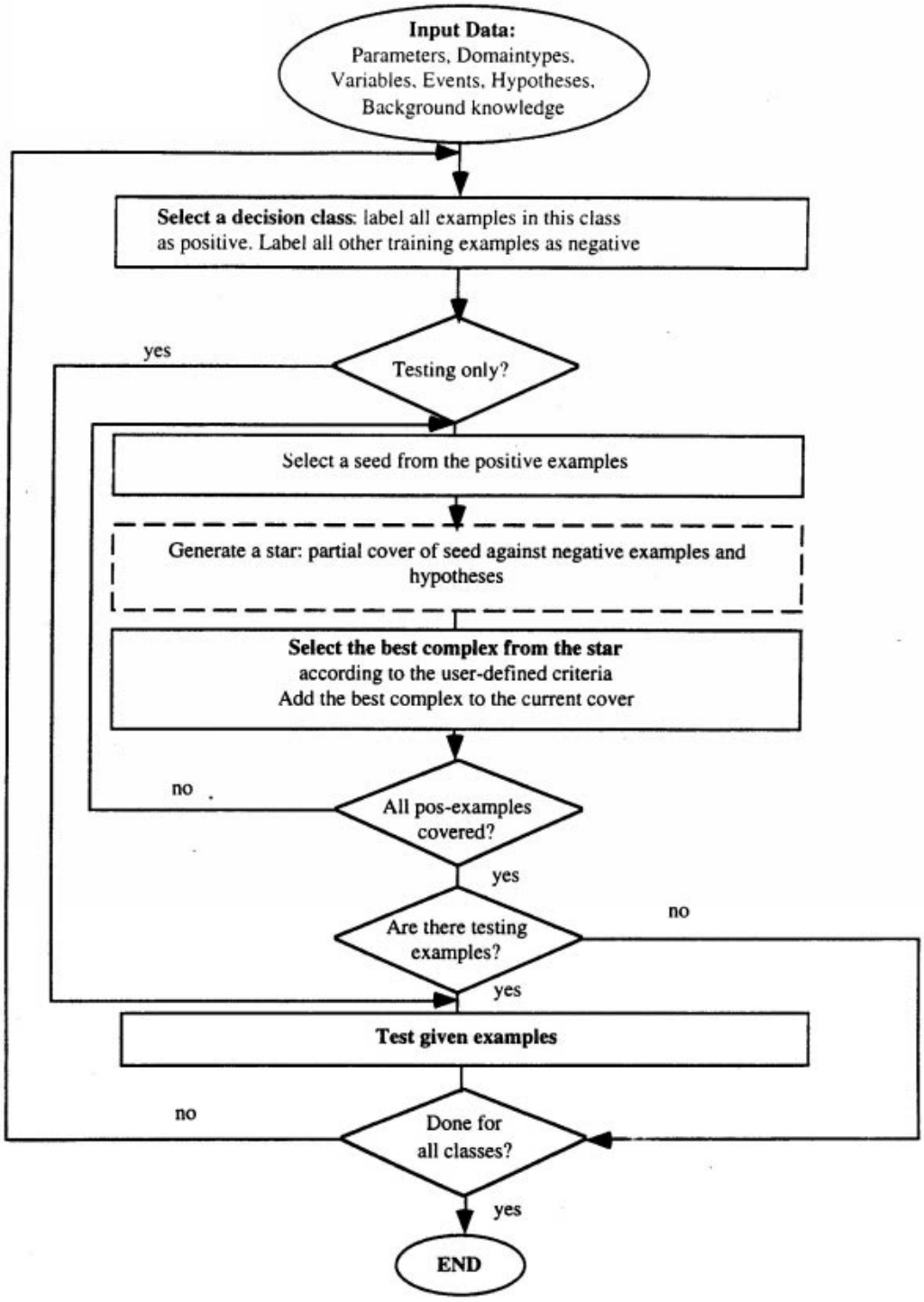


Figure 4. AQ15 algorithm

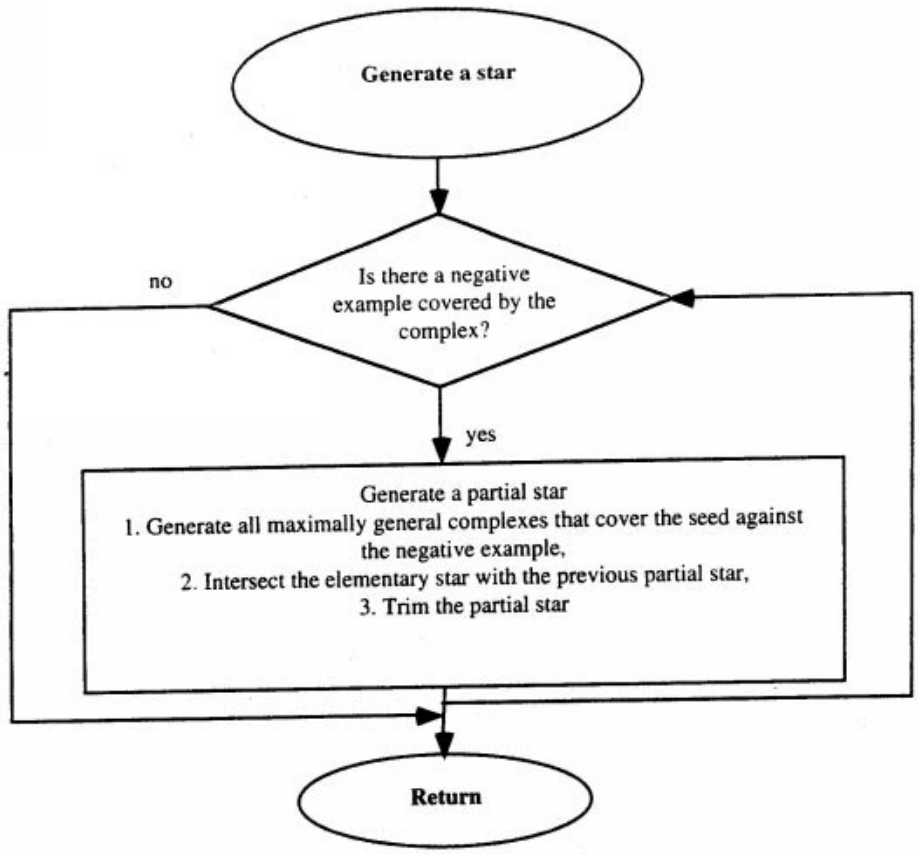


Figure 5. Star Generation

4. User's Guide

Input to AQ15c and all tools in the AQ family is in the form of a set of relational tables (Hong, Mozetic, Michalski, 1986). Relational tables have three parts – a table-name, a header, and a list of tuples. Legal table-names, headers and tuples are described below, and examples are given in Section 4.1.1. All tables must be separated by at least one blank line. Values within a header may be separated by spaces or tabs. In many cases some columns are optional (e.g., the cost column in the variables table). If an optional column is not used, a default value will be assigned. In some cases table columns may be ordered differently than is presented here. For example in the domaintypes table, the size column may come before (to the left of) or after (to the right of) the cost column.

4.1 Getting Started: A Simple Learning Problem

This section describes a sample use of AQ15c using the m1 files provided in the AQ15c package. m1 (or Monk #1) is the first in the set of three classification problems known collectively as the Monk's problems (Thrun et al., 1991). The goal of this problem is to correctly predict whether the given robot (as described in a single example) is in class1 or class2. Each robot is described by six

features (x1..x6). This example does not show how all of the AQ15c tables are used, but it serves as an introductory example featuring some of the most important table types.

If the AQ15c package has been succesfully installed (see the installation notes if this is not the case) the example can be run as follows:

(*Solaris, DOS*)	aq15c.run < m1.aqin	(with aq15c.run in the current directory or path)
(*Mac*)	double click on AQ15c.run	(m1 must be in the same folder as AQ15c.run)

The following sections describe the input file (m1.aqin) and output generated by AQ15c in more detail.

4.1.1 Input to the Learning Module

Input to AQ15c in this example includes a parameters table, variables table, and two events tables. Not all of the training events from the m1 file are shown here for space reasons. The output of AQ15c is shown in Section 4.1.2. The first table defines the parameters that will guide AQ15c's performance (Section 4.2.2). The second table defines the six variable domains (Section 4.2.5). The remaining tables define the event sets for AQ15c (Section 4.2.9).

```
parameters
run maxstar  trim  echo  wts
1    10      mini  pve   cpx

variables
    #    type    size  cost    name
    1    lin      4    1.00    x1.x1
    2    lin      4    1.00    x2.x2
    3    lin      3    1.00    x3.x3
    4    lin      4    1.00    x4.x4
    5    lin      5    1.00    x5.x5
    6    lin      3    1.00    x6.x6

class1-events
x1 x2 x3 x4 x5 x6
 3  2  1  1  4  2
 2  1  2  1  3  1
 1  2  2  3  2  2
 2  3  1  3  4  2
 1  2  1  1  4  2
 2  1  1  2  3  1
 1  3  1  3  2  2
 3  2  1  2  4  2
 2  1  2  1  4  2
 2  1  2  3  4  1
 1  2  2  3  3  2
 3  1  1  2  2  2

class2-events
x1 x2 x3 x4 x5 x6
 3  3  1  3  4  2
 2  2  1  1  2  2
 2  3  2  2  1  1
 2  2  1  3  3  2
 3  3  1  3  2  1
 2  2  1  3  4  2
```

4.1.2 Ouput from the Learning Module

This section shows what rules are generated after running AQ15c over the entire m1.aqin input file. Which tables are echoed depends on the value of the echo parameter in parameters table. The echo parameter in the parameters table in the input file tells AQ15c to echo the parameters, variables and events tables in addition to the learned rules (outhypo tables). The input tables are already given in section 4.1.1 so only the new outhypo table are shown here. The complete output is available in m1.aqout.

```
class1-outhypo
  #    cpx
  1    [x1=1] [x2=2..3] [x5=2..4]   (t:22, u:22)
  2    [x1=2..3] [x2=1] [x5=2..4]   (t:11, u:11)
  3    [x1=3] [x2=2] [x5=2..4]   (t:5, u:5)
  4    [x1=2] [x2=3] [x5=2..4]   (t:5, u:5)


class2-outhypo
  #    cpx
  1    [x5=1]   (t:16, u:11)
  2    [x1=3] [x2=3]   (t:15, u:11)
  3    [x1=2] [x2=2]   (t:10, u:10)
  4    [x1=1] [x2=1]   (t:7, u:6)
```

Outhypo tables containing decision rules (covers) are generated for each class. The class1-outhypo table gives rules describing the class1 examples (robots). Similarly the class2-outhypo table provides rules for class2. The weights associated with rules describe the coverage of individual rules. For instance, the weights associated with the first rule for class2 indicates that 16 examples are satisfied by that rule, and 11 of these examples are satisfied by this rule only. More details on these weights are provided in the description of the 'wts' parameter in Section 4.2.2.

4.1.3 An example using the Testing Module

This section provides sample input and output files for use with the testing module of AQ15c. This module tests rules against labeled testing examples. Input to the testing module consists of parameters, domains, rules or training events and testing events. The parameters are given to the system in a parameters table. The available parameters and their meanings are given in Section 4.2.2). The domains of variables are given in the standard variables table (Section 4.2.5). If rules have already been generated in a previous run or are from some other source, then AQ15c is run in testonly mode (as indicated in the parameters table). If rules have not yet been generated, training events in an events table must then be provided. In this case rules will be generated and immediately evaluated against the testing data.

The amount of information supplied by the testing module, and the method of evaluation between testing examples and rules is controlled by the test parameter (Section 4.2.2). This section provides examples of each of the available testing options.

4.1.3.1 Input to AQ15c with testing events

This section provides an example of an AQ15c input which contains testing examples. These files are provided under the name m1test.aqin and m1test.aqout. The evaluation methods designated by test parameter codes "m", "q" and "w" are all invoked, and the "c" parameter is present telling the

program to echo a full confusion matrix. When multiple test evaluation methods are selected, they are performed in the order that they appear in the parameters table. This example introduces two types of input tables not shown in Section 4.1.1: *names* tables, which define names for the different values of a variable (Section 4.2.6), and *tevents* tables, which provide the program with event sets by which rules are to be tested (Section 5).

```
 parameters
  run    mode    ambig    trim    wts    maxstar  echo      criteria  verbose  test
   1     ic      pos      spec    cpx      10     peq       default     1      mqwc

  variables
    #    type      size    cost    name
    1    nom        3      1.00    hs
    2    nom        3      1.00    bs
    3    nom        2      1.00    sm
    4    nom        3      1.00    ho
    5    nom        4      1.00    jc
    6    nom        2      1.00    ti

 ti-names
value name
0     no
1     yes

 sm-names
value name
0     no
1     yes

hs-names
value name
0     round
1     square
2     octagonal

 bs-names
value name
0     round
1     square
2     octagonal

 ho-names
value name
0     sword
1     balloon
2     flag

 jc-names
value name
0     red
1     yellow
2     green

Pos-events
    #    hs          bs          sm    ho        jc       ti
    1    round       round       yes   flag      yellow   yes
    2    round       round       yes   sword     red      no
```

```
  3   round      square     yes  sword    red      yes
  4   round      octagonal  yes  balloon  red      yes
  5   square     square     yes  balloon  red      yes
  6   square     square     no   balloon  green    yes

Neg-events
  #   hs         bs         sm   ho       jc       ti
  1   round      octagonal  yes  sword    yellow   no
  2   square     octagonal  yes  sword    yellow   no
  3   octagonal  square     no   sword    green    no
  4   octagonal  round      yes  sword    blue     yes
  5   octagonal  octagonal  no   balloon  green    no
  6   octagonal  round      no   balloon  blue     no
  7   octagonal  square     yes  flag     red      no
  8   octagonal  round      no   flag     green    no
  9   round      octagonal  no   flag     blue     yes
 10   round      octagonal  no   flag     green    yes
 11   square     round      yes  flag     yellow   yes

Pos-tevents
  #   hs         bs         sm   ho       jc       ti
  1   round      octagonal  yes  sword    blue     yes
  2   round      octagonal  yes  sword    green    yes
  3   round      round      yes  sword    red      no
  4   round      square     yes  sword    red      yes
  5   square     square     no   balloon  green    yes

Neg-tevents
  #   hs         bs         sm   ho       jc       ti
  1   round      octagonal  yes  sword    yellow   no
  2   square     octagonal  yes  sword    yellow   no
  3   round      round      yes  sword    red      no
  4   octagonal  square     no   sword    green    no
  5   octagonal  round      yes  sword    blue     yes
```

4.1.3.2 Output from AQ15c with testing results

The output of AQ15c when testing events are included consists of the rule sets and the requested testing results. The testing results are displayed in a confusion matrix that provides for every testing event the degree of confidence that that event belongs to each class. If the event matches a class other than the class to which it belongs, the name of that best matched class will be printed under the 'Class' heading. If the event most matches the correct class, no entry is given in the Class column. If a testing example matches multiple classes including the correct class, that example will be counted as correctly classified. Below are listed the same set of testing examples tested using all three evaluation methods. This list includes rform (test parameter setting "m"), hgform (test parameter setting "q") and igform (test parameter setting "w") (Section 5). The c option was also given to the test parameter so that full confusion matrices would be provided for each testing summary. If there are many testing events, the user may prefer not to see the entire confusion matrix, but rather a summary of the tests. The file containing this output is found in m1test.aqout

```
Pos-outhypo
  #   cpx
```

```
  1  [hs=round] [bs=round,square] [sm=yes] [ho=sword,flag] [jc=red,yellow]  (t:3, u:3)
  2  [hs=round,square] [bs=square,octagonal] [ho=balloon] [jc=red,green] [ti=yes]  (t:3, u:3)

Neg-outhypo
  #  cpx
  1  [hs=round,octagonal] [jc=green,blue]   (t:7, u:7)
  2  [hs=square,octagonal] [sm=yes] [ho=sword,flag] [jc=red,yellow]   (t:3, u:3)
  3  [hs=round] [bs=octagonal] [sm=yes] [ho=sword] [jc=yellow] [ti=no]   (t:1, u:1)

**********************************************************************
                        Testing Summary (Rform)
**********************************************************************
 # Event   Class          member   Pos      Neg
----------------------------------------------------------------------
  Pos:
    1      Neg                     0.00     0.64
    2      Neg                     0.00     0.64
    3                              0.50     0.00
    4                              0.50     0.00
    5                              0.50     0.00
  Neg:
    1                              0.00     0.09
    2                              0.00     0.27
    3      Pos                     0.50     0.00
    4                              0.00     0.64
    5                              0.00     0.64
----------------------------------------------------------------------
 # Events Correct:    7  # Events Incorrect:   3   Accuracy:  70%
......................................................................
 # Total Selectors:  22  # Total Complexes:    5
**********************************************************************

**********************************************************************
                        Testing Summary (Hgform)
**********************************************************************
 # Event   Class          member   Pos      Neg
----------------------------------------------------------------------
  Pos:
    1      Neg                     0.00     1.00
    2      Neg                     0.00     1.00
    3                              1.00     0.00
    4                              1.00     0.00
    5                              1.00     0.00
  Neg:
    1                              0.00     1.00
    2                              0.00     1.00
    3      Pos                     1.00     0.00
    4                              0.00     1.00
    5                              0.00     1.00
----------------------------------------------------------------------
 # Events Correct:    7  # Events Incorrect:   3   Accuracy:  70%
......................................................................
 # Total Selectors:  22  # Total Complexes:    5
**********************************************************************
```

```
*****************************************************************
                        Testing Summary (Igform)
*****************************************************************
 # Event    Class          member    Pos       Neg
-----------------------------------------------------------------
  Pos:
     1      Neg                      0.00      0.41
     2      Neg                      0.00      0.41
     3                               0.18      0.00
     4                               0.18      0.00
     5                               0.18      0.00
  Neg:
     1                               0.00      0.06
     2                               0.00      0.18
     3      Pos                      0.18      0.00
     4                               0.00      0.41
     5                               0.00      0.41
-----------------------------------------------------------------
 # Events Correct:    7  # Events Incorrect:    3   Accuracy:  70%
.................................................................
 # Total Selectors:  22  # Total Complexes:     5
*****************************************************************

This testing used:
  System time:       0.017 seconds
  User time:         0.00  seconds
```

In each of the test runs shown above, 7 of the 10 test events were correctly classified. Their degrees of match varied according to the testing algorithm.

4.2 AQ15C input tables

The following sections describe in detail the tables in AQ15c, their purpose, and syntax. Most tables are optional and many of the parts within each table are also optional.

4.2.1 Comments

Comments can be placed anywhere in the aq15c input file. The syntax of comments for AQ15c is the same as the syntax for comments in C: /* comment */. Comments may span multiple lines. Comments will not be echoed back at output.

Example

```
/* Filename: m1.aqin
   Date: 10/23/95
   Source: Monk's problem set */
```

4.2.2 The parameters table

The optional parameters table contains values which control the execution of AQ15c. All of the parameters have default values. The default values are provided in parentheses following the name of the parameter. Each row of the parameters table corresponds to one run of the program. In this way the user may specify in a single input file many runs using different parameter settings on the same data.

In order to allow parameters tables to be automatically generated after the input of other tables (for example to heuristically set the maxstar parameter based on the number of variables), an input file may include multiple parameters tables in various parts of the input file. However, there are a number of constraints and complexities if the user intends to use this feature directly.

1) If the first parameter table has no 'run' field, i.e., no run numbers are specified, then no subsequent parameters table may include this parameter. Similarly, if the first parameter table includes a 'run' parameter, then all following tables must include it explicitly as well.

2) Run numbers are used as labels for parameter settings. Run values must begin with 1 and each *new* run number must be one more than the highest one previously used. A previously used run number may appear on another parameter line, meaning that both parameter lines with this run number refer to the same run. If for that reason there are multiple values for any parameter in a given numbered run, the values appearing later will overwrite the earlier ones; the last value given supersedes the prior ones; this value will take effect. More details on the run parameter are given below.

A parameters table consists of a name line (that simply reads "parameters"), a header line defining the table's columns, and one or more lines defining parameter settings. The columns that may appear in a parameters table are as follows:

ambig (neg)
An optional parameter which controls the way ambiguous examples (i.e. overlapping examples from both the positive and negative class) are handled. Examples overlap when they have at least one common value for each variable. Legal values are:

neg	Ambiguous examples are always taken as negative examples for the current class, and are therefore not covered by any classification rule set.
pos	Ambiguous examples are always taken as positive examples for the current class, and are therefore covered by more than one classification rule set.
empty	Ambiguous examples are ignored, i.e., treated as though they were not part of the input event set. They may or may not be covered by some classification rule(s).

criteria (default)
Entry is the name of the criteria table (Section 4.2.3) to be applied to a given run. The name must be of alpha type, and a criteria table with that name (unless it is "default") must appear in the input file.

echo (pvne)
Specifies which tables are to be printed as part of the output. Values in this column consist of a string of characters. Each character represents a single table type. The order of characters in the string controls the order of the tables in the output. No blanks or tabs are allowed in this string (such white space separates words in the input and will confuse the parser). Legal values for the echo parameter and the tables they represent are shown below:

0 ----- no echo
b ----- childrens table
c ----- criteria table
d ----- domaintypes table
e ----- events tables
i ------ inhypo table
n ----- names tables
p ----- parameters table
q ----- tevents tables

s ----- structure table
t ------ title table
v ----- variables table

maxstar (10)
Optional parameter that specifies the maximum number of alternative solutions retained during each stage of rule generation. The program uses *a beam search*, in which at any intermediate stage, the best candidate hypotheses are retained up to a certain number. A higher number specifies a wider beam search, which also requires more computer resources and processing time. Empirical evidence indicates that in general the size of maxstar should be approximately the same as the number of variables used. The rules produced tend to indicate a good compromise between computational resources and rule quality. Maxstar values may range from 1 to 50.

mode (ic)
An optional parameter which controls the way in which AQ15c is to form rules. Legal values for this column are:

ic — "Intersecting covers" mode allows rules from different classes to intersect over areas of the learning space in which there are no examples.

dc — "Disjoint covers" mode produces covers that do not intersect at all with one another.

vl — "Variable-valued logic" mode produces rules are order-dependent. That is, the rules for class *n* will assume that the rules for classes 1 to *n*-1 were not satisfied. Hence there are no rules given for the last class; if none of the other rules were satisfied, an example is by default put into this class.

To illustrate the difference between these modes, consider two classes, one consisting of red circles, and the other one consisting of blue squares. In ic mode, the rules "Class 1 if red, Class 2 if square" might be produced. In dc mode, such a rule set would not be allowed, since red squares would be described by both rules. In vl mode, only the rule for Class 1 would be necessary; anything else would be assumed to belong to Class 2.

run (1..n)
An optional parameter which controls for which execution of the program the parameters line applies. Run numbers must be positive integers beginning with "1", with no succeeding integer appearing as a run number before its predecessor has already appeared. If used, this parameter must be in the first column of the parameters table. The default value is simply the number of the line, so the third parameters line would be assigned a default run number of 3.

test (m)
Optional parameter that controls the method of evaluation and the form of the output produced from testing the provided tevents (testing events) against the learned or provided rules. If no tevents are given, this parameter has no effect. If multiple methods are specified for this parameter, then multiple testing runs are performed in the order in which they were requested, and the results of each are summarized. This allows a direct comparison between testing methods. The three methods vary in their evaluation of the degree of match between the testing examples and the rules. The legal values for this parameter and an explanation of each method of evaluation are as follows:

m — INLEN, or rform mode used for testing. This is the mode used in the INLEN system (Michalski et al., 1992) .

q — hgform evaluation method first described in (Michalski and Chilausky, 1980) .

w — igform evaluation method described in (Michalski, 1986) .

c — This value controls whether a confusion matrix is reported in addition to the summary of the test run. If this value is present, the full matrix is echoed;

otherwise only the summary is provided. The summary provides the number of events correct, the number of events incorrect, the percentage of correct events, and the total numbers of conditions and rules in the rule set. A full confusion matrix provides for every testing event and every class the degree of confidence that that event belongs to the given class. In addition it gives the name of the class to which the event was best matched if that class was not the target class and tells whether or not the testing event was matched exactly (as opposed to being matched by "best fit") by any rule.

Example testing summaries using the m, q and w options are provided in section 4.1.3.2 along with examples of full confusion matrices. Details of the testing methods are given in section 5.2.

trim (mini)
An optional parameter that specifies the generality of the output rules (i.e., the number of possible events they satisfy). The legal values are:

gen	Rules are as general as possible, involving the minimum number of conditions, each with a maximum number of values.
mini	Rules are as simple as possible, involving the minimum number of conditions, each with a minimum of values.
spec	Rules are as specific as possible, involving the maximum number of conditions, each with a minimum of values.

Each condition restricts the set of events which will satisfy a rule, but each value in a condition relaxes these restrictions. Hence, rules with few conditions, all of which permit many values will be more general than ones with many conditions, each of which specifies only a few values.

verbose (1)
Optional parameter that specifies whether the time taken by the learning and/or testing process is to be added to the output from AQ15c. The verbose parameter is not solely responsible for controlling the contents of the output. The *echo* parameter controls which tables are echoed, the *wts* parameter (see below) controls the level of detail in the rule descriptions, and the *test* parameter controls the testing summary detail. The default value for the verbose parameter is 1. Legal values are:

0	No learning or testing times are echoed.
1	Learning time echoed. Testing time echoed if testing is performed.

wts (cpx)
Optional parameter that specifies whether AQ is to display weights with the rules it produces. The weights can give the user a gauge of the importance of individual rules or conditions based on how much they discriminate between the classes and how many of the input examples they actually applied to. Legal values are:

no	Include no weights in the output.
cpx	Two weights are associated with each complex (rule). The first weight is the total number of positive events that the rule covers (the *total* weight, denoted by "t"). The second weight is the number of events covered by this rule and no other rule in the cover (the *unique* weight, denoted by "u"). Rules for a given class will be displayed in decreasing order of the t-weight.
evt	In addition to the two weights (total and unique) calculated for each complex, a list of example indices is printed. These indices list by number, or by key field if keys were included in the events-tables, the positive examples which are covered by each rule.

sel — Weights are calculated for each selector (condition) in the rule set. There are two weights associated with each selector. The first weight is the number of positive examples covered by the condition, and the second weight is the number of negative examples covered by the condition. When selector weights are shown, the conditions within a rule are displayed in decreasing order of the ratio of the first weight to the sum of both weights, i.e., the percentage of positive events covered. Otherwise the conditions in a rule are displayed in the order that their attributes (as given in the TERM portion) are given in the variables table.

all — All weights and example information is printed for each selector and complex. In other words, *all* is the union of evt and sel.

A sample parameters table is shown below. This table directs AQ to run twice. Values in the first row are the default and control the first run. Values in the second row control the second run of AQ on the same data. Note that the default criteria table is the ONLY one for which it is unnecessary to follow with a full table description. The mincost criteria table, however, must be defined later in the input file. See the next section for a description of criteria tables. Parameters not present in the parameter table (e.g. "run", "mode" and "maxstar") take their default values in both runs.

Example:

```
parameters
ambig    trim    wts    echo    criteria
neg      mini    cpx    pvne    default
pos      gen     all    pvne    mincost
```

4.2.3 The criteria tables

All criteria tables other than the "default" *must* be defined. This table type is used to define a lexicographic evaluation function (LEF). An LEF evaluates a set of candidate hypotheses, using a series of preference criteria in order, with the most important criterion being used first, and so on. Examples that fail to meet the first criterion are eliminated, while those that qualify are only then evaluated on the second criterion. Those that qualify under that criterion are then examined according to the third one, and so forth. The LEF is used by AQ15c to judge the quality of each complex formed during learning. The LEF consists of several criterion-tolerance pairs. The ordering of the criteria in the LEF determines the relative importance of each. The tolerance specifies the allowable deviation from the optimal found value within each criterion.

A criteria table name consists of two parts - the specific name, which must appear in the "criteria" column of the parameters tables (in the example above "mincost" was used) and the table name, -criteria. In the previous example "mincost" in the parameters table refers to the existence of a "mincost-criteria" table later in the input file. Any value in the criteria column of the parameters table except "default" must have a corresponding -criteria table and vice-versa.

 # (1..n)
This column numbers the entries in the criteria table. Values must be sequential integers. This column is not required.

criterion (maxnew, minsel)
This mandatory column specifies the criterion which is to be applied at this point in the LEF. There are eight defined criteria. From these eight the LEF that best describes the user's rule preference is built. At least one and at most all eight criteria can be used in a criteria table. Criteria may also be selected by number rather than name. These numbers are the indices of the criteria in this table.

maxnew (1) - maximize the number of newly covered positive events, i.e. events that are not covered by previous complexes.
maxtot (2) - maximize the total number of positive events covered.
newvsneg (3) - maximize the ratio between the total number of newly covered positive examples and all negative events covered. Computationally expensive.
totvsneg (4) - maximize the ratio between the total number of positive covered examples and all negative evente covered. Computationally expensive.
mincost (5) - minimize the total cost of the variables used (see Section 4.3.4).
minsel (6) - minimize the number of extended selectors (conditions).
maxsel (7) - maximize the number of extended selectors.
minref (8) - minimize the number of references (permitted values) in the extended selectors.

tolerance (0.00)
This mandatory column specifies the relative tolerance in the importance of this criterion. In a strict LEF (tolerance = 0) any complex not having the best (or equal) value for a criterion is immediately eliminated. This real-valued number specifies the degree of tolerance in the importance of the criterion given in the same line. As an example, suppose the best complex in a list had a value of 100 for the first defined criterion, and the tolerance for this criterion was 0.2. The absolute tolerance value is the product of the tolerance value (0.2) and the best value (100) and thereby allows a leniency range of 20. Any complex with a value between 80 and 100 will not be eliminated from the list of rules under consideration; instead they would make up the set of rules evaluated under the second criterion..

An example is given below. The first -criteria table given is the default. In many experiments, this criteria table will produce good results. This is the only -criteria table that need not be defined. The second example is a user-defined table called "mincost". Note that row numbers may be omitted.

Example:

```
default-criteria
# criterion tolerance
1  maxnew    0.00
2  minsel    0.00

mincost-criteria
criterion  tolerance
 mincost        0.20
 maxtot         0.00
```

4.2.4 The domaintypes table

The domaintypes table is used to define domains for the attributes by which the input and output events are defined. This table is optional, but it is convenient if several attributes have the same set of possible values. The table consists of four columns. The type, size, and cost columns all have the same meanings as defined in the variables table description. There is no limit to the number of domains, or to the number of values (as defined in the size column) in a domain.

name (x_n)
This mandatory column is the name of the domain being defined and must be of alpha type. If the name is not provided, the default name will be x_n where n is the index of the entry in the domaintypes table.

type (nom)
This optional column specifies the type of the domain being defined. Four domain types are legal. The legal types are described below:

nom	A "nominal" domain consists of discrete, unordered values (e.g. colors)
lin	A "linear" domain consists of discrete, ordered alpha or numeric values (e.g. sizes – small, med, large)
cyc	A "cyclic" domain consists of discrete values in a circular order (e.g. months).
str	A "structured" domain has values in the form of a hierarchical taxonomy (e.g. types of food). A variable with a structured domain requires that domain be described in a structure table (Section 4.2.7) as well.

size (2)
This optional integer value specifies the number of values in the domain being defined. There is no preset limit on domain size.

cost (1.00)
This optional real value specifies the relative expense of the domain being described. This value is used by the mincost criterion in the LEF (see description of the criteria table, Section 4.2.3). The expense of an attribute may be determined by the difficulty or expense of acquiring the value, or it may be set by a domain expert to encourage or discourage this attribute's appearance in generated rules. For instance, in a medical domain, it would be desirable to make a diagnosis via a blood test, rather than exploratory surgery. To only generate rules that involved the results of such surgery when absolutely necessary, the attribute describing the results of the surgery could be assigned a prohibitively high cost, while the blood chemistry attributes would have low costs.

The domaintypes table is normally used in conjunction with the variables table and the names table. An example of both the variables table and domaintypes is given at the end of the following section.

4.2.5 The variables table

The mandatory variables table specifies the names and domains (legal values) of the variables used to describe events. The variables table must include at least one, and at most all 5 of the following columns. There is no preset limit to the number of variables, or to the domain size of a variable.

 # (1..n)
This optional column numbers the entries in the variables table. Values must be sequential integers.

name (x*n*)
This column specifies the name of the attribute. Names must be of alpha type. If the name column is omitted, the default value is x*n* where *n* is the number of the row in which this variable was defined. The value in the name column takes the form name.domain-name, where name is the alpha string name of the specific variable while domain-name is a more general name of the domain (if defined in the domaintypes table), or simply a repetition of the variable name.

type (nom)
This column specifies the type of the variable domain. Four domain types are legal:

nom	A "nominal" domain consists of discrete, unordered values (e.g. colors)

lin	A "linear" domain consists of discrete, ordered alpha or numeric values (e.g. sizes – small, med, large)
cyc	A "cyclic" domain consists of discrete values in a circular order (e.g. months).
str	A "structured" domain has values in the form of a hierarchical taxonomy (e.g. types of food). A variable with a structured domain requires that domain be described in a structure table as well.

size (2)
This integer gives the number of legal values in the domain being defined. There is no limit on the domain size. The size of a structured variable is defined as the total number of nodes in the hierarchy (internal and leaf).

cost (1.0)
This real number specifies the relative 'expense' of the domain being defined on this line. The value is used when computing criterion mincost is used in the LEF. The default cost is 1.0. Further explanation is given in Section 4.2.4.

The following example shows domaintypes and variables tables for a computer selection task. Notice that five of the ten variables take on boolean domains.

Example:

```
domaintypes
name          type    size
boolean       nom     2
op_system     nom     2
floppies      lin     4
processor     nom     3
memory        str     8
```

```
variables
#    name                cost
1    pascal.boolean      10.0
2    fortran.boolean     10.0
3    cobol.boolean       10.0
4    op_system           10.0
5    floppies            100.0
6    disk.boolean        0.0
7    processor           1.0
8    memory              100.0
9    printer.boolean     0.0
```

The variables table may be used in conjunction with the domaintypes and names tables. In the example given above, the type and size columns were defined for all domaintypes so that these columns were not needed in the variables table.

4.2.6 The names tables

The names tables are used to specify the legal domain values for an attribute. These must include the attribute values that appear in the events tables. If no names table is present in the input file, then the values for that domain are assumed to be the integers from 0 to size-1 (size is defined in the variables or domains table). The specific name of a names table must be the same as that used in the domains or variables table. There are two required columns in each names table, value and name.

value (1..n)
This column must be an integer beginning with '0' and continuing sequentially up to size-1. This column is the integer equivalent of the name to be defined in the next column. This column is required. There is no preset limit to the number of values for an domain being defined.

name (1..n)
This column defines the input and output name of the value being defined. Alpha, integer or real types are allowed. Only two decimal places are stored for real types. This column is required.

Below are examples of the -names tables for the computer selection problem defined above. All variables that are of type "boolean" may take values "yes" and "no". The domain "make" has the values "IBM", "Compaq", "Zenith" and "Apple". Note that for the variable "floppies" the default values of 0,1,2 and 3 are acceptable, so there is no need for a "floppies" names table.

Example:

```
boolean-names
  value    name
   0        no
   1        yes

make-names
  value   name
  0       IBM
  1       Compaq
  2       Zenith
  3       Apple
```

4.2.7 The structure tables

The structure table is optional and is used to define a structured domain for any variable of the structured type (as specified in the domaintypes or variables table). A structured domain has the form of a hierarchical graph, in which the lowest level corresponds to the values of the variable at the lowest level of generality. Higher levels (as defined the structure table) specify parent nodes in the hierarchy of values and are used to simplify classification rules. For instance, the hierarchy for the structured variable shape may have curve and polygon as top-level nodes, circle and ellipse as leaves under curve, and triangle and square among the leaves under polygon.

The specific name of a structure table must be the name of the domain, as specified in the name column of the domaintypes table, or if the domaintypes table is not specified, in the variable name from the variables table. A structure table consists of three columns:

name
This optional alpha or integer type entry specifies the name of the element in the hierarchy corresponding to the value given in the "value" column of the table. If this column is included, the names used in this column may appear in classification rules instead of the values named in the names table or the events-tables.

value
This mandatory integer entry specifies a parent node in the hierarchy. This node will be defined as the parent of the nodes specified in the subvalues column. If this value is a subvalue of some other values, the row in which this value is declared as a parent must appear before any rows in which it is listed as a subvalue (e.g. entries must be given in a bottom-up order). This integer value must always be greater than any of the subvalues in the following column.

subvalues
This mandatory entry specifies a set of children values for the parent node as defined in the value column. This entry consists of a string of integers separated by commas or by ".." to indicate

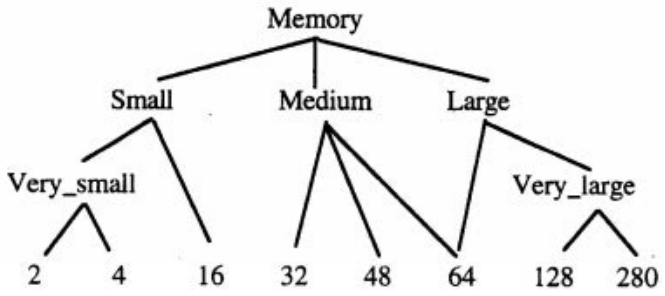
ranges. These numbers correspond to values as defined in the names table of the variable or previous rows of the structure table.

The hierarchical graph below shows an example of a structured domain for the variable "memory". Note that the same node (64 in this example) may be shared by multiple parents ("medium" and "large"). Below are the input tables to define such a structure. Note that for the variable "memory" a names table must first be defined, because a domain of all values between 2 and 280 would be inefficient and might cause inaccurate rules. Furthermore, the node "large" must be defined after the node "very_large", as it is higher in the tree.

Example:

```
memory-names
  value   name
  0       2
  1       4
  2       16
  3       32
  4       48
  5       64
  6       128
  7       280
  8       very_small
  9       small
 10       medium
 11       very_large
 12       large
```

```
memory-structure
 name          value      subvalues
 very_small    8          0,1
 small         9          8,2
 medium        10         3..5
 very_large    11         6,7
 large         12         5,11
```



4.2.8 The inhypo tables

The inhypo tables are optional and are used to input rules for incremental learning (i.e., learning by modifying prior hypotheses, rather than from examples alone). The specific name of this table must match the name of one of the decision classes. The rules input in the inhypo table have two possible roles. In the first, when there is at least one events table specified, the input rules are used as initial covers for incremental learning. If no events tables are present, the inhypo rules are treated as events for rule optimization. If inhypo rules for different classes intersect in the event space, then their intersection is treated as determined by the "ambig" parameter in the parameters table (Section 4.2.2). There are two columns in the inhypo table.

 # (1..n)

This mandatory column associates a number with each complex in the rule set for the class being described by this inhypo table. It is a sequentially increasing integer (1..#complexes). Only in inhypo tables may an entry span more than one line. There must always be a # entry for each complex in the table.

cpx ([])
This mandatory column specifies the VL_1 rule. A complex (rule) is presented as a conjunction of selectors (conditions), each enclosed in square brackets. Selectors and complexes are defined in Section 2. Below are examples of an inhypo table:

Example:

```
Under1000-inhypo
  #  cpx
  1  [floppies=0]

From1000_4000-inhypo
  #    cpx
  1    [Floppies = 1,2][memory > 16]
  2    [Floppies > 3][memory >= 4]
```

4.2.9 The event and tevent tables

Events and tevents tables have identical structure except the examples contained in events tables are used for learning and those in tevents are used in testing. Both types of tables contain a specific name corresponding to the name of the decision class. This name must be of type alpha.

The column headers for this table consist of the attribute names (as defined in the variables table). The values in the row of the table must be legal values for the appropriate attribute. In the case that many attributes are used, events tables may be split. Each split table must contain the specific name and 'events' and a different set of attributes. Attributes can not overlap between split events tables. Events tables consists of three column types: 1) row number, 2) Key and 3) attribute name.

 # (1..n)
This optional column is an integer index of the example. Values must be sequential integers beginning with 1. This column is optional.

Key ()
This optional column permits the user to include an identifier for the example in alpha form. This feature is useful when the wts parameter is set to 'all' (Section 4.2.2). It appears like any other variable column, but in its role as strictly an identifier, its values are not used in learning.

variables (x1..xn)
This definition consists of an arbitrary number of columns, one for each attribute in the variables table. The entries in the rows of the table must contain legal values of the corresponding variables in the heading. Entries may be single values or they may be an 'unknown' symbol (*). Unknown values (*) are internally represented as taking all legal values for that attribute domain. Below is an example of a set of events tables:

Example:

```
Under1000-events
 Pascal   Fortran   Cobol   floppies   Disk   Processor   Memory   Printer
 no       no        no      0          no     M6502       2..16    no
```

```
 no        no          no        0          no      Z80          32        no

Over4000-events
 #  Pascal    Fortran    Cobol
 1  yes         yes      yes
 2  no          yes      yes

Over4000-events
  #    floppies  Disk    Processor  Memory    Printer
  1    1         yes      Z80        128       no
  2    2         no       I8085      64        yes
```

4.2.10 The children tables

The optional children tables define the hierarchical ordering of the values of the decision variable (i.e., the different classes) in cases in which the decision variable is structured. The specific name of the table must be the name of a class already defined, i.e. the name must have appeared as the specific title in an events table. The rule base may be structured to arbitrary depth. The children table consists of two columns:

node
This mandatory alpha column specifies the name of the child node being defined, i.e., the subclass of the class by which the table is titled.

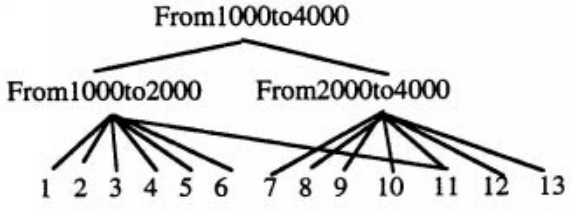
events
This mandatory column is a list of indices of events belonging to the parent node (the node on which the table's name is based) which are examples of this child node. The list of indices may use commas (1,2,3,4) or ranges (1..4). The events are numbered in the order they appear in the parent node's events table.

The tree below shows how a decision attribute may be structured. In this case classes "Under1000", "From1000to4000" and "Over4000" are siblings at the top of the structure. The class "From1000to4000" has thirteen events and two sub-classes — "From 1000to2000" and "From2000to4000".

The example below defines the two classes "From1000to2000" and "From2000to4000" which are subclasses of the class "From1000to4000". Assuming that there is already an events table for "From1000to4000" with 13 events, the following children tables would assign events 1 to 6 and 11 to class "From1000to2000" and events 7 to 13 to class "From2000to4000".

Example:

```
From1000to4000-children
  node                events
  From1000to2000       1..6,11
  From2000to4000       7..13
```



5. Testing Rules

5.1 *Introduction*

The Testing module is used to evaluate the performance of a rule base. This new module incorporates the functionality of the separate testing program known as ATEST (Reinke, 1984). By sharing the input format with AQ15c, it becomes a convenient operator in the AQ15c package, allowing the user immediate analysis of discovered rules.

5.2 *Testing Methods*

In this module the fundamental operation is the calculation of the degree of match between a rule and an example in the form of a vector of attribute values. The calculated value is called a degree of confidence. The calculation of confidence varies with the method of evaluation selected by the user in the test parameter in the parameters table. Full descriptions of these methods are given below based on the following test parameters:

m INLEN, or rform mode used for testing. This mode is named after INLEN in which it is used (Michalski et al, 1992). In this method of evaluation:

> For every testing example, evaluate each class ruleset (cover) against the testing example.
>
> > If a match exists between one or more covers and an example:
> >
> > For each cover, confidence is the *probabilistic sum* of confidences for each rule in the cover, where the probabilistic sum of occurrences A and B is P(A) + P(B) - (P(A) * P(B)). A rule's confidence calculation depends on the source of the evidence for that rule. If the rule was learned from training examples provided in the current learning session, then the rule's confidence is the percentage of examples of its class covered by that rule (i.e. a heavier rule or one that better describes a large share of the training examples will have a higher confidence when matching a new testing example than a match to a 'lighter' rule.)
> >
> > Else:
> >
> > The rule's confidence is the ratio of testing examples of that class covered by that rule to the total number of testing examples of that class covered by any rule for that class (i.e. in the absence of training examples, a 'stronger' rule is one that matches more of the testing examples).
>
> Else (if no match exists between a class ruleset and the testing example):
>
> > A class confidence is a probabilistic sum of confidences for each rule in the cover. A rule's confidence is calculated as the ratio of the number of conditions in the rule matching the example to the total number of conditions in the rule.

q hgform evaluation method first described in (Michalski and Chilausky, 1980). In this method of evaluation:

Evaluate each cover against all testing examples. If a match exists between a rule for a given class and an example:

For each class, the degree of match (confidence) is the number of training examples of the class covered by the rules that match the testing example. This value is normalized by dividing all confidence values by the maximum calculated confidence.

Else:

For each class, confidence is the probabilistic sum of confidences for each rule in the cover. A rule's confidence is calculated as the ratio of the number of conditions in the rule matching the example to the total number of conditions in the rule.

w igform evaluation method described by Michalski (1986). In this form of evaluation:

Evaluate each class ruleset against all testing examples. If a match exists between a cover and an example:

For each class, the degree of match (confidence) is the probabilistic sum of confidences for each rule's confidence in that class. A rule's confidence is calculated as the ratio of the number of events covered by the rule to the total number of all events (for all classes). This evaluation scheme assumes the number of training examples in a class are characteristic of the distribution of examples in the universe of possible examples.

Else:

For each class, confidence is the probabilistic sum of confidences for each rule in the cover of that class. A rule's confidence is calculated as the product of the confidences for each condition in the rule weighted by the percentage of all training examples covered by the rule. A condition's confidence is 1 if it covers the testing example otherwise it is a value between 0 and 1 proportional to the ratio of the number of possible values satisfying the condition to the full domain size.

Example testing summaries are provided in section 4.1.3.2, along with examples of full confusion matrices using the m, q and w options.

6. Concept Learning Using AQ15c: Examples

6.1 Learning and Visualizing Three Concepts in the ROBOTS Domain Using Various Parameter Settings

This section provides examples of concept learning using AQ15c. All of these examples are in the 'robots' files provided with AQ15c. This section describes the problem domain and the example sets that will be used in each of these runs. We present here a variation on the "robots world" domain in which there are five variables: Whether or not the robot is smiling, what it is holding, its size, the time of year (season) in which it was manufactured and location of its home planet. These represent a binary, a nominal, a linear, a cyclic, and a structured domain. The goal of this problem is to predict whether a given robot is likely to be friendly, unfriendly or neutral based on the behavior of other observed robots.

The first run has the parameter settings from which the variations in the following runs are based. As we begin to study the ambig parameter, note that two examples -- Friendly #2 and Neutral #2 -- are identical. Runs 2 and 3 show the effects of the ambig parameter, runs 4 and 5 show the mode parameter's effects, runs 6 and 7 focus on the trim parameter, run 8 on the maxstar parameter, and run 9 will show the effects of different preference criteria.

Figures 6-14 illustrate the Robots representation space and results of learning usingthe DIAV diagrammatic visualization system (Wnek, 1993). The system employs a General Logic Diagram (GLD) which is a planar model of a multidimensional space spanned over a set of multivalued discrete attributes (Michalski, 1973). Each cell in the GLD represents a unique combination of attribute values. Each attribute partitions the diagram into areas corresponding to the individual values of the attribute. Conjunctive rules correspond to certain regular arrangements of cells that can be easily recognized visually.

Diagrams visualizing the Robots representation space use the following abbreviations of attribute names and attribute values.

```
smiling            yes, no
holding            sword, flag, balloon
size               short, medium, tall
season             spr, sum, fal, win
home planet HP     Titan, Ganymede, Mars, Venus, Sirius, Rigel, Betelguese,
                   Halley_Comet
```

Diagrams visualizing the Robots representation space consist of 576 cells (points in the representation space). Training examples of Friendly, Unfriendly and Neutral Robots are marked using "1" "2" "3", respectively. Ambiguous examples are marked using "#". Shaded areas show images of learned concepts. Different shades are used for different classes and their intersections (see the legend under the diagram). Empty cells represent areas of the representation space not covered by any concept.

6.1.1 AQ15c - input file for Robots example

```
parameters
 run   mode   ambig   trim   wts   maxstar   echo   criteria   verbose
  1     ic     pos    mini   cpx     10      pcvnse default      0
  2     ic     neg    mini   cpx     10      p      default      0
  3     ic     empty  mini   cpx     10      p      default      0
  4     dc     pos    mini   cpx     10      p      default      0
  5     vl     pos    mini   cpx     10      p      default      0
  6     ic     pos    spec   cpx     10      p      default      0
  7     ic     pos    gen    cpx     10      p      default      0
  8     ic     pos    mini   cpx     1       p      default      0
  9     ic     pos    mini   cpx     10      pc     lowcost      1

variables
 #   type   levels   cost   name
 1   nom      2.     1.00    smiling
 2   nom      3      1.00    holding
 3   lin      3      1.00    size
 4   cyc      4      1.00    season
 5   str     12      10.0    home

smiling-names
```

```
 value  name
 0      yes
 1      no

holding-names
 value  name
 0      sword
 1      flag
 2      balloon

size-names
 value  name
 0      short
 1      medium
 2      tall

season-names
 value  name
 0      spr
 1      sum
 2      fal
 3      win

home-names
 value  name
 0      Titan
 1      Ganymede
 2      Mars
 3      Venus
 4      Sirius
 5      Rigel
 6      Betelguese
 7      Halley_Comet
 8      Moons
 9      Planets
 10     Orion
 11     Stars

home-structure
 name     value  subvalues
 Moons    8      0,1
 Planets  9      2,3
 Orion    10     5,6
 Stars    11     4,10

lowcost-criteria
  #  criterion  tolerance
  1  mincost    0.00
  2  maxnew     0.10

Friendly-events
  #  smiling holding  size   season home
  1  yes     flag     short  spr    Venus
  2  yes     sword    tall   win    Halley_Comet
  3  yes     sword    tall   win    Mars
  4  yes     balloon  short  win    Rigel
  5  yes     balloon  medium fal    Betelguese
  6  no      balloon  short  win    Ganymede
```

```
Unfriendly-events
  #   smiling holding  size    season home
  1   yes     sword    tall    sum    Sirius
  2   yes     sword    medium  spr    Sirius
  3   no      sword    short   spr    Moons
  4   yes     sword    medium  sum    Sirius
  5   no      balloon  medium  fal    Sirius
  6   no      balloon  tall    fal    Titan
  7   yes     flag     short   spr    Sirius
  8   no      flag     medium  spr    Sirius
  9   no      flag     short   sum    Sirius
 10   no      flag     short   fal    Titan
 11   yes     flag     tall    fal    Ganymede

Neutral-events
  #   smiling holding  size    season home
  1   no      flag     medium  fal    Venus
  2   yes     sword    tall    win    Halley_Comet
  3   no      balloon  medium  sum    Mars
  4   yes     sword    tall    win    Ganymede
  5   no      flag     tall    spr    Titan
  6   no      sword    short   win    Halley_Comet
```

6.1.2 AQ15c - annotated output file

```
***  Run #1.  We list here the problem domain and the example sets that will
***  be used in each of these runs.  We present here a variation on the
***  "robots world" domain in which there are five variables:  Whether the
***  robot is smiling, what it is holding, its size, and the time of year
***  and location of its home planet.  These represent a binary, a nominal,
***  a linear, a cyclic, and a structured domain.

***  This run has the parameter settings from which the variations in the
***  following runs are based.  Runs 2 and 3 show the effects of the ambig
***  parameter, runs 4 and 5 show the mode parameter's effects, runs 6 and
***  7 focus on the trim parameter, run 8 on the maxstar parameter, and
***  run 9 will show the effects of different preference criteria.

***  As we begin to study the ambig parameter, note that two examples --
***  Friendly #2 and Neutral #2 -- are identical.  In this run, they will
***  be treated as positive examples of their classes, and as such are
***  covered by the rules Friendly-1 and Neutral-2.


  parameters
  run   mode   ambig   trim   wts   maxstar  echo      criteria  verbose
   1    ic     pos     mini   cpx     10     pcrnse    default     0


  default-criteria
    #   criterion   tolerance
    1   maxnew        0.00
    2   minsel        0.00
```

```
variables
  #    type     size     cost    name
  1    nom         2     1.00    smiling.smiling
  2    nom         3     1.00    holding.holding
  3    lin         3     1.00    size.size
  4    cyc         4     1.00    season.season
  5    str        12    10.00    home.home


smiling-names
  value    name
    0      yes
    1      no


holding-names
  value    name
    0      sword
    1      flag
    2      balloon


size-names
  value    name
    0      short
    1      medium
    2      tall


season-names
  value    name
    0      spr
    1      sum
    2      fal
    3      win


home-names
  value    name
    0      Titan
    1      Ganymede
    2      Mars
    3      Venus
    4      Sirius
    5      Rigel
    6      Betelguese
    7      Halley_Comet
    8      Moons
    9      Planets
   10      Orion
   11      Stars


home-structure
name                value      subvalues
Moons               8          0,1
Planets             9          2,3
Orion               10         5,6
```

Stars 11 4,10

Friendly-events

#	smiling	holding	size	season	home
1	yes	flag	short	spr	Venus
2	yes	sword	tall	win	Halley_Comet
3	yes	sword	tall	win	Mars
4	yes	balloon	short	win	Rigel
5	yes	balloon	medium	fal	Betelguese
6	no	balloon	short	win	Ganymede

Unfriendly-events

#	smiling	holding	size	season	home
1	yes	sword	tall	sum	Sirius
2	yes	sword	medium	spr	Sirius
3	no	sword	short	spr	Moons
4	yes	sword	medium	sum	Sirius
5	no	balloon	medium	fal	Sirius
6	no	balloon	tall	fal	Titan
7	yes	flag	short	spr	Sirius
8	no	flag	medium	spr	Sirius
9	no	flag	short	sum	Sirius
10	no	flag	short	fal	Titan
11	yes	flag	tall	fal	Ganymede

Neutral-events

#	smiling	holding	size	season	home
1	no	flag	medium	fal	Venus
2	yes	sword	tall	win	Halley_Comet
3	no	balloon	medium	sum	Mars
4	yes	sword	tall	win	Ganymede
5	no	flag	tall	spr	Titan
6	no	sword	short	win	Halley_Comet

Friendly-outhypo

#	cpx	
1	[smiling=yes] [home=Planets,Orion,Halley_Comet]	(t:5, u:4)
2	[holding=balloon] [size=short]	(t:2, u:1)

Unfriendly-outhypo

#	cpx	
1	[season=spr,sum,fal] [home=Ganymede,Sirius]	(t:8, u:8)
2	[smiling=no] [size=short] [season=spr,fal]	(t:2, u:2)
3	[holding=balloon] [home=Titan]	(t:1, u:1)

Neutral-outhypo

#	cpx	
1	[smiling=no] [home=Planets,Halley_Comet]	(t:3, u:3)
2	[size=tall] [season=spr,win] [home=Moons,Halley_Comet]	(t:3, u:3)

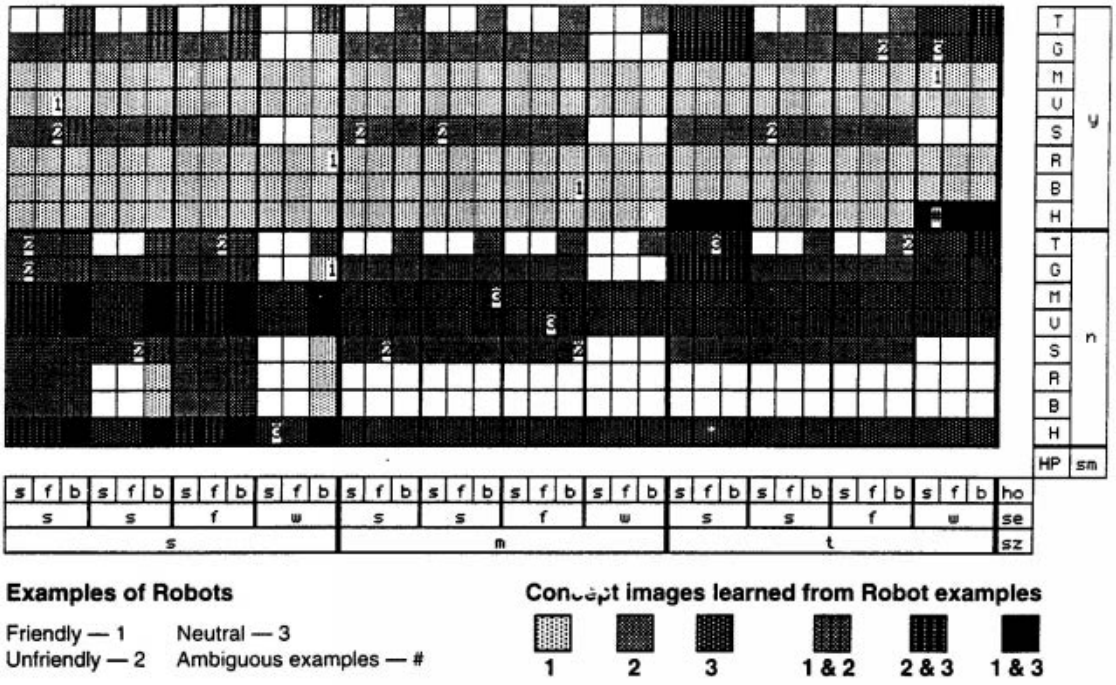


Figure 6. Run #1: The basis example using the initial parameter settings.

```
***  Run #2.  In this run, the identical events are treated as negative
***  examples of their classes, and as such are not covered by any of the
***  rules for the Friendly and Neutral classes.

  parameters
  run    mode    ambig    trim    wts    maxstar  echo     criteria  verbose
   2     ic      neg      mini    cpx     10      p         default     0


  Friendly-outhypo
    #    cpx
    1    [smiling=yes] [home=Planets,Orion]    (t:4, u:3)
    2    [holding=balloon] [size=short]    (t:2, u:1)


  Unfriendly-outhypo
    #    cpx
    1    [season=spr,sum,fal] [home=Ganymede,Sirius]    (t:8, u:8)
    2    [smiling=no] [size=short] [season=spr,fal]    (t:2, u:2)
    3    [holding=balloon] [home=Titan]    (t:1, u:1)


  Neutral-outhypo
    #    cpx
    1    [smiling=no] [home=Planets,Halley_Comet]    (t:3, u:3)
    2    [size=tall] [season=spr,win] [home=Moons]    (t:2, u:2)
```

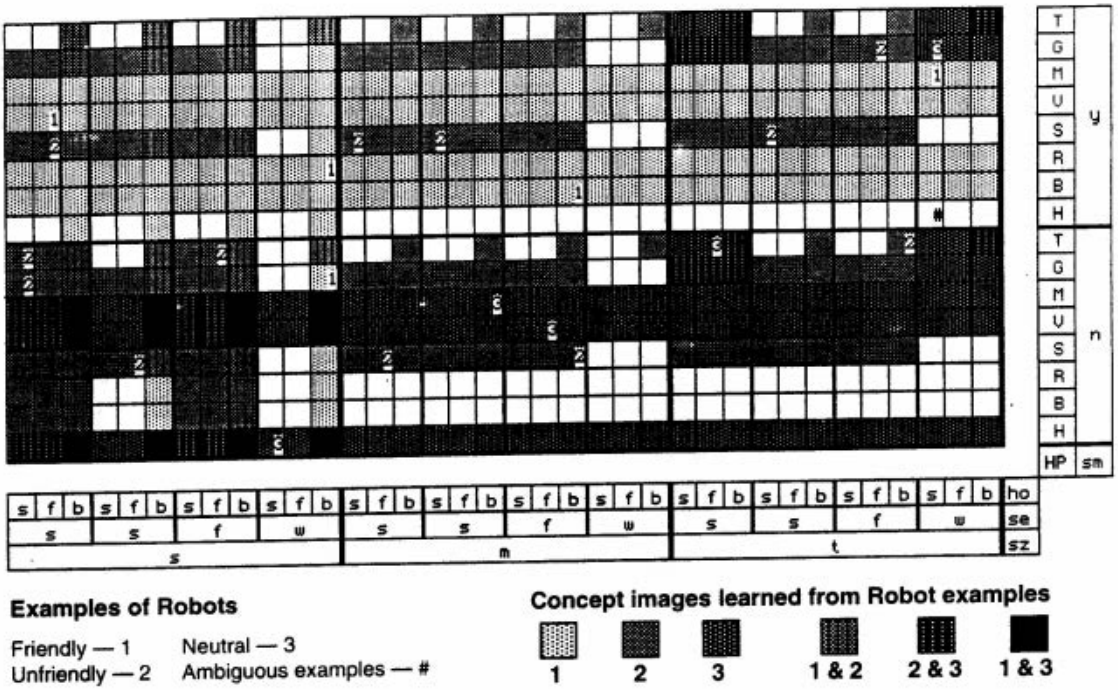
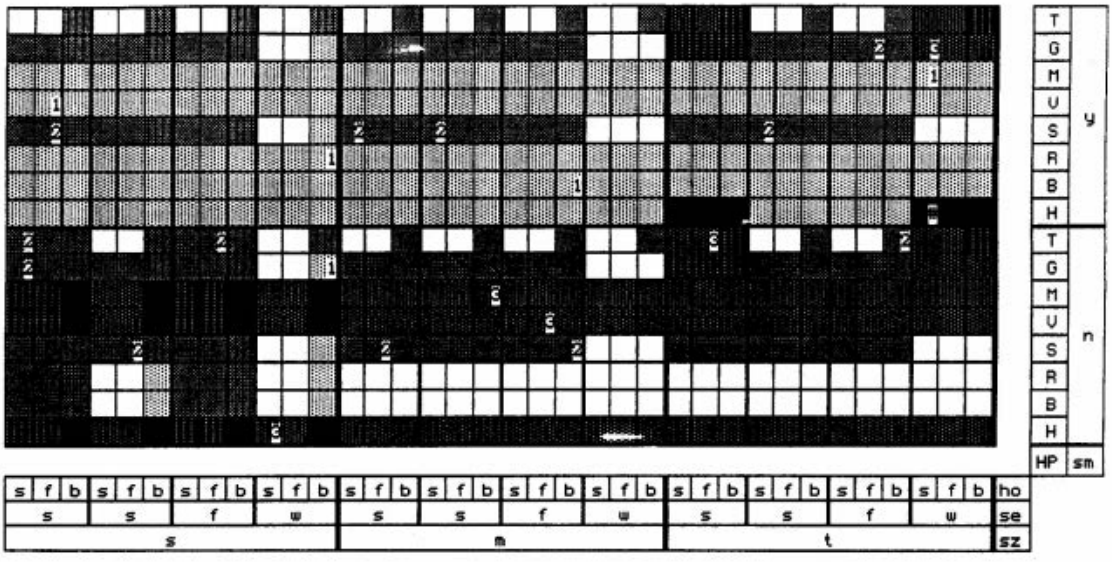


Figure 7. Run #2: Rules that exclude the ambiguous examples

```
***  Run #3.  In this run, the identical events are ignored by the program,
***  rather than being considered to belong to any class.  As it turns out,
***  the chosen rules for the Friendly and Neutral classes happen to cover
***  these events, so the final trimmed versions do include them.  This will
***  not always be the case.

parameters
run   mode   ambig   trim   wts   maxstar  echo     criteria  verbose
 3    ic     empty   mini   cpx    10      p         default     0


Friendly-outhypo
  #    cpx
  1    [smiling=yes] [home=Planets,Orion,Halley_Comet]    (t:5, u:4)
  2    [holding=balloon] [size=short]    (t:2, u:1)


Unfriendly-outhypo
  #    cpx
  1    [season=spr,sum,fal] [home=Ganymede,Sirius]    (t:8, u:8)
  2    [smiling=no] [size=short] [season=spr,fal]    (t:2, u:2)
  3    [holding=balloon] [home=Titan]    (t:1, u:1)


Neutral-outhypo
  #    cpx
```

```
1   [smiling=no] [home=Planets,Halley_Comet]    (t:3, u:3)
2   [size=tall] [season=spr,win] [home=Moons,Halley_Comet]    (t:3, u:3)
```



Examples of Robots

Friendly — 1 Neutral — 3
Unfriendly — 2 Ambiguous examples — #

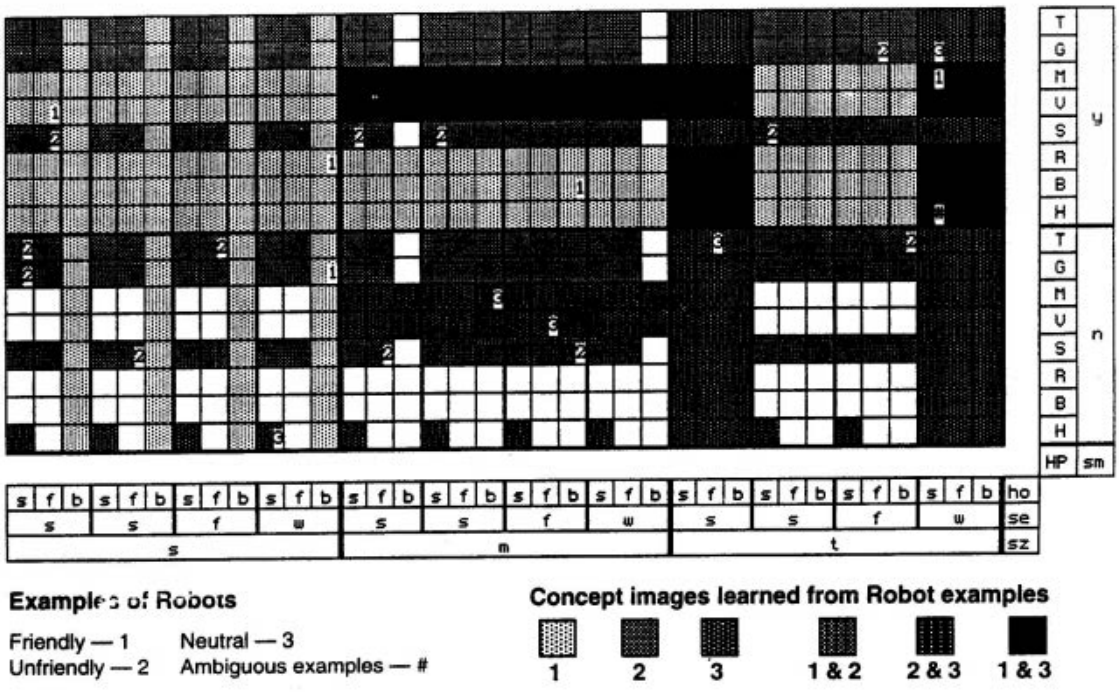
Concept images learned from Robot examples

1 2 3 1&2 2&3 1&3

Figure 8. Run #3: Rules generated by ignoring the ambiguous examples

```
***  Run #4.  We now do a run in dc mode. This means that the Unfriendly rules
***  may not apply to any part of the event space covered by the Friendly
***  rules, hence their changed, more complex form.  The Neutral rules may
***  not intersect with either of the other classes, although because of the
***  ambiguous event, this requirement is bent somewhat with respect to the
***  Friendly class.

  parameters
  run    mode    ambig    trim    wts    maxstar  echo      criteria  verbose
   4     dc      pos      mini    cpx      10     p          default     0


  Friendly-outhypo
    #    cpx
    1    [smiling=yes] [home=Planets,Orion,Halley_Comet]    (t:5, u:4)
    2    [holding=balloon] [size=short]    (t:2, u:1)


  Unfriendly-outhypo
    #    cpx
    1    [holding=sword,flag] [size=short..medium] [home=Moons,Sirius]    (t:7,
u:6)
    2    [size=medium..tall] [season=sum,fall] [home=Moons,Sirius]    (t:5, u:4)
```

```
Neutral-outhypo
  #    cpx
  1    [size=tall] [season=spr,win]    (t:3, u:3)
  2    [size=medium] [home=Planets]    (t:2, u:2)
  3    [smiling=no] [holding=sword] [home=Halley_Comet]    (t:1, u:1)
```



Examples of Robots

Friendly — 1 Neutral — 3
Unfriendly — 2 Ambiguous examples — #

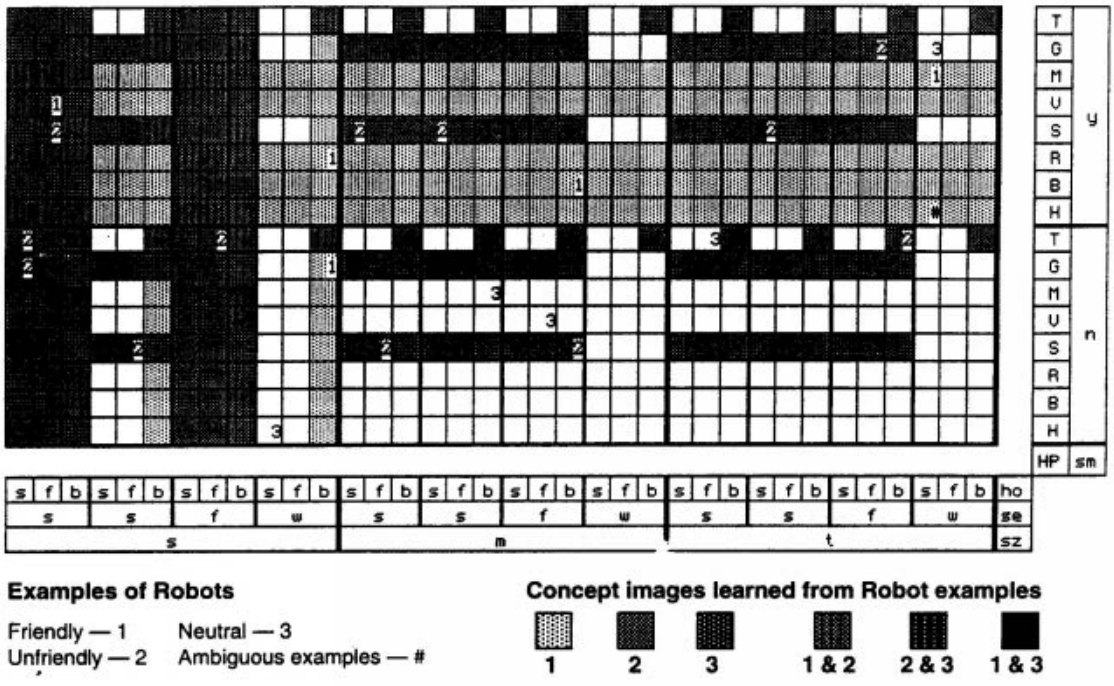
Concept images learned from Robot examples

1 2 3 1 & 2 2 & 3 1 & 3

Figure 9. Run #4: The separation of rules is encouraged .

```
***  Run #5.  In vl mode, the rules are applied sequentially.  That is, the
***  rules for the Unfriendly class assume that none of the rules for the
***  Friendly class were satisfied.  No rules are needed for the last class;
***  the fact that none of the others were satisfied is sufficient to classify
***  an example in this class.

parameters
run    mode    ambig    trim    wts    maxstar    echo      criteria    verbose
 5     vl      pos      mini    cpx     10        p          default      0


Friendly-outhypo
  #    cpx
  1    [smiling=yes] [home=Planets,Orion,Halley_Comet]    (t:5, u:4)
  2    [holding=balloon] [size=short]    (t:2, u:1)


Unfriendly-outhypo
```

```
#    cpx
1    [season=spr,sum,fal] [home=Ganymede,Sirius]    (t:8, u:7)
2    [size=short] [season=spr,fal]    (t:3, u:2)
3    [holding=balloon] [home=Titan]    (t:1, u:1)

Neutral-outhypo
  #    cpx
  1      (t:6, u:6)
```



Examples of Robots

Friendly — 1 Neutral — 3
Unfriendly — 2 Ambiguous examples — #

Concept images learned from Robot examples

1 2 3 1 & 2 2 & 3 1 & 3

Figure 10. Run #5: Rules to be evaluated in "1 then 2 else 3" order. Rule for class 3 that covers whole representation space is not visualized.

```
***  Run #6.  The previous runs showed "minimal complexity" trimming,
***  in which superfluous conditions were dropped, and the remaining ones
***  were specialized to only show the actual values found in the training
***  set.  This run uses "characteristic" mode, in which conditions are
***  not dropped.  The resulting rules are most specific.

  parameters
  run    mode    ambig    trim    wts    maxstar  echo      criteria  verbose
   6     ic      pos      spec    cpx     10      p          default     0


  Friendly-outhypo
    #    cpx
    1    [smiling=yes] [season=spr,fal,win] [home=Planets,Orion,Halley_Comet]
(t:5, u:5)
```

```
    2    [smiling=no] [holding=balloon] [size=short] [season=win]
[home=Ganymede]    (t:1, u:1)


  Unfriendly-outhypo
    #    cpx
    1    [season=spr,sum, fal] [home=Ganymede,Sirius]    (t:8, u:8)
    2    [smiling=no] [holding=sword,flag] [size=short] [season=spr,fal]
[home=Moons]    (t:2, u:2)
    3    [smiling=no] [holding=balloon] [size=tall] [season=fal] [home=Titan]
(t:1, u:1)


  Neutral-outhypo
    #    cpx
    1    [smiling=no] [size=short..medium] [season=sum,fal,win]
[home=Planets,Halley_Comet]    (t:3, u:3)
    2    [holding=sword,flag] [size=tall] [season=spr,win]
[home=Moons,Halley_Comet]    (t:3, u:3)
```

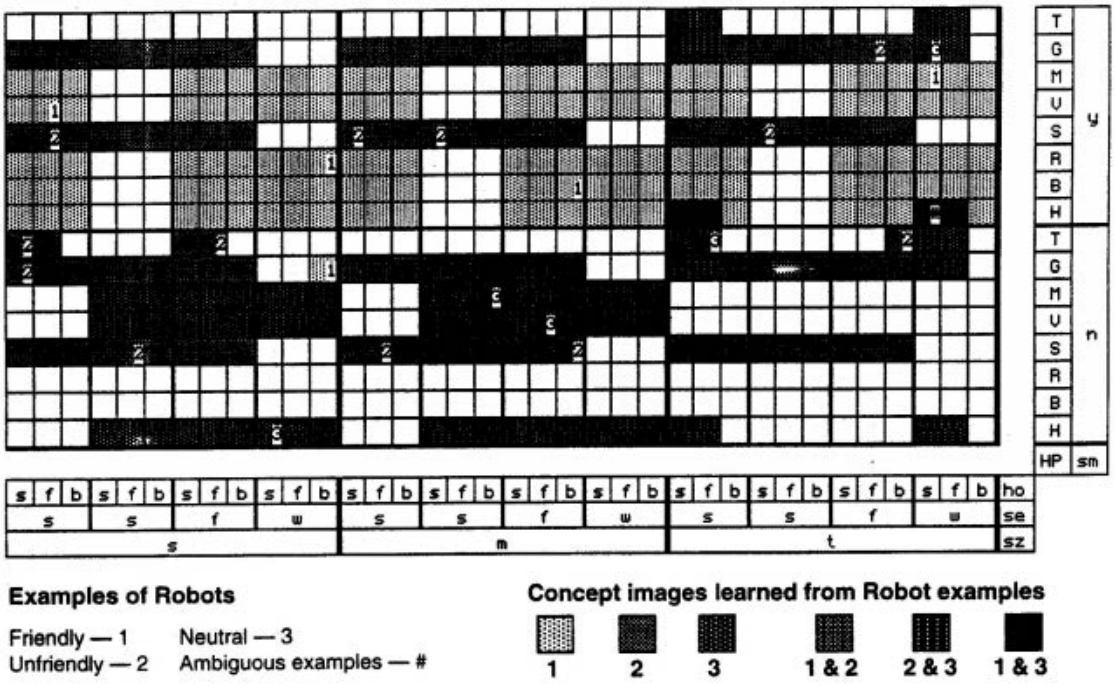


Figure 11. Run #6: Maintaining very specific information

```
***  Run #7 involves the learning of "discriminant rules" -- maximally general
***  ones providing only the information necessary to discriminate among
***  the classes.  Unlike minimal complexity rules, no specialization is done
***  on the value sets.  Instead these rules carve out a wide range of
***  values, some of which may not have been covered by any training examples.

  parameters
```

```
  run    mode    ambig    trim    wts    maxstar  echo      criteria  verbose
   7      ic      pos      gen     cpx     10      p          default     0


  Friendly-outhypo
    #    cpx
    1    [smiling=yes] [home=Planets,Orion,Halley_Comet]    (t:5, u:5)
    2    [holding=balloon] [size=short] [home=Moons]    (t:1, u:1)


  Unfriendly-outhypo
    #    cpx
    1    [season=spr,sum,fal] [home=Ganymede,Sirius]    (t:8, u:8)
    2    [smiling=no] [size=short] [season=spr,sum,fal] [home=Moons]    (t:2,
u:2)
    3    [holding=balloon] [home=Titan]    (t:1, u:1)


  Neutral-outhypo
    #    cpx
    1    [smiling=no] [home=Planets,Halley_Comet]    (t:3, u:3)
    2    [size=medium..tall] [season=spr,sum,win] [home=Moons,Halley_Comet]
(t:3, u:3)
```

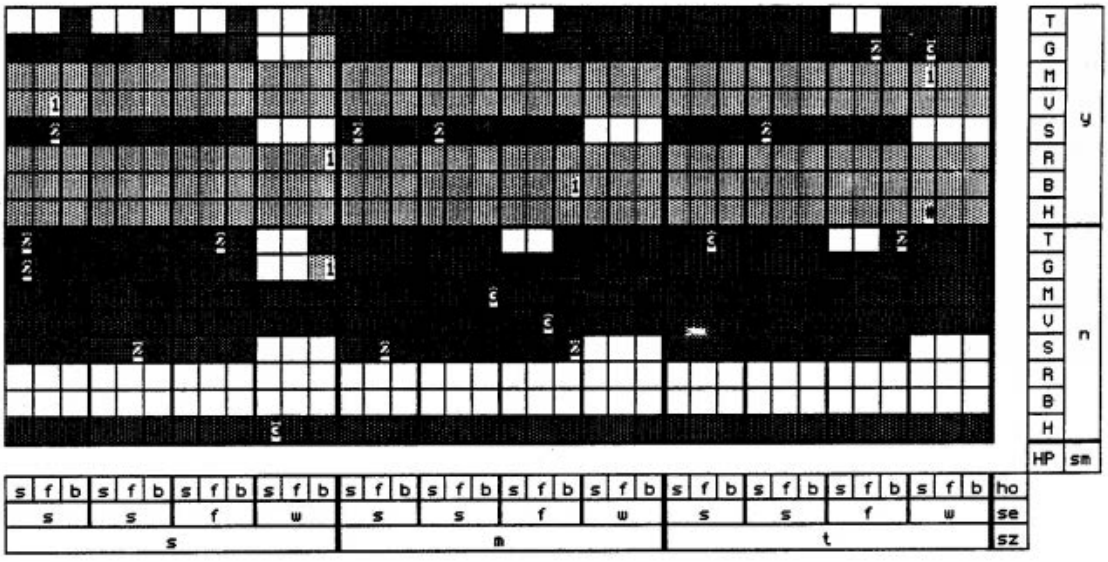


Figure 12. Run #7: A maximally general ruleset

```
***  Run #8 differs from the first run only in its maxstar parameter.  With
***  maxstar reduced to 1, the program does not retain as many intermediate
***  results, thereby reducing its ability to find the optimal rule.  The
```

```
***  rules found here are somewhat more complex than before (note that there
***  are now three Neutral rules.)

  parameters
  run    mode    ambig    trim    wts    maxstar  echo      criteria  verbose
   8     ic      pos      mini    cpx     1       p          default     0


  Friendly-outhypo
    #    cpx
    1    [smiling=yes] [home=Planets,Orion,Halley_Comet]    (t:5, u:4)
    2    [holding=balloon] [size=short]    (t:2, u:1)


  Unfriendly-outhypo
    #    cpx
    1    [season=spr,sum,fal] [home=Ganymede,Sirius]    (t:8, u:7)
    2    [smiling=no] [holding=sword,balloon] [season=spr,fal] [home=Moons]
(t:2, u:2)
    3    [smiling=no] [holding=flag] [size=short]    (t:2, u:1)


  Neutral-outhypo
    #    cpx
    1    [holding=sword] [season=win] [home=Ganymede,Halley_Comet]    (t:3, u:3)
    2    [size=medium] [home=Planets]    (t:2, u:2)
    3    [smiling=no] [holding=flag] [size=tall]    (t:1, u:1)
```

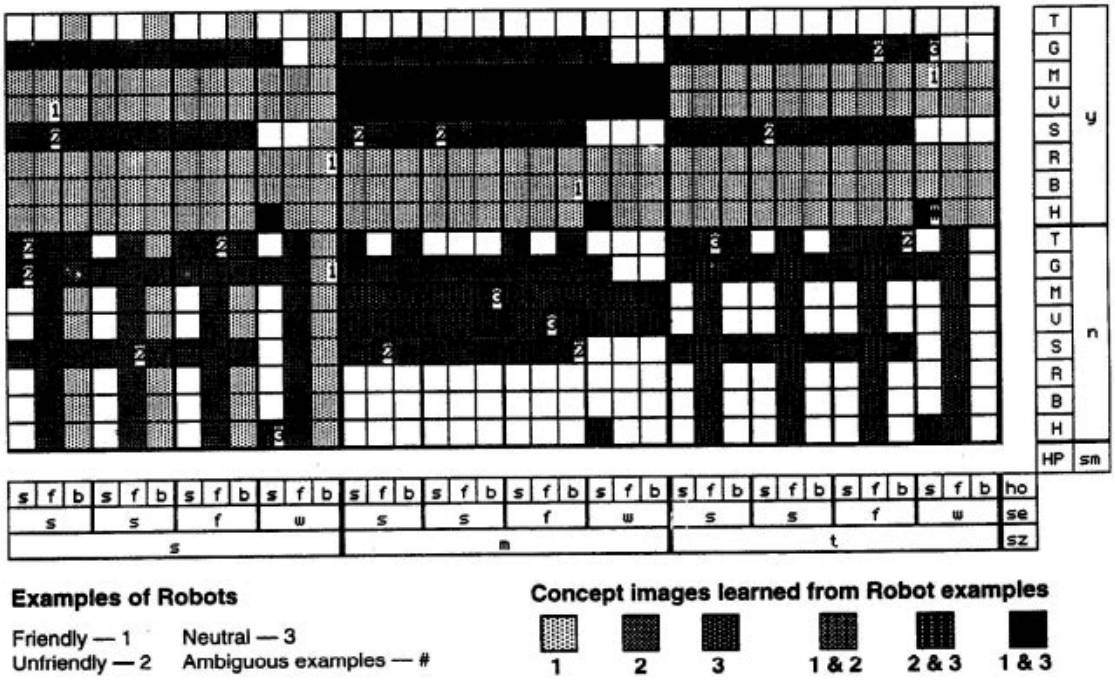


Figure 13. Run #8: The result of trading performance for efficiency

```
***  Run #9 uses a "minimum cost" criterion to select among candidate rules.
***  With a high cost assigned to "home", the program will try to generate
***  rules that do not involve this variable.  It is only sometimes possible.
***  Also, verbose is set to 1 in this run; as a result, the processing time
***  (it was run on a SparcStation 2) is given.

  parameters
  run    mode    ambig    trim    wts    maxstar  echo      criteria  verbose
   9     ic      pos      mini    cpx     10      pc         lowcost     1


  lowcost-criteria
    #    criterion   tolerance
    1    mincost       0.00
    2    maxnew        0.10


  Friendly-outhypo
    #    cpx
    1    [smiling=yes] [home=Planets,Orion Halley_Comet]    (t:5, u:4)
    2    [holding=balloon] [size=short]    (t:2, u:1)


  Unfriendly-outhypo
    #    cpx
    1    [home=Sirius]    (t:7, u:7)
    2    [size=tall] [season=fal]    (t:2, u:2)
    3    [holding=sword] [season=spr]    (t:2, u:1)
    4    [size=short] [season=fal]    (t:1, u:1)


  Neutral-outhypo
    #    cpx
    1    [size=tall] [season=spr,win] [home=Moons,Halley_Comet]    (t:3, u:2)
    2    [home=Halley_Comet]    (t:2, u:1)
    3    [holding=flag] [size=medium] [season=fal]    (t:1, u:1)
    4    [holding=balloon] [season=sum]    (t:1, u:1)


This learning used:
  System time:       0.300 seconds
  User time:         0.00  seconds
```

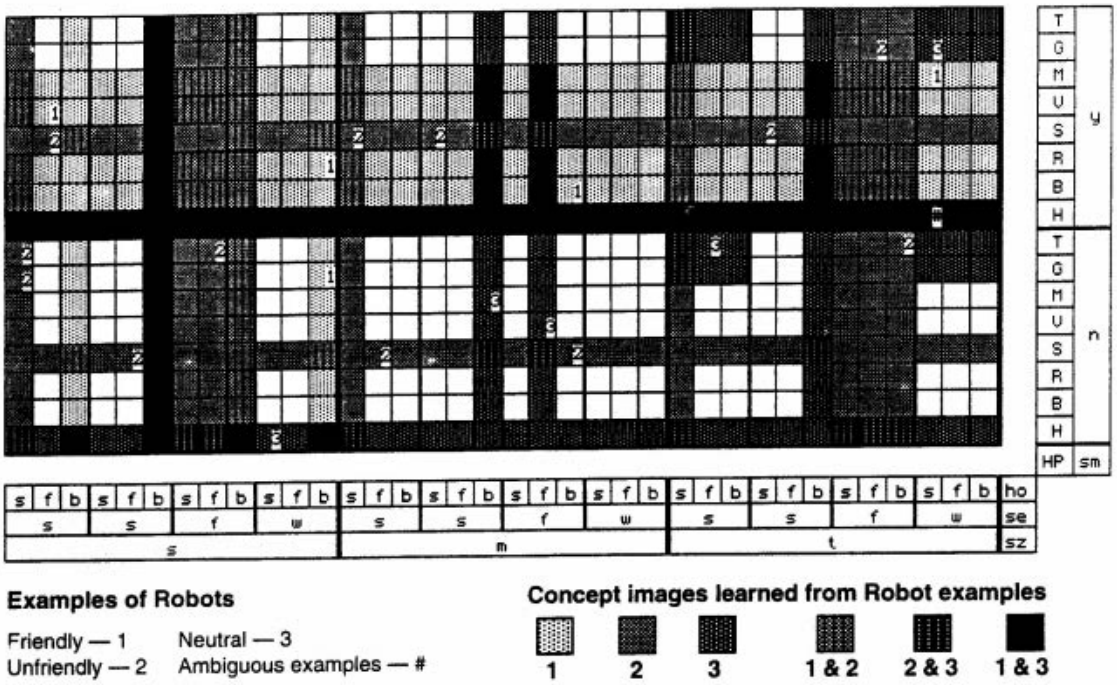


Figure 14. Run #9: Rules that attempt to avoid "Home Planet"

6.2 *Learning Example1 Problem by Larson and Michalski (1975)*

We repeat here the experiment "Example 1," an abstract classification problem defined by Larson & Michalski (1975). Our goal is to show the differences in running AQ7 and AQ15c. The first difference is in input data formulation. AQ7 uses a fixed and encrypted input specification. It is hard to read such a specification when additional comments are not provided. On the other hand AQ15c has a clear structure: parameters, knowledge structure definitions, and data specification—all in the form of annotated tables. The specification of knowledge structures allows for flexible definition of various types, and cost assignment. The size of variables is unlimited. Each class of examples is in a separate table.

AQ7 output carries many specific comments regarding AQ execution, such as size of Star, sizes of intermediate Stars, etc. AQ15c output is compact and controlled by the "echo" parameter.

In general, AQ15c input and output specifications are much more user-friendly than AQ7.

The first three runs listed below use the same mode and trim parameter settings as those reported for AQ7. The results, i.e. rules generated by both programs are identical for the three modes: IC, DC, and VL. The execution time was 6 seconds for AQ7 on an IBM 360/75 vs. 0.05 seconds for AQ15c on a Sun SPARCstation. Runs 4 to 7 provide additional variations of parameter settings (not available for AQ7). These input and output files are prowith AQ15c under the name example1run*x*. (*x* specifies the run number). Input files end with .aqin and output files end with .aqout. So, for example this first input file will be found under the name example1.run1.aqin.

```
parameters
  run    mode     ambig    trim     wts     maxstar   echo      criteria   verbose
   1     dc       pos      mini     cpx      11       pvse       default      1


  variables
    #    type      size      cost    name
    1    nom          4     10.00    x1.x1
    2    nom          3      1.00    x2.x2
    3    nom          4      1.00    x3.x3
    4    nom          4      1.00    x4.x4


  Class0-events
    #    x1   x2   x3   x4
    1    0     0     0     0
    2    0     2     0     2
    3    0     1     0     3
    4    1     1     0     1


  Class1-events
    #    x1   x2   x3   x4
    1    0     1     1     2
    2    1     0     1     1
    3    1     1     1     3
    4    1     2     1     0


  Class2-events
    #    x1   x2   x3   x4
    1    2     1     0     2
    2    3     1     0     0
    3    2     1     2     3
    4    3     1     2     0
    5    3     1     2     3


  Class3-events
    #    x1   x2   x3   x4
    1    2     0     1     2
    2    2     2     1     0
    3    2     0     2     3
    4    2     2     2     0
    5    2     2     2     3


  Class4-events
    #    x1   x2   x3   x4
    1    0     0     3     0
    2    0     2     3     1
    3    0     2     3     3
    4    1     0     3     3
    5    1     2     3     0


  Class0-outhypo
    #    cpx
```

```
   1    [x1=0,1] [x3=0]    (t:4, u:4)

  Class1-outhypo
    #    cpx
    1    [x1=0,1] [x3=1]    (t:4, u:4)

  Class2-outhypo
    #    cpx
    1    [x1=2,3] [x2=1]    (t:5, u:5)

  Class3-outhypo
    #    cpx
    1    [x1=2] [x2=0,2]    (t:5, u:5)

  Class4-outhypo
    #    cpx
    1    [x1=0,1] [x3=3]    (t:5, u:5)


This learning used:
  System time:       0.050 seconds
  User time:         0.00  seconds


  parameters
  run    mode     ambig    trim    wts    maxstar  echo     criteria  verbose
   2     vl       pos      mini    cpx     10      p         default     1

  Class0-outhypo
    #    cpx
    1    [x1=0,1] [x3=0]    (t:4, u:4)

  Class1-outhypo
    #    cpx
    1    [x1=0,1] [x3=1]    (t:4, u:4)

  Class2-outhypo
    #    cpx
    1    [x2=1]    (t:5, u:5)

  Class3-outhypo
    #    cpx
    1    [x1=2]    (t:5, u:5)

  Class4-outhypo
    #    cpx
    1      (t:5, u:5)
```

```
This learning used:
  System time:       0.050 seconds
  User time:         0.00  seconds


  parameters
  run    mode    ambig    trim    wts    maxstar  echo      criteria  verbose
   3     ic      pos      mini    cpx     10      p          default     1

  Class0-outhypo
    #    cpx
    1    [x1=0,1] [x3=0]    (t:4, u:4)

  Class1-outhypo
    #    cpx
    1    [x1=0,1] [x3=1]    (t:4, u:4)

  Class2-outhypo
    #    cpx
    1    [x1=2,3] [x2=1]    (t:5, u:5)

  Class3-outhypo
    #    cpx
    1    [x1=2] [x2=0,2]    (t:5, u:5)

  Class4-outhypo
    #    cpx
    1    [x3=3]    (t:5, u:5)


This learning used:
  System time:       0.067 seconds
  User time:         0.00  seconds


  parameters
  run    mode    ambig    trim    wts    maxstar  echo      criteria  verbose
   4     ic      pos      spec    cpx     10      p          default     1

  Class0-outhypo
    #    cpx
    1    [x1=0,1] [x3=0]    (t:4, u:4)

  Class1-outhypo
    #    cpx
```

```
  1   [x1=0,1] [x3=1]   (t:4, u:4)

Class2-outhypo
  #   cpx
  1   [x1=2,3] [x2=1] [x3=0,2] [x4=0,2,3]   (t:5, u:5)

Class3-outhypo
  #   cpx
  1   [x1=2] [x2=0,2] [x3=1,2] [x4=0,2,3]   (t:5, u:5)

Class4-outhypo
  #   cpx
  1   [x1=0,1] [x2=0,2] [x3=3] [x4=0,1,3]   (t:5, u:5)


This learning used:
  System time:      0.083 seconds
  User time:        0.00  seconds


  parameters
  run   mode    ambig   trim    wts   maxstar  echo     criteria  verbose
   5    ic      pos     gen     cpx    10      p         default     1

  Class0-outhypo
    #   cpx
    1   [x1=0,1] [x3=0,2]   (t:4, u:4)

  Class1-outhypo
    #   cpx
    1   [x1=0,1] [x3=1,2]   (t:4, u:4)

  Class2-outhypo
    #   cpx
    1   [x1=2,3] [x2=1]   (t:5, u:5)

  Class3-outhypo
    #   cpx
    1   [x1=2,3] [x2=0,2]   (t:5, u:5)

  Class4-outhypo
    #   cpx
    1   [x3=3]   (t:5, u:5)


This learning used:
  System time:      0.067 seconds
```

```
User time:        0.00  seconds


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

Class0-outhypo
  #    cpx
  1    [x1=0,1] [x3=0]    (t:4, u:4)

Class1-outhypo
  #    cpx
  1    [x1=0,1] [x3=1]    (t:4, u:4)

Class2-outhypo
  #    cpx
  1    [x1=2,3] [x2=1]    (t:5, u:5)

Class3-outhypo
  #    cpx
  1    [x1=2] [x2=0,2]    (t:5, u:5)

Class4-outhypo
  #    cpx
  1    [x3=3]    (t:5, u:5)


This learning used:
  System time:       0.017 seconds
  User time:         0.00  seconds


parameters
run    mode     ambig    trim    wts    maxstar  echo      criteria  verbose
 7     ic       pos      mini    cpx     10      p         mincost     1

Class0-outhypo
  #    cpx
  1    [x2=0,2] [x3=0]    (t:2, u:2)
  2    [x3=0] [x4=1,3]    (t:2, u:2)

Class1-outhypo
  #    cpx
  1    [x1=0,1] [x3=1]    (t:4, u:4)

Class2-outhypo
```

```
  #   cpx
  1   [x2=1] [x3=2]    (t:3, u:3)
  2   [x2=1] [x3=0] [x4=0,2]    (t:2, u:2)


Class3-outhypo
  #   cpx
  1   [x1=2] [x2=0,2]    (t:5, u:5)


Class4-outhypo
  #   cpx
  1   [x3=3]    (t:5, u:5)


This learning used:
  System time:       0.100 seconds
  User time:         1.00  seconds
```

7. References

Bergadano, F., Matwin, S., Michalski, R.S., and Zhang, J., "Learning Flexible Concept Descriptions Using a Two-Tiered Knowledge Representation: Ideas and a Method," *Reports of the Machine Learning and Inference Laboratory*, MLI-88-4, Center for Artificial Intelligence, George Mason University, 1988.

Hong, J., Mozetic, I., Michalski, R.S., "AQ15: Incremental Learning of Attribute-Based Descriptions from Examples, the Method and User's Guide," *Reports of the Intelligent Systems Group*, University of Illinois at Urbana-Champaign, ISG 86-5, May, 1986.

Michalski, R.S., "Recognition of Total or Partial Symmetry in a Completely or Incompletely Specified Switching Function," *Proceedings of the IV Congress of the International Federation on Automatic Control (IFAC)*, Vol. 27 (Finite Automata and Switching Systems), pp. 109-129, Warsaw, June 16-21, 1969.

Michalski, R.S, "A Theory and Methodology of Machine Learning, in Michalski, R.S, Carbonell, J.G. and Mitchell, T.M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Tioga Publishing Company, 1983, pp. 83-134.

Michalski, R.S., and Chilausky, R.L., "Learning By Being Told and Learning From Examples: An Experimental Comparison of the Two Methods of Knowledge Acqusition in the Context of Developing an Expert System for Soybean Disease Diagnosis," *Policy Analysis and Information Systems,* Vol. 4, No. 2, 1980.

Michalski, R.S., Kerschberg, L., Kaufman, K. and Ribeiro, J., "Mining for Knowledge in Databases: The INLEN Architecture, Initial Implementation and First Results," *Journal of Intelligent Information Systems: Integrating AI and Database Technologies*, Vol. 1, No. 1, August 1992, pp. 85-113.

Michalski, R.S. and Larson, J., "AQVAL/1 (AQ7) User's Guide and Program Description." Report No. 731, Department of Computer Science, University of Illinois, Urbana, June 1975.

Michalski, R.S. and Larson, J., "Selection of Most Representative Training Examples and Incremental Generation of VL_1 Hypotheses: The Underlying Methodology and the Description of Programs ESEL and AQ11," Report No. 867, Department of Computer Science, University of Illinois, Urbana, May 1978.

Michalski, R.S. and Larson, J., "Incremental Generation of VL_1 Hypotheses: The Underlying Methodology and the Description of Program AQ11," ISG 83-5, UIUCDCS-F-83-905, Department of Computer Science, University of Illinois, Urbana, January 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.

Reinke, R.E., "Knowledge Acquisition and Refinement Tools for the ADVISE Meta-Expert System," University of Illinois at Urbana-Champaign, Master's Thesis, 1984.

Thrun, S.B., Bala, J., Bloedorn, E., Bratko, I., Cestnik, B., Cheng, J., De Jong, K., Dzerowski, 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., Zhang, J., (1991). "The MONK'S Problems: A Performance Comparison of Different Learning Algorithms," *(revised version)*, Carnegie Mellon University, Pittsburgh, PA, CMU-CS-91-197.

Wnek, J., "DIAV 2.0 User Manual: Specification and Guide through the Diagrammatic Visualization System," *Reports of Machine Learning and Inference Laboratory*, MLI 95-5, George Mason University, Fairfax, VA, 1995.

Appendix: Programmer's Guide to Data Structures

A.1 Introduction

The most important change in the implementation involved the binary representation of the complex/selector/event data. The original Pascal data structure supporting the complex/selector/event limited the size of individual structures, as well as the number of structures. This was due to the static allocation of memory. This was not a trivial limitation. On the PC version this limit was approximately 200 events and 12 variables. The new C version uses dynamic memory allocation, allowing the user to analyze much more input data.

As in the original Pascal implementation, decision classes are organized hierarchically in a tree structure. Each node in the tree represents a class. A complex is a variant record representing a complex from a rule or an individual selector. As opposed to the original implementation, event information is stored in a separate structure. Future implementation will also provide a separate structure for selector information. This change reduces memory use and improves speed.

The essential building block of the complex/selector/event structure is called "bits." It is the address of the contiguous memory location holding the complex information. Consider a simple event space as follows:

There are 4 variables: x1, x2, x3, x4.

Each of these variables has the following domain sizes:
x1: 3 (0..2)
x2: 2 (0..1)
x3: 4 (0..3)
x4: 2 (0..1)

The complex Comp has the following variable values:
x1=1, x2=0, x3=3, x4=1

The size of the complex is 3+2+4+2=11.

Two bytes (16 bits) will be allocated for storage. The last five last bits of the second byte are unused and are zeroed. The complex bytes will look as follows:

01010000 10100000

A.2 New Data Types

typedef unsigned char * bits
This is the essential data type in the new implementation of AQ. Every variable of this type holds the pointer to the contiguous memory area storing the complex/selector/event.

```
typedef struct Complex {
pcomplex next;
...
union {
struct {
  bits compexbits;
  ...
}
```

```
struct {
  bits selectorbits;
  ...
}
}v;
}Complex;
typedef struct Complex *pcomplex;
```

Data type pcomplex is holds the pointer to the variant record containing information about complex and selector. In addition to storing the core logical part of complex or selector, it holds some other data needed in the algorithm. The previous implementation of AQ also included event information in this data type. This version has a separate data type for holding the event information. In future implementations, the pcomplex data type will pertain only to the complex from a rule. Selector data will be kept by separate structures.

```
typedef struct Event {
pevent next;
bits eventbits;
....
} Event;
typedef struct Event *pevent;
```

Type pevent holds the pointer to the data area containing information about events. Note that the eventbits part of Event is of the same type as complexbits and selectorbits in the Complex type.

```
typedef struct locate_ {
int firstbyte;
unsigned char firstbit_;
}
```

Type locate_ holds the pointer to the memory area containing information about variable configuration in the complex/selector/event. Firstbyte holds the byte number within the complex where the feature's bits (variable data) starts. Firstbit holds the bit number where the feature starts.

A.3 New Global Variables

The size of the complex/selector/event is determined in the setup function and is represented by the global variables *size_of_complex_in_bits* and *size_of_complex_in_bytes*. The bitwise location of each feature is kept by the dynamic array pointed by *firstbit*. The size of each variable is kept in the dynamic array pointed by *varsize*. In addition to these variables there is a structure type locate_ holding the bytewise and bitwise locations of each variable. This approach reduces the overhead associated with the access to features. Also there is no longer a need to create special bitwise masks to access the features like in previous implementations. Some operations in the new implementation are done without entering the internal structure of the particular complex/selector/event - we call such operations bytewise as opposed to bitwise functions that deal with the internal "partitioning" of the complex/selector/event .

From previous example:
comp = 01010000 10100000

If the presented complex name is comp and the memory location of its first byte is 100. The allocated space for it consists of two bytes.

comp = 100 (address of the first byte in complex)

size_of _complex_in_bits = 11
size_of_complex_in_bytes = 2
number_of _variables = 4

The domain size of each variable is represented by the pointer *varsize*. In this case :

*varsize = 3 (size of first domain)
*(varsize + 1) = 2 (size of second domain)
*(varsize + 2) = 4 (size of third domain)
*(varsize + 3) = 2 (size of fourth domain)

Global variable *positions_* is the pointer to data type locate_.

positions_->firstbyte = 0 (the byte number where the first feature starts)
(positions_ +1)->firstbyte = 0 (the byte number where the second feature starts)
(positions_ +2)->firstbyte = 0 (the byte number where the third feature starts)
(positions_ +3)->firstbyte = 1 (the byte number where the fourth feature starts)

positions_->firstbit = 0 (bit location of the first feature)
(positions_ +1)->firstbit = 3 (bit location of the second feature)
(positions_ +2)->firstbit = 5 (bit location of the third feature)
(positions_ +3)->firstbit = 1 (bit location of the fourth feature)

Global variable *firstbit* is the pointer to the array containing the bitwise distances between the features and the beginning of complex/event/selector.

*firstbit = 0 (bitwise distance between first feature and the beginning of complex)
*(firstbit+1) = 3 (bitwise distance between second feature and the beginning of complex)
*(firstbit+2) = 5 (bitwise distance between third feature and the beginning of complex)
*(firstbit+3) = 9 (bitwise distance between fourth feature and the beginning of complex)

A.4 New Functions

Most of our effort was put to streamline the basic set and logical operations in the new C version. Following are short descriptions of some new functions associated with the core of AQ15c. Most of the "high level" functions like traversetree, formrule, coverll etc remained unchanged.

The core of new logical and set operations is included in the new file called aqs.c. This file contains routines that deal with complexes at the lowest level. They can be viewed as a set of basic tools or drivers needed for the new C version of AQ15.

`void compl_allocation``(bits *c)`
This routine allocates the space needed to hold the complex/selector/event pointed by c. It uses the global variable *size_of_complex_in_bytes*.

`void put_1``(bits c, int _bytenr, unsigned char _bitnr)`
This function places a one bit in the location pointed by the above parameters. C is the pointer to the beginning of the complex/selector/event, _bytenr represents the byte number in complex under consideration, _bitnr is the bit number in the byte. The beginning of the complex/selector/event has a _bytenr and _bitnr of zero.

`void put_0``(bits c, int _bytenr, unsigned char _bitnr)`

Function similar to put_1 but places zero in the location- pointed by the above parameters. Parameters are the same as in put_1.

void invert_bit(bits c, int _bytenr, unsigned char _bitnr)
This function inverts the bit value in the location pointed to by the above parameters. The parameters are the same as in put_1.

char check_bit_status(bits c, int _bytenr, unsigned char _bitnr)
Check_bit_status returns one if the bit pointed by the above parameters is set to one. Otherwise it returns zero. See put_1 for the description of parameters.

void allocate_complexes(void)
This function is called by setup after the size of the complex is determined. It allocates complexes used locally in frequently called functions.

void create_bytes(void)
Create_bytes is called by setup. After domain and variable sizes are acquired, create_bytes establishes the configuration of the complex/selector/event structure. This function creates global variables holding information of the bytewise and bitwise locations of each feature. (See GLOBAL VARIABLES)

int subset_compl(bits c1, bits c2)
Subset_compl returns one if complex c1 is a proper subset of complex c2, otherwise zero is returned.

char check_if_sel_fill(bits a, int fid)
The function returns 1 if complex a has all bits of feature number fid set to 1. Fid must be in range of zero to number_of _variables - 1.

void copy_sel(bits a, bits b, int fid)
Copy_sel copies all bits assigned to feature fid of complex a to the same feature number of complex b. Fid must be in range of zero to number_of_variables - 1.

void blank_feature(bits c, int fid)
Blank_feature sets all bits belonging to the feature fid to 0. Fid must be in range from 0 to number_of_variables - 1

void copy_complex(bits c1, bits c2)
This function copies the complex pointed by c1 to c2. The operation is bytewise, this means that no feature check is performed. It is different from original Pascal implementation where such operation had to performed on the array of sets.

void intersect_compl(bits c1, bits c2, bits c2)
The function intersects logically complexes c1 and c2 and places the resulting complex into c3. This operation is bytewise.