AQ15: INCREMENTAL LEARNING OF ATTRIBUTE-BASED DESCRIPTIONS FROM EXAMPLES, THE METHOD AND USER'S GUIDE

by

Jiarong Hong
Igor Mozetic
R. S. Michalski

ISG 86-5, UIUCDCS-F-86-949, Department of Computer Science, University of Illinois, Urbana, May 1986.

AQ15: Incremental Learning of Attribute-Based Descriptions from Examples

The Method and User's Guide

by

Jiarong Hong*
Igor Mozetic**
Ryszard S. Michalski

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

File No. UIUCDCS-F-86-949
ISG Report 86-5

July, 1986

---

*On leave from: *Harbin Institute of Technology, Harbin, The People's Republic of China*

**On leave from: *Josef Stefan Institute, Ljubljana, Yugoslavia*

This research was supported in part by the National Science Foundation under Grant No. DCR 84-06801, Office of Naval Research under Grant No. N00014-82-K-0186 and Defense Advanced Research Project Agency under grant N00014-K-85-0878, and, for the second author, by the Slovene Research Council.

TABLE OF CONTENTS

ABSTRACT

AQ15 is a program that *incrementally* learns decision rules from examples and counterexamples of decisions, and possibly previously learned rules. In learning the rules, the program uses (i) *background knowledge* that consists of rules and concepts the program already knows, (ii) the definition of descriptors and their types, and (iii) a *preference criterion* that evaluates competing candidate hypotheses. Each training example characterizes an object (situation, process, etc.) and specifies the correct decision associated with that object. The generated decision rules are expressed as symbolic descriptions involving relations among object attributes. The rules are optimized according to a flexible criterion selected by the user. The criterion measures the quality of the rules from the viewpoint of the specific problem under consideration. The user may also specify initial decision hypotheses to be used for incremental learning. In this case, the program will improve them until they are consistent with all available facts. The *AQ15* program also has the capability of constructive induction. It can use new descriptions as the input data to simplify previously generated decision rules. Finally, the *AQ15* program includes a decision rule hypothesis testing utility.

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

1. INTRODUCTION

The *AQ15* program learns decision rules by performing inductive inference on examples and optional initial decision rules. Training examples are expressed as conjunctions of attribute values, and initial or induced decision rules are logical expressions in disjunctive normal form. The program performs heuristic search through a space of logical expressions, until it finds a decision rule that is satisfied by all positive examples and by no negative ones and optimized by a rule preference criterion. The program implements the *STAR* method of inductive learning (Michalski & Larson 83). Specifically, it is based on the *AQ* algorithm for solving the general covering problem (Michalski 69).

The *AQ15* inductive learning program is the immediate descendant of the *GEM* program. The name *GEM* derives from "Generalization of Examples by Machine". The *GEM* program was originally written in Pascal by Bob Stepp and Mike Stauffer as an improved version of the *AQ7-AQ11* series of programs on CYBER 175 (Michalski 69; Michalski & Larson 75; Michalski & Larson 78; Michalski & Larson 83). The *GEM* program itself was written from scratch; it is not just a modified version of *AQ11*. Later the program was transcribed to Berkeley Pascal running under the Unix operating system on VAX and SUN machines. Then Kaihu Chen contributed the structured variable part, and Robert Reinke implemented the *incremental learning* facility (Reinke 84).

Our contribution was to add many new features, make changes and remove bugs from the program, which makes the program more powerful and useful for many purposes. Specifically, we improved the data structure for handling *extended selectors* and implemented *constructive induction* by using *A-rules and L-rules* (background knowledge). Also, we improved and reimplemented the *incremental learning* part and incorporated the testing facility into the program. In addition, we improved the *trimming facility* and implemented a capability of handling *ambiguous examples*, a capability of inducing *most general, most specific* and *minimal* descriptions, and some other minor features. We also have written the program documentation.

Currently the *AQ15* program is implemented in Berkeley Pascal, and runs under Unix on SUN and VAX machines. It consists of approximately 13000 lines of Pascal code (including comments). The program is still evolving and new features are being added. In general, the program has enough comments to be self explanatory. A document in the appendix provides a general overview of the algorithms and data structures used by the program and may serve as introductory reading. In order to fully understand the structure of *AQ15*, the program listing should be read together with this document.

The paper is organized in the following way. First we explain the **basic concepts and terminology** used throughout the paper. Then we give an **outline** of the *AQ15* algorithm.

Next we give a **user's guide** with examples for using the program. Then we show deficiencies of the current **implementation** and give some ideas for **further work**. Finally, in the appendix, we give a detailed description of the **learning algorithm** and describe of the most important **data structures** and global constants used by the program.

2. BASIC CONCEPTS and TERMINOLOGY

All concepts to be described below are expressed in the VL_1 (**Variable–valued Logic System 1**) (Michalski 75) and APC (**Annotated Predicate Calculus**) (Michalski 83).

Training examples are given to *AQ15* in the form of ***events***. Events belong to decision ***classes***. Given a class, of events belonging to it represent ***positive examples*** of the class, and all other events are its ***negative examples***. For each class, a ***decision rule*** or a ***cover*** is produced that must be satisfied by all positive examples and by no negative ones. The user may supply some decision rules as a starting point for the program. We will call such rules ***initial hypotheses***. Intermediate results during the search for a cover are called ***hypotheses*** or ***partial covers***.

Each event and decision rule is described by extended selectors. An ***extended selector***, or briefly a ***selector***, is a relational statement and is defined as:

TERM REL REFERENCE

where TERM is a ***variable*** an arithmetic expressions of constants and variables, or an ***internal conjunction*** of terms; REL stands for one of the relation symbols: $<$, $<=$, $=$, $<>$, $>=$, $>$; REFERENCE is a value (constant) or the internal disjunction of values.

Extended selectors defined above state that the variable or each arithmetic expression in TERM takes values defined in REFERENCE. The following are examples of extended selectors:

[color = red V white V blue]
[width & height 5]
[temperature 20..25, 50..60]
[length*width & length*height = 36..40]

A *complex* is a conjunction of selectors.
The following are complexes:

[color = red V white V blue] [stripes = 3..13] [stars = 1..50]
[length & 3*width = 12][color = yellow]

A *cover* is a disjunction of complexes. The following cover consists of two complexes:

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

A cover is *satisfied* if any of its complexes are satisfied, while a complex is satisfied if all extended selectors in it are satisfied. An extended selector is satisfied if all variables and expressions in it actually take one of the specified values. Thus, the example cover above can be interpreted: A piece of cloth is a flag, if it satisfies one of the following conditions:

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

Besides events, declarations of variables, and possibly input hypotheses, *AQ15* also needs some *parameters* that specify how the rules should be constructed. The user may select some preference *criteria* according to which the rules are optimized. The criteria measure the quality of the rules from the viewpoint of the specific problem under consideration. All input data given to *AQ15* and output results are in the form of *relational tables*. *AQ15* has also the capability of constructive induction by the *pre-cover processing*. In the pre-cover processing, the arithmetic formulas (A-rules) or VL1 logical assertions (L-rules) given by the user as *background knowledge* are input to *AQ15* . *AQ15* then uses the rules to generate new variables and selects some of them (by a filter) to add the corresponding extended selectors to the input events and hypotheses, which may specialize the descriptions of events and simplify the resulting hypotheses. The reader should consult the user's guide for details about format and examples.

3. ALGORITHM

The Fig. 1 is the flowchart of the *AQ15* algorithm, Fig. 2 and Fig. 3 depict incremental learning and star generation components of this algorithm. The algorithm is described in detail in the APPENDIX.

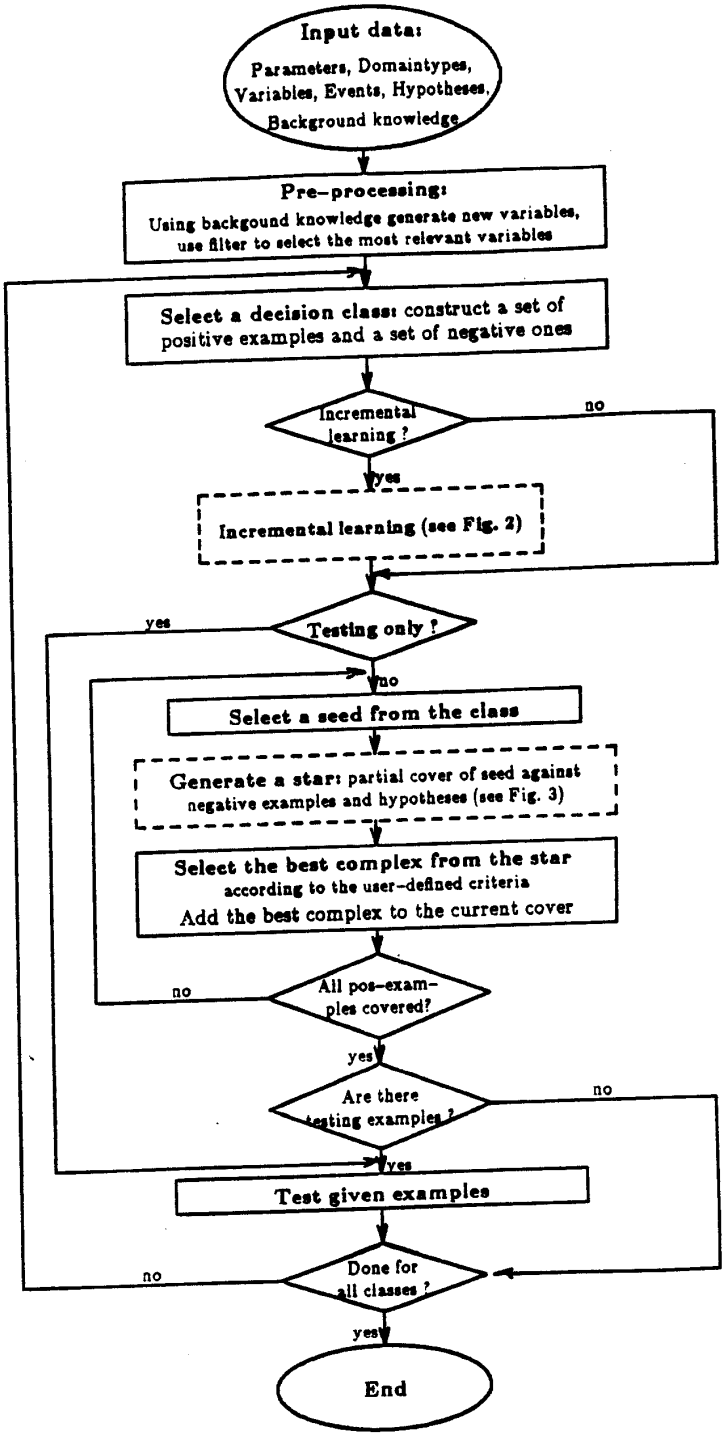


Fig. 1. *AQ15* Algorithm

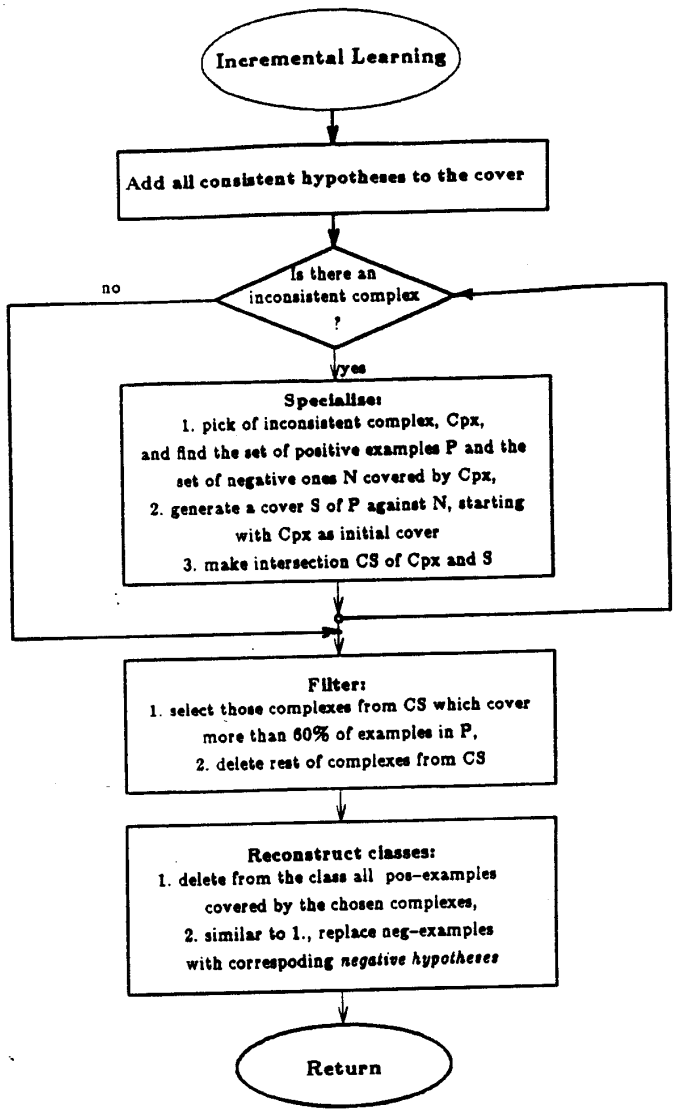


Fig. 2. Incremental Learning

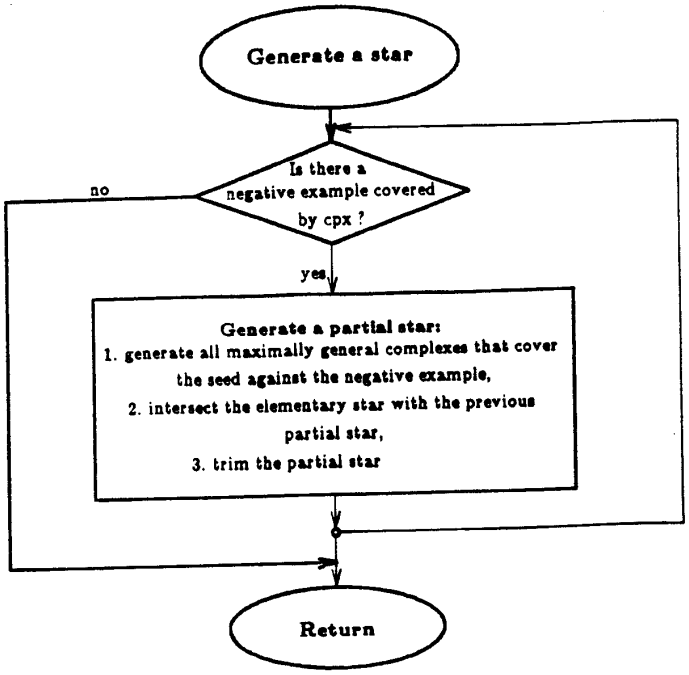


Fig. 3. Star Generation

4. USER'S GUIDE

4.1. Purpose of the Program

The *AQ15* program learns decision rules by performing inductive inference on examples and optional initial decision rules. The program implements the *STAR* method of inductive learning. It is an extension of the *AQ* algorithm.

4.2. Summary of New Features

AQ15 has a number of new features not present in *AQ11* and *GEM*. Some of these are:

1. *Extended selectors* – an extension of the descriptions used in GEM,

2. *Improved incremental learning* – using full Memory and membership, successively adding new learning examples and modifying rules generated so far or generating new rules,

3. *Constructive induction* – using A-rules and L-rules to generate new variables not present in input data and selecting some of them to produce better rules,

4. *Testing* – using truncation of covers and analogical matching to test which rule best matches a test example (Michalski, Mozetic, Hong & Lavrac 86).

4.3. An Introductory Example

The following example is based upon statistics for personal computers existing in the year 1980. Suppose we have 12 examples of personal computers, each described with a set of attributes. Attributes specify interesting characteristics of a computer, such as its software ("Pascal", "Fortran", "Cobol"), type of operating system ("Op_system"), number of floppy disk drives ("Floppies"), presence of hard disk ("Disk"), type of processor ("Processor"), size of memory ("Memory"), and whether it has a printer or not ("Printer"). All computers may be divided into three classes, regarding their cost: under $1000from $1000 to $4000, and over $4000. In the following table each row represents an example of a computer:

Cost = Under1000 <::

Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer
no	no	no	other	0	no	M6502	16	no
no	no	no	other	0	no	M6502	2	no
no	no	no	other	0	no	M6502	4	no
no	no	no	other	0	no	Z80	32	no

Cost = From1000to4000 <::

Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer
no	no	no	other	1	no	M6502	16	no
no	yes	yes	cpm	2	no	I8085	32	no
yes	yes	yes	cpm	1	no	Z80	64	no
no	yes	yes	cpm	2	no	Z80	16	no
no	yes	yes	cpm	1	no	I8085	48	no

Cost = Over4000 <::

Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer
yes	yes	yes	other	1	yes	Z80	128	no
no	no	yes	other	2	no	I8085	64	yes
yes	yes	yes	cpm	1	yes	Z80	280	no

From this set of examples we would like to obtain simple classification rules for deciding costs of computers. For these training events and a criterion requiring the minimum number of terms, *AQ15* produces the the following results:

[Cost = Under1000] <:: [Floppies = 0]

Paraphrase: A computer costs under \$1000 if it does not have any floppy drives.

[Cost = From1000to4000] <:: [Floppies = 1..2] & [Disk&Printer = no]

Paraphrase: A computer costs from \$1000 to \$4000 if it has 1 to 2 floppy drives *and* it has no hard disk *and* no printer.

[Cost = Over4000] <:: [Disk = yes] v
[Printer = yes]

Paraphrase: A computer costs over \$4000 if it has hard disk *or* if it has a printer.

It is obvious that the rules produced by *AQ15* correctly classify all computers in this simple example. However, despite of the simplicity of the example, it is interesting that only three attributes were used for correct classification. The power of the *AQ15* grows when used for solving considerably larger problems, e.g. there are a few thousands of examples, described with few tens of attributes.

4.4. General Table Format

Input to *AQ15* is in the form of relational tables. Relational tables are a convenient format for representing events of the type dealt with by *AQ15*. This also allows the program to be used as an operator by the QUIN relational database system (Spackman 83). The program reads from standard input and writes to standard output.

Input to *AQ15* consists of a single file containing a series of relational tables. A relational table is composed of three parts: a *table-name*, a list of *column names*, and a set of *tuples* (rows) containing the data. In general, columns may be entered in any order. The columns acceptable for each table type are defined in the section describing the table. There must be at least *two columns* and may be at most 40 columns in each table. The length of the tuples and the type of information in them must correspond to the appropriate column names. The each column

heading and each tuple in the table must be on a *single input line* (which may be more than 80 characters long). Individual items on a line are separated by any number of spaces or an exclamation mark (!).

Table names are of two types. First, there are tables which have only a single part name (such as "parameters"). There are also table names which consist of two parts. These are of the form *specific–general*, where *specific* and *general* are each an alphanumeric string. Tables of this type (e.g. the "–names" tables) must have a specific name associated with them because there may be several tables of the same general type. In the table definitions that follow, any table whose name is given with "–" must have a specific name preceding the "–" in program input. For instance, one could choose to have block–names, hoop–names and pyramid–names in the same file. Though these would all be manipulated as "–names" tables, each would need to be distinguished.

Any alphanumeric string (e.g. a specific table name, a variable) entered as input must be a continuous string (without spaces) of any characters except ?!$'"*– and begin with a letter. Throughout the paper we refer to such strings to be of the type *alpha*. Their maximum length is a program constant, currently set to 20.

4.5. Tables Guide and Sample Input

This section gives the format for tables to be used in the *AQ15* input file. For each table, the purpose of the table, the columns used in the table, and the allowed entries in each column are given. Each formal description of a table is accompanied with an example from the sample input relevant to the problem introduced in section 2, i.e., cost of personal computers. The initial set of examples from that problem is augmented with additional examples, and the classification rules already produced are used as initial hypotheses for incremental learning. The final rules

that classify all available examples correctly are given in the last section of the paper.

4.5.1. The title table

The title table is *optional* as it provides only a header for an input file. It is not used in any way by the *AQ15* program. This table must consist of two columns:

- #

  Mandatory column which contains the row number of the text in the next column. Row numbers are consecutive *integers* beginning with "1". sequentially.

- **text**

  Each entry in this *mandatory* column consists of a *string of characters* that are a single line in the title of the input file. If there are any blanks or tabs in the row, the string must be surrounded with quotes. If single quotes appear in the string, double quotes must be used to surround it, and vice versa.

Example:

title

#	**text**
1	"This is a 'title' table of the sample input to *AQ15*."
2	"The sample used throughout the tables guide is an"
3	"example of data, where all types of tables are used."

4.5.2. The **parameters** table

The **parameters** table is *mandatory*. This table contains values which control the execution of the program. All of the parameters have default values, as noted below in parentheses. These columns need not be entered in the table if the default value is acceptable; however, recall that at least two columns must be entered for each table. Each row of the parameters table represents one run of the program. This allows the user to specify many different runs on the same data in a single input file.

- **run** (1..n)

Optional row number. It is an *integer*, the first row must be numbered "1" and rows must be numbered sequentially.

- **mode** (ic)

Optional specification of the way in which *AQ15* is to form rules. Legal values for this column are:

ic — "intersecting covers" mode produces rules that may intersect over areas of the event space where there are no learning events (empty space).
dc — "disjoint covers" mode produces rules that do not intersect at all.
vl — "variable–valued logic" mode produces rules that are order dependent. That is, the rule for class "n" will assume that the rules for the classes "1" through "n–1" are not satisfied.

- **ambig** (neg)

Optional parameter that specifies how to handle ambiguous examples (i.e. examples from more than one class that overlap; two examples overlap if 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.
pos — ambiguous examples are always taken as positive examples for the current class, and are therefore covered by more than one classification rule. If the number of examples is large this may be computationally expensive.
empty — ambiguous examples are ignored, i.e. treated as that they do not exist (empty space), and may or may not be covered by some classification rules. If the number of examples is large this may be computationally expensive.

- **trim** (mini)

Optional **parameter** that specifies forms of covers to be produced. The legal values are:

gen — induced rules are as general as possible, i.e. they involve minimum number of extended selectors, each with maximum number of values, as described above.
mini — rules are as simple as possible, i.e. with the minimum number of extended selectors and minimum number of values. Redundant values are removed from extended selectors in a cover, sequentially for each complex, after complexes are sorted in decreasing order of total events covered. A value is removed if it does not appear in any event newly covered by current complex (i.e. not covered by any previous complex).
spec — rules are as specific as possible, i.e., they involve maximum number of extended selectors, each with minimum number of values. First, the most

general description is taken and for each complex all missing variables are added. Each such variable can take the union of values that appear in events covered by the current complex. Then extended selectors in which all possible values appear are deleted and redundant values are removed from the remaining extended selectors, as above.

- **wts** (cpx)

Optional parameter that causes *AQ15* to associate weights with complexes and extended selectors it produces. Legal values are:

no — no weights.
cpx — two weights are associated with each complex. The first weight is the total number of positive events that the complex covers. The second weight produced is the number of events that this complex, and no other complex in the rule, covers. Complexes are always sorted in decreasing order of the first weight.
all — besides weights on complexes there are also two weights associated with each extended selector. The first weight is the number of positive events that the extended selector covers, and the second weight is the number of negative events covered by the extended selector (independently of other extended selectors). In case of this value only, extended selectors in each complex are sorted in decreasing order of the first weight.

- **maxstar** (10)

Optional parameter that must be an *integer* and can take any value between 1 and 100. This parameter controls the number of alternative solutions kept during a complex formation. A higher number may produce better solutions but will require more computer resources. In general, if a maxstar is approximately equal to the number of variables used, the results are optimal in terms of quality of classification rules produced and computational resources required. The maximum value is a program constant, currently set to 100.

- **evtcovd** (no)

Optional specification of whether or not the names of events covered by a complex are stored in the complex, where by *names* we mean their identification number in the corresponding classes. When doing incremental learning, the names are used to test if the complex covers the events by testing the membership of the set of the names stored in the complex instead of using the covering algorithm which may cost much time.

yes — the names of positive events covered by a complex are stored into the complex, and used to test if the events are covered by the corresponding complex when doing incremental learning.
no — no name is stored.

- **increment** (0)

Optional **parameter** that must be an *integer* and can take any value between 0 and 100. This **parameter** controls n, the number of times incremental learning will be done. n = 100/increment if increment $\neq 0$; otherwise, $n = 1$ which means no incremental learning is required. If increment $\neq 0$, then the sets of examples will be divided into n subsets. For each subset, an incremental learning run is effected using previous results.

~ testonly (no)

Optional specification of whether or not running the program is only for testing events.

yes — running on testing examples without learning task.
no — running on learning examples.

~ testpart (0)

Optional parameter that must be an *integer* and is the percent of randomly chosen testing events. Indirectly, it controls the number of times testing will be done since the number of times = 100/testpart.

~ chosevts (0)

Optional parameter that must be an *integer* and is the percent of randomly chosen testing examples when testing or new learning examples when incremental learning.

~ reinconhyps (yes)

Optional specification of whether or not specialized input hypotheses need to be filtered (see Fig. 3).

yes — select those specialized hypotheses which covers more than 60% of events that were covered by original hypotheses, and delete the rest of the specialized hypotheses. Keep the hypotheses selected as negative examples later, and those examples which are covered by the deleted hypotheses as current positive examples.
no — all specialized hypotheses will be kept for the future use.

~ echo (pcv)

Optional specification of which tables are to be echoed to output. Values in this column consist of a string of characters, each character of which represents a single table to be echoed. There must be no blanks or tabs in this string. Legal characters for the echo column, and the tables they represent are:

t — the title table
p — the parameters table
c — the –criteria tables
d — the domaintypes table
v — the variables table

n	—	the –names tables
s	—	the –structure tables
i	—	the –inhypo tables
h	—	the –ghypo tables
e	—	the –events tables
f	—	the –events tables with new variables
b	—	the –children tables
a	—	the arules tables
l	—	the lrules tables
q	—	the reduction criterion 1 tables
w	—	the reduction criterion 2 tables
x	—	the expression variable tables
0	—	no echo

- **criteria** (default)

Entry in the *optional* column is the name of the criteria table to be used for this run. The name must be of the *alpha* type, and a –criteria table with that name must appear in the input file.

A sample parameters table is shown below. Values shown in the first row are the default values for the parameters, except for the "echo" parameter. Note that the default value for the criteria column is the only –criteria table specific name for which it is not necessary to actually define a table. Since the second row contains the string "mincost" for the criteria column, a table named "mincost–criteria" must be defined. Note also that the columns "run", "mode" and "maxstar" described above are omitted here, as their default values are acceptable.

Example:

parameters

ambig	**trim**	**wts**	**evtcovd**	**echo**	**criteria**	**increment**	**testpart**
neg	mini	cpx	no	pcvb	default	0	0
pos	gen	all	yes	pcis	mincost	20	33

4.5.3. The –criteria tables

All –criteria tables other than "default" *must* be defined. This table type is used to define a lexicographic functional (LEF). The LEF is used by *AQ15* to judge the quality of complexes

formed during learning. A 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 error within each criterion.

A criteria table name consists of two parts — the specific name, which must appear in the "criteria" column of the parameters table (above we used "mincost") and the general name, –criteria. In the example in section 4.2, "mincost" reflects the existence of a table named "mincost–criteria". Any value in the criteria column of the parameters table except "default" must have a corresponding –criteria table and vice versa.

- **#**

The order of this criterion in the LEF. It is an *integer*, the first row must be numbered "1" and rows must be numbered sequentially. This column is *optional*.

- **criterion**

This *mandatory* column specifies the functional which is to be used for this row of the LEF (the rows of the table give the ordering in which the functional will be applied). There are 8 different criteria available, and at most 8 criteria may appear in any –criteria table:

maxnew	—	maximize the number of newly covered positive events, i.e. events that are not covered by previous complexes.
maxtot	—	maximize the total number of positive events covered.
newvsneg	—	maximize the ratio between the numbers of newly covered positive events and all negative events covered. Computationally expensive criterion.
totvsneg	—	maximize the ratio between the total numbers of positive events covered and negative events covered. Computationally expensive criterion.
mincost	—	minimize the total *cost of the variables* used (see 4.5.4).
minsel	—	minimize the number of extended selectors.
maxsel	—	maximize the number of extended selectors.
minref	—	minimize the number of references in extended selectors.

- **tolerance**

This column is *mandatory* and must be a *real* number. The tolerance specifies the relative fractonal uncertainty in the associated criterion. For example, say the best complex in a list had a value of 100 for some criterion, and the tolerance for the criterion was 0.2. The absolute tolerance allowed is computed by multiplying the tolerance by the best value,

yielding an absolute tolerance of 20. Then any complex with a value between 80 and 100 would be regarded as having the same value as the best complex for this criterion.

The first –criteria table shown below is the default. In majority of experiments performed with *AQ15*, it produced the best results. This is the only instance of the –criteria table which need not be entered explicitly. The second –criteria table must be defined, as its specific name appears in the "criteria" column of the parameters table. Note that row enumeration 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

4.5.4. The domaintypes table

The domaintypes table is used to define domains for attributes. This table is *optional*, but it is convenient if several attributes have the same set of possible values. The table consists of four columns, of which at least two must be included:

- **name**

This is the name of the domain being defined and must be of the *alpha* type. The column is *mandatory*.

- **type** (nom)

An *optional* column with type of the domain being defined. Four domain types are legal:

nom — a "nominal" domain consists of discrete, unordered values, which are

different from their internal values.
(e.g. a Processor name is a typical nominal domain).

lin — a "linear" domain consists of discrete, ordered values, which are same as their internal values.
(e.g. number of floppy drives).

int — a "interval" domain has discrete values in an interval, which can be different from their internal values.
(e.g. a special use in event tables).

cyc — a "cyclic" domain has discrete values in a circular order
(e.g. the integers modulo 4).

str — a "structured" domain is in the form of a hierarchical graph
(see 4.7).

The default value for the type is "nominal" domain.

^ **levels** (58)

This column is also *optional* and can take any *integer* value between 1 and 58 specifying the number of possible values. The maximum number of values is a declared program constant, currently set to 58. If not specified, the maximum number of levels, 58, is taken.

^ **cost** (1.00)

A *real* number specifying how "expensive" this variable should be to use compared to other variables. It is used in computing criterion "mincost" in the LEF (see the definition of the -criteria table) and is *optional.* If it is omitted it is assumed that all variables have a cost of 1.00.

^ **transf** (yes)

An *optional* column. If its value is "yes" then the references in examples and hypotheses are treated as internal values, and finally they are converted into their real values defined in the variables table or domaintypes table. If transf = "no" then the values are the same as those in examples and hypotheses, and no conversion is required.

The **domaintypes** table is normally used in conjunction with the variables table and the -names table.

Below are examples of both, the domaintypes and variables table (refer to the next section for explanation).

Example:

domaintypes

name	type	levels
boolean	nom	2
Op_system	nom	2
Floppies	lin	4
Processor	nom	3
Memory	str	8

variables

#	name	cost
1	Pascal.boolean	10
2	Fortran.boolean	10
3	Cobol.boolean	10
4	Op_system	10
5	Floppies	100
6	Disk.boolean	0
7	Processor	0
8	Memory	100
9	Printer.boolean	0

4.5.5. The variables table

The variables table is ***mandatory*** — it specifies the names and domains of the variables used to describe events. It may contain up to five columns, but it must have at least two:

- # (1..n)

  Optional numbering of variable declarations. It is an *integer*, the first row must be numbered "1" and rows must be numbered sequentially.

- **name** (x#)

  Optional column associating a name with the variable, must be of the *alpha* type. If this column is omitted, variables will be given names of the form x#, where # is the number of the row the variable appears in. If a domaintypes table is being used, then the variable name may consist of two parts — "name"."domain-name", separated by a period, where "domain-name" is a string appearing in the name column of the domaintypes table. Alternately, the name in the domaintypes table and variables table may be shared if a domain is unique to a single variable. Maximum number of variables is a program constant, currently set to 60.

- **type** (nom)

  Optional column, with the same meaning as the type column in the domaintypes table. It should be specified if the domaintypes table is not used and the default "nominal" type is not acceptable.

- **levels** (58)

  Optional column, with the same meaning as the levels column in the domaintypes table. It should be specified if the domaintypes table is not used and the default number of levels 58 is not acceptable.

- **cost** (1.00)

  Optional column, with the same meaning as the cost column in the domaintypes table. It should be specified if the domaintypes table is not used and the default cost 1.00 is not acceptable.

The variables table may be used in conjunction with the domaintypes and –names table. In the example above, if the domaintypes table were excluded, then the "type" and "levels" columns would have to appear in the variables table. In that example, the row enumeration may be omitted, as the "cost" column is present, thus including the minimum of 2 columns.

4.5.6. The –names tables

This table is *optional.* The –names table is used to specify names for values in a domain as they appear in input examples and will appear in classification rules. If no –names table appears for a variable or domain, then the values for that domain are assumed to be the integers beginning with "0". The specific name of a –names table must be the name of a domain, as specified in the name column of the domaintypes table. If the domaintypes table is not specified it may be a variable name from the variables table. A –names table consists of two columns, both of which must be included:

- **value**

This column is *mandatory* and must be an *integer* beginning with "0" and continuing sequentially up to levels–1 (maximum 57). It is the integer equivalent of the name to be defined in the next column.

- **name**

The column is *mandatory* and must be of the *integer* or *alpha* type. It specifies the input and output name of the value being defined.

Below is a typical example of use of the –names tables. All variables that are of "boolean" domain type may take values "yes" or "no". The variable "Processor" may take any of the values "Z80", "M6502" or "I8085". Note that the variable "Floppies" does not need to have its –names table, as the default integer values 0,1,2,3 are acceptable.

Example:

boolean–**names**

value	**name**
0	yes
1	no

Processor–**names**

value	**name**
0	Z80
1	M6502
2	I8085

Op_system–**names**

value	**name**
0	cpm
1	other

4.5.7. The –structure tables

The –structure table is *optional* and is used to define a structured domain for any variable that is of structured type (as specified in the domaintypes or variables table). A structured domain has the form of hierarchical graph, where the lowest level corresponds to the values of the variable as they appear in the input examples (and are possibly defined in the corresponding –names table). Higher levels (as defined in the –structure table) specify parent nodes in the hierarchy of values and are used to simplify classification rules.

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

- **name**

  Optional name of the corresponding value, must be of the *integer* or *alpha* type. If specified, it will appear in classification rules instead of the value.

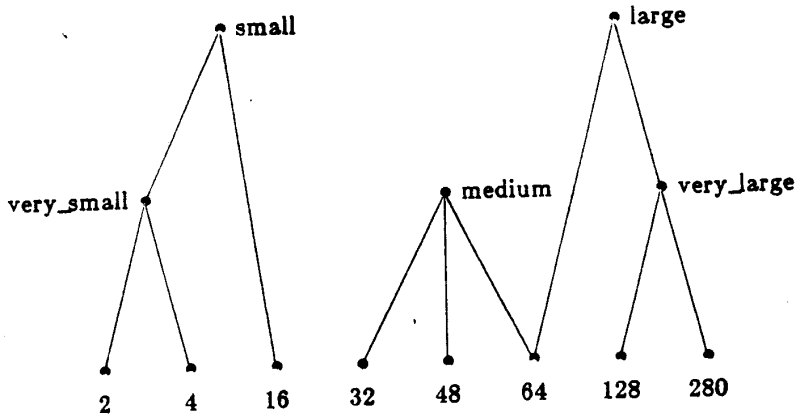
- **value**

  Mandatory column, specifying a node in the hierarchy which is a parent of the nodes specified in the "subvalues" column. May be any *integer*, as its role is only to discriminate between different nodes in the hierarchy and to be possibly referred to later in the "subvalues" column of some other node. It must be always greater than any of the following "subvalues".

- **subvalues**

  This column is *mandatory* and specifies a set of children values for the parent node as defined in the previous column. It consists of a *string of integers* separated by commas or by "..", as in extended selectors. 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 (e.g. "64") may be a children of different parents ("medium", "large").



For the variable "Memory" we must first define its –names table, because it may not take all values between 2 and 280, as this would exceed the maximum number of levels allowed (58). The –structure table then defines the hierarchy of values. Note that the node "large" must be defined after the node "very_large" as it is higher in the hierarchy.

Example:

Memory–**names**

value	**name**
0	2
1	4
2	16
3	32
4	48
5	64
6	128
7	280

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

4.5.8. The arules tables and lrules tables

The arules tables and lrules tables are *optional* and are used to input A–rules and L–rules to *AQ15* for *constructive learning*. when applying A–rules or L–rules to the input data, those variables in the A–rules or L–rules not present in the variables table are introduced as new variables. The new variables are arithmetic expressions as references in the A–rules or L–rules. Moreover, *AQ15* selects some new variables by a certain criteria and *constructs* corresponding new selectors for the events and hypotheses. The constructive learning may specialize the descriptions of input data and simplify the resulting hypotheses since the new variables selected

may be better than original ones for the star generation.

• #

Mandatory symbol associating an optional number with each rule. The number is *optional* and followed by the body of an A–rule or L–rule.

• **aexpr** or **lexpr**

Mandatory column giving an A–rule or a L–rule. The body of an A–rule is an arithmetic expression, and the body of a L–rule is a VL1 expression, whose consequence is an extended selector for a new variable and whose condition is a disjunction of complexes (see format for cpx in next section).

• **if**

Mandatory word following the body of a rule and associating an optional number with a condition of the rule if the rule has one. The condition of a rule is a disjunction of complexes.

• **v**

Mandatory symbol linking two adjacent disjunctive conditions.

Below are examples of arules tables and lrules tables.

Examples:

```
arules
  #  aexpr
  1  Tot_Memory := 512*Floppies + Memory  if
       [Disk=no]

lrules
  #  lexpr
  1  [Main_device = no]  if
       [Disk&Printer=no]

  2  [Device_storage = yes]  if
       [Floppies = 1 V 2]  v
       [Disk = yes]

  3  [Computer_cost = medium]  if
       [Main_device = no][Device_storage = yes]
```

4.5.9. The –inhypo tables

The –inhypo tables are *optional* and are used to input rules to *AQ15* for incremental learning. The specific name of this table must be the name of a decision class. If the name given for an –inhypo table has not been seen before (i.e. in a –events table, see 4.5.11), the associated class is assumed to be at the top of a structured rule base (see 4.5.12).

The rules input in –inhypo tables may have two roles. In the first case, when there is at least one –events table specified, the input rules are used as initial covers for *incremental learning.* If incremental learning is not desired, then this table may be excluded. In the second case, when there is no –events table, the input rules are treated as *events.* This means that produced decision rules will cover a whole input rule from the corresponding class. However, each input rule typically comprises a set of events which may be in different classes, and some rules in different classes may comprise the same events. In that case, when input rules intersect, their intersection is treated according to the "ambig" parameter (see 4.2).

- #

Mandatory column associating a number with each complex in the rule. It is an *integer*, the first row must be numbered "1" and complexes must be numbered sequentially. *Only* in –inhypo tables may a single relational tuple span more than one line. However, there must be only one # entry for each complex in the table.

- **cpx**

Mandatory column giving a VL_1 (variable–valued logic) declaration of the complex. A complex is presented as a series of extended selectors. Each extended selector is an expression of the form [variable$_i$ & variable$_j$ & ... & variable$_k$ # values]. The symbol "&" in a selector means *internal conjunction.* Thus, the extended selector above is equivalent to conjunction of extended selectors (complex) [variable$_i$ # values][variable$_j$ # values]...[variable$_k$ # values]. Selectors may be separated by any amount of white space or new lines. A new complex is started only when a new entry for the # column (i.e. a number) is found. The brackets are mandatory. The variable may be any variable declared in the variables table, but the same variable may not appear twice in one complex. The values must be defined values (e.g. in the –names tables) for the variable given on the right of the "#" relation. The *relation* # is one of the following relations: $<$, $<=$, $=$,

<>, >=, and >. Several values , separated by "," or "v", may be specified in one extended selector in any one or arbitrary mixtures of the following forms:

value

$value_1..value_2$

The symbol "," and "v" in an extended selector means "or", and the symbols ".." indicate a range of acceptable (predefined) values. So, the extended selector [width & height >= 5] is read "both the width and height are greater than or equal to 5". The extended selector [temperature <> 20..25, 50..60] is read "the temperature is neither between 20 and 25 nor between 50 and 60. In other words, if the range of temperature is between 0 and 80, then the extended selector says that "temperature is between 0 and 19, 26 and 49, or 61 and 80".

Below is an example of –inhypo tables which use the variables defined in the previous sections. These rules are results of application of *AQ15* to the introductory example. They will be used as initial hypotheses for incremental learning on a new set of examples. Recall that concatenation of extended selectors is used to express a conjunction, and that different complexes in a cover represent a disjunction.

Example:

Under1000–inhypo

#	**cpx**
1	[Floppies=0]

From1000to4000–inhypo

#	**cpx**
1	[Floppies=1..2 Disk&Printer=no]

Over4000–inhypo

#	**cpx**
1	[Disk=yes]
2	[Printer=yes]

4.5.10. The –inghypo tables

Their roles are the same as –inhypo tables except (see 4.5.9.) for some extension as follows.

- **#**

Similar to that in section 4.5.8.

- **tcpx**

Similar to that in section 4.5.8., except that in a selector, $[term_1\&...\&term_k\#values]$, for some i, $term_i$ may be a variable or an expression involving a variable that was not specified in the domaintypes tables or the variables tables. Thus, *AQ15* can introduce some new variables which stand for expressions or specifies variables undefined in the previous sections.

In the following example, the first and third tables are –inghypo tables which use the new variables undefined in the previous sections, and the second is an –inhypo table.

Example:

Under1000–**inghypo**

#	tcpx
1	[512*Floppies+Memory<=32]
	{Events covered: 1..4}

From1000to4000–**inhypo**

#	cpx
1	[Floppies=1..2] [Disk&Printer=no]
	{Events covered: 1..5}

ver4000–**inghypo**

#	tcpx
1	[Floppies*Memory=128]
	{Events covered: 1..3}

4.5.11. The –events tables

These tables are used to input events to *AQ15* and are *optional*. If there is no –events table, *AQ15* will treat the input hypotheses as events. The specific name given to an –events table

corresponds to a single decision class. In *AQ15*, if the specific table name has not been seen previously (in an –inhypo table), then a new class is created at the top of the rule base structure (see next section).

The column headers for this table type consist of variable names which were defined in the variables table (see 4.5). The values in the rows of the table must be legal values for the appropriate variables. Since many attributes may be needed to describe an event, it is possible to split an –events table into several tables. This is done by repeating the table name (both specific and general), and using different column headings in each occurrence. When splitting, column headings may not overlap, and each table must have the same number of events. The table must have at least two columns:

- **#**

  Optional row number. It is an *integer*, the first row must be numbered "1" and rows must be numbered sequentially.

- **variable(s)**

  Arbitrary number of columns, each for a variable declared in the variables table. Recall that the maximum number of columns allowed is 40, and the maximum number of variables is currently set to 60. The entries in the rows of the table must take legal values of the corresponding variables in the heading. They may consist of single values or strings of values, separated by commas or ".." as in extended selectors. The default value for a variable, if its column is missing or a placeholder $?*$ is entered, is its entire range, i.e. all legal values for the variable (0..levels–1).

The –events tables shown below use the attributes defined in the variables and the –names tables from examples above. Some new examples are added to the initial set from section 2. The first event in the first table represents three examples from the initial set, as the variable "Memory" may take values of 2, 4 or 16. In the last event a few placeholders '*' appear, which means that the corresponding variables may take all possible values (in this case "yes" and "no"). Note that in the first –events table for the class "Under1000" there is no row enumeration.

The events for the class "Over4000" have been split into two –events tables with the same specific name. This can be helpful when dealing with tables with too many columns.

Example:

Under1000–events

Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer
no	no	no	other	0	no	M6502	2..16	no
no	no	no	other	0	no	Z80	32	no
yes	yes	no	other	0	no	Z80	16,48	no
yes	yes	yes	other	0	no	M6502	16	no
*	*	*	other	2	no	M6502	4	no

Over4000–events

#	Pascal	Fortran	Cobol	Op_system
1	yes	yes	yes	other
2	no	no	yes	other
3	yes	yes	yes	cpm
4	no	yes	yes	cpm
5	yes	yes	yes	other
6	no	yes	yes	other
7	yes	yes	no	other
8	no	yes	yes	other
9	no	no	no	other
10	no	no	yes	cpm
11	yes	yes	yes	cpm
12	no	yes	yes	cpm
13	yes	yes	no	cpm

Over4000–events

#	Floppies	Disk	Processor	Memory	Printer
1	1	yes	Z80	128	no
2	2	no	I8085	64	yes
3	1	yes	Z80	280	no
4	2	no	I8085	64	no
5	2	no	I8085	64	yes
6	2	no	Z80	64	yes
7	2	no	Z80	48	yes
8	2	yes	Z80	64	no
9	2	no	I8085	32	yes
10	0	yes	Z80	64	no
11	2	no	I8085	64	yes
12	2	no	I8085	64	yes
13	0	yes	Z80	64	no

4.5.12. The -children tables

AQ15 accepts *optional* -children tables in order to define a structuring of a rule base. The specific name of the table must be the name of a class already defined, i.e., the name must have appeared as the name of an -events table. The rule base may be structured to arbitrary depth.

The -children table consists of two columns:

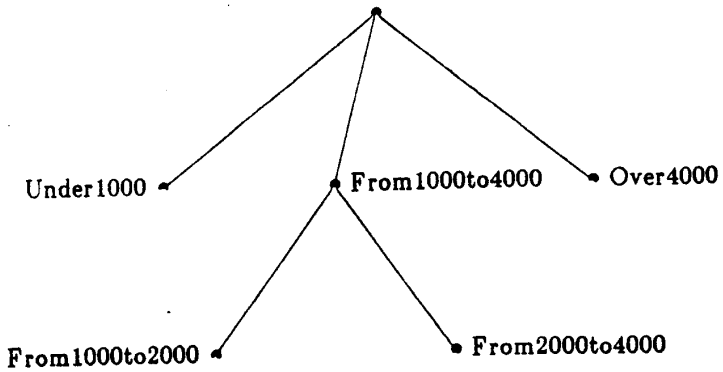
- **node**

  This column is *mandatory* and must be of the *alpha* type. It specifies the name of the node to be defined.

- **events**

  Mandatory column that specifies which events attached to the parent class also belong to this child node. It consists of a *string of integers* separated by commas or by "..", as in extended selectors. These numbers correspond to events associated with the parent. The parent's events are numbered in the order they appear in the -events table. Classes more than one level deep in the rule base use the event numbers associated with their ancestor at the *top* of the structure. This allows the user to specify all events with the same set of numbers.

The tree below shows how a sample rule base might be structured. In this case classes "Under1000", "From1000to4000", and "Over4000" are brothers at the top of the structure. The class "From1000to4000" has two sub-classes, "From1000to2000" and "From2000to4000".



The **–events** table below defines 13 events of the class "From1000to4000". The –children table assigns the first 6 events of these and the 11th event to class "From1000to2000" and the last 7 events to the class "From2000to4000". Note that the 11th event belongs to both sub-classes, and will be treated as specified by the "ambig" parameter in the parameters table (see 4.2).

Example:

From1000to4000–**events**

#	Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer
1	no	no	no	other	1	no	M6502	16	no
2	no	yes	yes	cpm	2	no	I8085	32	no
3	yes	yes	yes	cpm	1	no	Z80	64	no
4	no	yes	yes	cpm	2	no	Z80	16	no
5	no	yes	yes	cpm	1	no	I8085	48	no
6	yes	yes	yes	cpm	2	no	Z80	32	no
7	no	yes	yes	cpm	2	no	Z80	64	no
8	yes	yes	yes	cpm	2	no	Z80	48	yes
9	no	no	yes	cpm	2	no	Z80	64	no
10	no	yes	yes	other	2	no	Z80	64	no
11	no	yes	no	other	2	no	M6502	48	no
12	no	no	yes	other	3	no	Z80	64	no
13	yes	yes	yes	other	2	no	Z80	64	no

From1000to4000–**children**

node	**events**
From1000to2000	1..6,11
From2000to4000	7..13

4.5.13. The –tevents tables

Similar to the –events tables except that events are used only for testing.

4.6. How to Run *AQ15*

This section gives a detailed explanation of the sample output of the *AQ15* program. The sample input was presented earlier, in the tables guide. If the input is stored in the file "sample.inp", then executing:

aq15.run <sample.inp >sample.out

under the UNIX shell would cause *AQ15* to run the example and save the result in the file "sample.out". This command also assumes that the user has stored the executable version of the *AQ15* program under the name "aq15.run", or that he has defined an appropriate alias.

4.6.1. Sample Output

In the output file we find a set of relational tables in the same format as required for input. This allows the *AQ15* output to be directly used as an input for some other programs which are handled as operators by the QUIN relational database system. In the output file we find all tables that were requested by the "echo" parameter (see 4.5.2) for each run of *AQ15* plus tables that contain produced classification rules.

parameters

run	mode	ambig	trim	wts	maxstar	echo	criteria
1	ic	neg	mini	cpx	10	pcib	default
2	ic	pos	gen	all	10	pcvs	mincost

The program augments the parameters table from the input with default parameters that were omitted. It defines the way in which *AQ15* is to run on the input data. The first run of *AQ15* uses the intersecting covers mode, treats ambiguous examples as negative examples, produces the minimal descriptions, associates weights with each complex, uses a maxstar of 10, default criteria, and echoes the parameters, –criteria, –inhypo, and –children tables. In most cases, if a maxstar is approximately equal to the number of variables used, the results are optimal in terms of quality of classification rules produced and computational resources required.

default–criteria

#	criterion	tolerance
1	maxnew	0.00
2	minsel	0.00

The default–criteria table specifies a LEF of two criteria: first maximize the number of new events covered, and second (in the case of ties) minimize the number of variables used in each complex. Both criterions have the tolerance of 0%. The costs of variables are not taken into account when this criteria table is used. In the majority of cases the default criteria cause *AQ15* to produce the most compact classification rules.

Under1000–inhypo

#	cpx
1	[Floppies=0]

This is an initial rule for the class "Under1000". *AQ15* modifies it when contradictory new examples are encountered and may produce additional complexes to cover all positive events of the class.

From1000to4000–inhypo

#	cpx
1	[Floppies=1..2] [Disk&Printer=no]

This initial rule for the class "From1000to4000" is read: "Our initial hypothesis states that computer costs between 1000 and 4000 if it has 1 to 2 Floppies, and it has no Disk, and it has no Printer". Note that conjunction of extended selectors is expressed by their concatenation.

Over4000–inhypo

#	cpx
1	[Disk=yes]
2	[Printer=yes]

This is an initial rule for the class "Over4000". Note that disjunction is expressed with different lines, where each line is a complex, i.e., a conjunction of extended selectors (in general).

From1000to4000–children

node	events
From1000to2000	1,2,3,4,5,6,11
From2000to4000	7,8,9,10,11,12,13

The –children table introduces two new classes, sub–classes of the "From1000to4000" class. As there are no input hypotheses for these two classes, *AQ15* starts with an empty assumption, i.e., that the initial hypothesis for each of these two classes is unconditionally true. The assumption is specialized when negative examples are encountered.

This ends the echo of some of the input tables for the first run. Next, the **–outhypo** tables follow, which contain produced classification rules for the input examples.

Under1000–outhypo

#	cpx
1	[Floppies=0] [Memory=small,32,48] *(total:4, unique:4)*
2	[Memory=2..4] *(total:1, unique:1)*

This is the classification rule for the class "Under1000". The first complex was produced from the initial hypothesis, which was specialized to exclude negative events (e.g. events 10 and 13 from the class "Over4000"). It covers totally 4 events (1,2,3,4) and these events are not covered by any other complex. The second complex uniquely covers one remaining event (5). The weights at the end of each complex summarize these results.

From1000to4000–outhypo

#	cpx
1	[Floppies=1..2] [Disk&Printer=no] [Processor=Z80,M6502] [Memory=medium,16] *(total:9, unique:8)*
2	[Op_system=cpm] [Memory=32,48] *(total:4, unique:3)*
3	[Floppies=3] *(total:1, unique:1)*

This is the classification rule for the class "From1000to4000". Again, the first complex is the specialized initial hypothesis, while the other two complexes cover the remaining positive events. In this case, the 6th event in the class is covered by the first and the second complex. Therefore, the number of uniquely covered events is one less than the number of totally covered events.

Over4000-outhypo

#	cpx
1	[Memory=32,64] [Printer=yes]
	(total:6, unique:1)
2	[Disk=yes]
	(total:5, unique:5)
3	[Processor=I8085] [Memory=64]
	(total:5, unique:1)
4	[Cobol=no] [Printer=yes]
	(total:2, unique:1)

This is the classification rule for the class "Over4000". The first complex in the initial hypothesis was not modified, but after sorting of complexes it became the second one. All events that it covers are uniquely covered. The first and last complexes resulted from the specialization of the second complex in the initial hypothesis, and the third complex is produced to cover the remaining uncovered positive events.

From1000to2000-outhypo

#	cpx
1	[Memory=16,32]
	(total:4, unique:3)
2	[Floppies=1]
	(total:3, unique:2)

This is the classification rule for the sub-class "From1000to2000". Note that it does not cover the 11th event, which also belongs to the sub-class "From2000to4000" and was treated as a negative example.

From2000to4000-outhypo

#	cpx
1	[Cobol=yes] [Floppies=2..3] [Memory=48,64]
	(total:6, unique:6)

This is the classification rule for the sub-class "From2000to4000". As there were no initial hypothesis for these two classes, their descriptions are considerably simpler than the description for their parent class "From1000to4000".

This run used (milliseconds of CPU time):

System time:	200
User time:	10783

Time taken to form the rules for the first run. This does not include input and output time.

Now the second run starts on the same input data.

parameters

run	mode	ambig	trim	wts	maxstar	echo	criteria
1	ic	neg	mini	cpx	10	pcib	default
2	ic	pos	gen	all	10	pcvs	mincost

This is the echoed parameters table for the second run of *AQ15*. There are some differences in parameters that cause *AQ15* to produce different covers: the mincost–criteria is used instead of the default, covers are as general as possible, weights are associated with each extended selector in complexes, and ambiguous examples are treated as positive examples. Besides the parameters table, the mincost–criteria, variables, and –structure tables are echoed.

mincost–criteria

#	criterion	tolerance
1	mincost	0.20
2	maxtot	0.00

This criteria table specifies a LEF that first minimizes the total cost of the variables used, and then (in case of ties) maximizes the total number of positive events covered. The first criterion has a tolerance of 20%.

variables

#	type	levels	cost	name
1	nom	2	10.00	Pascal.boolean
2	nom	2	10.00	Fortran.boolean
3	nom	2	10.00	Cobol.boolean
4	nom	2	10.00	Op_system
5	lin	4	100.00	Floppies
6	nom	2	0.00	Disk.boolean
7	nom	3	0.00	Processor
8	str	8	100.00	Memory
9	nom	2	0.00	Printer.boolean

In the output variables table the columns "type" and "levels" are deduced from the input domaintypes table. The cost associated with each variable is taken into account only when the criterion "mincost" is used. With these costs we intended to discourage the use of the variables "Floppies" and "Memory", and to encourage the use of the variables "Disk", "Processor" and "Printer" in the classification rules. By changing the costs the user can influence the terms by which the results are described.

Memory-structure

name	value	subvalues
very_small	8	0,1
small	9	0,1,2
medium	10	3,4,5
very_large	11	6,7
large	12	5,6,7

This is the echoed -structure table for the variable "Memory". Note that in the subvalues column, all nodes higher in the hierarchy were substituted by their corresponding leaves.

Under1000-outhypo

#	cpx
1	[Disk=no]*(pos:5, neg:21)* [Floppies=0]*(pos:4, neg:2)* *(total:4, unique:4)*
2	[Memory=very_small]*(pos:1, neg:0)* *(total:1, unique:1)*

This is the rule for the class "Under1000". Note that in the first complex the expensive variable "Memory" from the first run was replaced with the cheaper variable "Disk". This extended selector alone covers 5 events from this class and 21 from the other two classes. However, all these negative events are eliminated by the second extended selector. In the second complex, the leaves values for the variable "Memory" (2 or 4) were replaced by the value as high in the hierarchy as possible (very_small).

From1000to4000-outhypo

#	cpx
1	[Disk=no]*(pos:13, neg:13)* [Printer=no]*(pos:12, neg:11)* [Cobol=yes]*(pos:11, neg:11)* [Processor=Z80]*(pos:9, neg:9)* *(total:8, unique:3)*
2	[Disk=no]*(pos:13, neg:13)* [Printer=no]*(pos:12, neg:11)* [Floppies=1..2]*(pos:12, neg:12* [Memory=very_large,2,16,32,48]*(pos:7, neg:7)* *(total:6, unique:4)*
3	[Disk=no]*(pos:13, neg:13)* [Processor=Z80,M6502]*(pos:11, neg:12)* [Op_system=cpm]*(pos:8, neg:6)* *(total:6, unique:1)*

This is the rule for the class "From1000to4000". Note that extended selectors are ordered within each complex in decreasing order of the number of positive events covered.

Over4000-outhypo

#	cpx
1	[Disk=yes]*(pos:5, neg:0)* *(total:5, unique:5)*
2	[Printer=yes]*(pos:7, neg:1)* [Op_system=other]*(pos:7, neg:10)* *(total:5, unique:3)*
3	[Memory=small,large]*(pos:11, neg:11)* [Processor=I8085]*(pos:6, neg:2)* *(total:5, unique:3)*

This is the rule for the class "Over4000". In all events that are covered by the third complex, the variable "Memory" has value 64 but because the classification rule is to be as general as possible (due to the setting of the parameter "trim"), the program climbs the generalization tree as high as it can without covering any negative examples.

From1000to2000-outhypo

#	cpx
1	[Printer=no]*(pos:7, neg:6)* [Memory=small,very_large,32,48]*(pos:6, neg:2)* *(total:6, unique:5)*
2	[Printer=no]*(pos:7, neg:6)* [Op_system=cpm]*(pos:5, neg:3)* [Pascal=yes]*(pos:2, neg:2)* *(total:2, unique:1)*

This is the rule for the sub-class "From1000to2000". Note that the 11th event is now covered by the classification rules for both sub-classes, due to the parameter setting of "ambig".

From2000to4000–outhypo

#	cpx
1	[Processor=Z80,M6502]*(pos:7, neg:5)* [Memory=very_small,medium,large]*(pos:7, neg:* [Pascal=no]*(pos:5, neg:5)* *(total:5, unique:3)*
2	[Processor=Z80,I8085]*(pos:6, neg:5)* [Op_system=other]*(pos:4, neg:2)* *(total:3, unique:1)*
3	[Printer=yes]*(pos:1, neg:0)* *(total:1, unique:1)*

This is the rule for the sub-class "From2000to4000". The differences of the descriptions for these two sub-classes between the first and the second run are much greater than for other classes, as there was no initial hypothesis. The expensive variables "Floppies" and "Memory" are avoided as much as possible, which results in more complicated descriptions.

4.6.2. Pre–Processing and Sample Output

AQ15 does *constructive learning* by applying A–rules or L–rules to the events read or input hypotheses. The constructive learning is as follows. In the input stage, *AQ15* picks up all expressions in the extended selectors read in and the variables that were not specified neither in the domaintypes tables nor in the variables tables, puts them into the list of variables, counts the levels for them, and constructs new selectors corresponding to the new variables. *AQ15* starts to apply the A–rules or L–rules to the events and input hypotheses as soon as input processing finishes. If there is a condition of the current rule, the program tests first if the condition is satisfied by the event or hypothesis currently treated. If satisfied, it adds the value for a new variable corresponding to the expression of A–rule or the consequence of L–rule to the event or hypothesis. If the condition is not satisfied, it adds a default value named "*others*" to the event or hypothesis. When the processing is done for all rules and all events or hypotheses, a filter is invoked to select a few best new variables according to a criterion described below, and add them to the original variables. The processing is detailed as follows. For each new variable, and for the corresponding attributes in the positive events, the filter first calculates the average number of disappearances of the attributes in the negative events. It then sorts the numbers and selects those variables to be used further that have the largest numbers. This utility can make the new variables chosen better than original ones in the sense of producing better rules. Now, *AQ15* starts generating output hypotheses.

The following two examples show that the pre–processing is a powerful tool for inductive learning. The first example is for applying an A–rule to the example in section 4.3, viz., introducing a new variable Tot_Memory into the events. The second example is for applying L–rules, viz., introducing three new variables, Main_device, Device_storage and Computer_cost. Both results are better than original ones in section 4.6.1.

Example: (for A–rules)

Note that in the variables table, variable Tot_Memory is a newly added one.

arules

#	aexpr
1	Tot_Memory := 512 * Floppies + Memory if [Disk=no]

parameters

run	mode	ambig	trim	wts	maxstar	echo	criteria
1	ic	pos	mini	cpx	10	pdvns	default

domaintypes

type	levels	cost	name
nom	2	1.00	boolean
lin	4	1.00	Floppies
nom	2	1.00	Op_system
nom	3	1.00	Processor
str	8	1.00	Memory
nom	11	1.00	Tot_Memory

variables

#	type	levels	cost	name
1	nom	2	1.00	Pascal.boolean
2	nom	2	1.00	Fortran.boolean
3	nom	2	1.00	Cobol.boolean
4	nom	2	1.00	Op_system
5	lin	4	1.00	Floppies
6	nom	2	1.00	Disk.boolean
7	nom	3	1.00	Processor
8	str	8	1.00	Memory
9	nom	2	1.00	Printer.boolean
10	nom	1	1.00	Tot_Memory

boolean–names

value	name
0	yes
1	no

Op_system–names

value	name
0	cpm
1	other

Processor–names

value	name

0	Z80
1	M6502
2	I8085

Memory–names

value	name
0	2
1	4
2	16
3	32
4	48
5	64
6	128
7	280

Tot_Memory–names

value	name
0	2
1	0
2	1
3	3
4	514
5	1027
6	517
7	1026
8	516
9	1029
10	others

Memory–structure

name	value	subvalues
2_4	0	0,1

Under1000–events

#	Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer	Tot_Memory
1	no	no	no	other	0	no	M6502	6	no	2
2	no	no	no	other	0	no	M6502	2	no	0
3	no	yes	no	other	0	no	M6502	4	no	1
4	no	yes	no	other	0	no	Z80	32	no	3

From1000to4000–events

#	Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer	Tot_Memory
1	no	no	no	other	1	no	M6502	16	no	514
2	no	yes	yes	cpm	2	no	I8085	2	no	1027
3	yes	yes	yes	cpm	1	no	Z80	64	no	517
4	no	yes	yes	cpm	2	no	Z80	16	no	1026
5	no	yes	yes	cpm	1	no	I8085	48	no	516

Over4000–events

#	Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer	Tot_Memory
1	yes	yes	yes	other	1	yes	Z80	128	no	others
2	no	no	yes	other	2	no	I8085	64	yes	1029
3	yes	yes	yes	cpm	1	yes	Z80	280	no	others

Under1000–outhypo
 # cpx
 1 [Tot_Memory = 0..3]
  (total:4, unique:4)

From1000to4000–outhypo
 # cpx
 1 [Tot_Memory = 514 v 516 v 517 v 1026 v 1027]
  (total:5, unique:5)

Over4000–outhypo
 # cpx
 1 [Tot_Memory >= 1029]
  (total:3, unique:3)

This run used (milliseconds of CPU time):
System time: 433
User time : 21833

Example: (for L–rules)

Note that in the variables table, variables 10–12 are newly added.

lrules

#	lexpr
1	[Computer_cost=medium] if
	[Main_device=no][Device_storage=yes]
2	[Device_storage=yes] if
1	[Disk=yes] v
2	[Floppies = 1,2]
3	[Main_device=no] if
	[Disk&Printer=no]

parameters

run	mode	ambig	trim	wts	maxstar	echo	criteria
1	ic	pos	mini	cpx	10	pdvns	default

domaintypes

type	levels	cost	name
nom	2	1.00	boolean
lin	4	1.00	Floppies
nom	2	1.00	Op_system
nom	3	1.00	Processor
str	8	1.00	Memory
nom	2	1.00	Main_device
nom	2	1.00	Device_storage
nom	2	1.00	Computer_cost

variables

#	type	levels	cost	name
1	nom	2	1.00	Pascal.boolean
2	nom	2	1.00	Fortran.boolean
3	nom	2	1.00	Cobol.boolean
4	nom	2	1.00	Op_system
5	lin	4	1.00	Floppies
6	nom	2	1.00	Disk.boolean
7	nom	3	1.00	Processor
8	str	8	1.00	Memory
9	nom	2	1.00	Printer.boolean
10	nom	2	1.00	Main_device
11	nom	2	1.00	Device_storage
12	nom	2	1.00	Computer_cost

boolean–names

value	name
0	yes
1	no

Op_system–names

value	name
0	cpm
1	other

Processor–names

value	name
0	Z80
1	M6502
2	I8085

Memory–names

value	name
0	2
1	4
2	16
3	32
4	48

5	64
6	128
7	280

Main_device–names

value	names
0	no
1	others

Device_storage

value	names
0	yes
1	others

Computer_cost–names

value	names
0	medium
1	others

Memory–structure

name	value	subvalues
2_4	0	0,1

Under1000–events

#	Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer	Main_device	Device_storage	Computer_cost
1	no	no	no	other	0	no	M6502	6	no	no	others	others
2	no	no	no	other	0	no	M6502	2	no	no	others	others
3	no	yes	no	other	0	no	M6502	4	no	no	others	others
4	no	yes	no	other	0	no	Z80	32	no	no	others	others

From1000to4000–events

#	Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer	Main_device	Device_storage	Computer_cost
1	no	no	no	other	1	no	M6502	16	no	no	yes	medium
2	no	yes	yes	cpm	2	no	I8085	2	no	no	yes	medium
3	yes	yes	yes	cpm	1	no	Z80	64	no	no	yes	medium
4	no	yes	yes	cpm	2	no	Z80	16	no	no	yes	medium
5	no	yes	yes	cpm	1	no	I8085	48	no	no	yes	medium

Over4000–events

#	Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer	Main_device	Device_storage	Computer_cost
1	yes	yes	yes	other	1	yes	Z80	128	no	others	yes	others
2	no	no	yes	other	2	no	I8085	64	yes	others	yes	others
3	yes	yes	yes	cpm	1	yes	Z80	280	no	others	yes	others

Under1000–outhypo
 # cpx
 1 [Device_storage=others]
   (total:4, unique:4)

From1000to4000–outhypo
 # cpx
 1 [Computer_cost=medium]
   (total:5, unique:5)

Over4000–outhypo
 # cpx
 1 [Main_device=others]
   (total:3, unique:3)

 2 [Device_storage=yes] if
 1 [Disk=yes] v
 2 [Floppies = 1..2]

 3 [Main_device=no] if
   [Disk&Printer=no]

Recall that "others" is the default value for a variable, that is, a complement of the set of values really read in. For example, [Main_device=others] is equivalent to ¬[[Main_device=no], i.e., [Disk=yes] v [Printer=yes], which is the same as results in section 4.6.1.

This run used (milliseconds of CPU time):
 System time: 267
 User time : 7767

4.6.3. Incremental Learning and Sample Output

AQ15 also has the capability of *incremental learning* by using input or previously generated hypotheses (see procedure formrule in the Appendix). The incremental learning works as follows. Suppose the classes of the training examples and the input hypotheses, if any, are given.

- **Step 1**: Partition of Training Examples

AQ15 first partitions each class of input training examples into several subsets according to a percentage of total partition given by the user. The partitioned examples are stored into temporary nodes, while working nodes initially are left empty.

- **Step 2**: Specialization of Hypotheses

If all temporary nodes become empty, then the program is terminated, otherwise the subset of partitioned examples on the top of each temporary node is transferred to the corresponding working node and will be used as new training examples later.

If a new added example is covered by hypotheses, its name is stored into the corresponding hypotheses. Now, *AQ15* specializes the hypotheses in the way that if a hypothesis covers positive examples and negative examples as well, then *AQ15* generates a cover which covers the positive examples against the negative ones. A filter then chooses those complexes in the cover that cover more than 50% of the examples for that cover and deletes the rest of the complexes and the examples covered by the chosen complexes. Now, *AQ15* makes an intersection of the chosen complexes and original hypothesis and uses the resulting complexes as training examples for the next run. It is obvious that the specialization of hypotheses can speed up the inductive learning by using hypotheses each of which may cover more examples rather than just the current examples themselves.

- **Step 3:** Generation of New Partial Covers

$AQ15$ generates new covers which cover the chosen hypotheses and remaining positive examples in Step 2 against negative ones.

- **Step 4:** Generation of Partial Cover

The program uses the output hypotheses generated in this run as the input hypotheses for the next run, and returns to Step 2.

The following is an example of incremental learning.

Example:

Suppose we are given the same input data as that in section 4.6.2. for A-rules. Suppose we are also given a parameters table as follows:

parameters

run	mode	ambig	trim	wts	maxstar	echo	criteria	increment
1	ic	pos	mini	cpx	10	piw	default	50

where increment = 50 indicates that $AQ15$ will *randomly* partition the input data into subsets with 50% of the input data.

$AQ15$ uses the following first 50% of the events as training examples for the first run:

Under1000–events

Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer
no	no	no	other	0	no	Z80	32	no
no	yes	no	other	0	no	M6502	4	no

From1000to4000–events

Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer
no	yes	yes	cpm	1	no	I8085	48	no
yes	yes	yes	cpm	1	no	Z80	64	no
no	yes	yes	cpm	2	no	I8085	32	no

Over4000–events

Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer
yes	yes	yes	cpm	1	yes	Z80	280	no
yes	yes	yes	other	1	yes	Z80	128	no

AQ15 runs on the data above and generates a partial cover:

NO. 1 (Incremental Learning)

Under1000–outhypo

cpx

1 [Fortran = no]

{Events covered: 1, 2}

(Total:2, Unique:2)

From1000to4000–outhypo

cpx

1 [Floppies = 1 v 2][Memory = 32 v 48 v 64]

{Events covered: 1..3}

(Total:3, Unique:3)

Over4000–outhypo

cpx

1 [Disk = yes]

{Events covered: 1, 2}

(Total:2, Unique:2)

$AQ15$ then uses the output hypotheses as the input hypotheses, adds the other 50% training events to the corresponding classes, runs again, and generates the following result:

NO. 2 (Incremental Learning)

Under1000–outhypo

cpx

1 [Fortran = no][Floppies = 0]

{Events covered: 1..4}

(Total:4, Unique:4)

From1000to4000–outhypo

cpx

1 [Floppies = 1 v 2][Disk = no][Processor = Z80 v M6502]

{Events covered: 2, 4, 5}

(Total:3, Unique:2)

2 [Fortran = yes][Floppies = 1 v 2][Memory = 32 v 48 v 64]

{Events covered: 1..3}

(Total:3, Unique:2)

Over4000–outhypo

cpx

1 [Disk = yes]

{Events covered: 1, 3} (new)

(Total:2, Unique:2)

2 [Fortran = no][Floppies = 2]

{Events covered: 2}

(Total:1, Unique:1)

4.6.4. Testing and Sample Output

A testing facility which recognizes the concept *membership of examples* was incorporated into $AQ15$. There are two matching measures, *strict matching measure* and *analogical matching measure*. In the strict matching measure, $AQ15$ tests if a testing example is covered by output hypotheses or input hypotheses which depends on the existence of training examples. In the analogical matching measure, $AQ15$ determines the degree of similarity or closeness between the testing example and the hypotheses. If 'w' occurs in the echo column, i.e., reduction criterion 1 applies, $AQ15$ uses the measure described in (Michalski, Mozetic, Hong &Lavrac 86). If 'm' occurs in the echo column, i.e., reduction criterion 2 applies, $AQ15$ uses the measure introduced in (Michalski &Chilausky 80).

Example:

Suppose we divide the set of training examples in section 4.6.2 into two subsets. One of them is the same as the first run of the previous example and will be used as the set of training examples. The other, as shown below, as the set of testing examples.

Under1000–events

Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer
no	no	no	other	0	no	M6502	16	no
no	no	no	other	0	no	M6502	2	no

From1000to4000–events

Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer
no	yes	yes	cpm	2	no	Z80	16	no
no	no	no	other	1	no	M6502	16	no

Over4000–events

Pascal	Fortran	Cobol	Op_system	Floppies	Disk	Processor	Memory	Printer
no	no	yes	other	2	no	I8085	64	yes

AQ15 first runs on the training examples and gives output hypotheses the same as those in the first run (NO. 1) of the previous example. The program then tests each testing example and gives its matching degree as shown in the resulting *Confusion Matrix*. We explain the meaning of the confusion matrix by examples. In the matrix, testing event 1 in class Under1000 has the maximum matching degree (0.29) in its own class, so the match is correct; event 1 in class From1000to 4000 has a correct match although it is covered by none of the output hypotheses (indicated by *member=no*); and the testing event in class Over4000 has incorrect match since it

has the maximum degree (0.43) in class From1000to4000r. Also, there are 3 events which have correct match and 2 incorrect, so the matching accuracy is 60%.

Confusion Matrix (Reduction 1)

# Event	Class	member	Under1000	From1000to4000	Over4000
1			0.29	0.0	0.00
2			0.29	0.00	0.00
From1000to4000:					
1		no	0.14	0.16	0.14
2	Under1000		0.29	0.00	0.00
Over4000:					
1	From1000to4000		0.29	0.43	0.00

# Event Correct:	3	# Event Incorrect:	2	Accuracy: 60%
# Total Selector:	4	# Total Complex:	3	Complexity

5. FURTHER WORK

The current implementation of the *AQ15* runs well, but has a few deficiences. The first and probably the most unpleasant problem from the user's and the programmer's point of view are its **input routines**. The "setup" procedure, which reads in relational tables, is not capable of detecting all syntax errors that may occur in the input, and may therefore abruptly terminate the execution without any error message or even fall into a dead loop.

The second problem is of a more conceptual nature and concerns the **data representation**. Like *GEM, AQ15* manipulates actual examples and spends majority of the time testing if particular complexes cover an individual example. In order to avoid redundant computation, *AQ15* should first enumerate examples and then manipulate the set of numbers representing the events. As a result, when testing which negative examples are still covered by a partial cover, a simple and efficient set intersection operation can be used.

Another very frequent time and space consuming operation is copying sets of examples and complexes. By marking the examples covered and linking the complexes to be copied, the problem can be solved.

6. REFERENCES

R.S. Michalski, On the Quasi-Minimal Solution of the General Covering Problem, *Proc. of the V International Symposium on Information Processing (FCIP 69)*, Vol. A3 (Switching Circuits), pp. 125–128, Bled, Yugoslavia, October 8–11, 1969.

R.S. Michalski, Variable-Valued Logic and its Application to Pattern Recognition and Machine Learning, Chapter in in the monograph, *Computer Science and Multiple-Valued Logic Theory and Application*, D. C. Rine (Editor), North-Holland Publishing Co., pp. 506–534, 1975.

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

R.S. Michalski, J. Larson, Selection of the 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, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, May 1978.

R.S. Michalski, R.L. Chilausky, Learning by Being Told and Learning from Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis, *International Journal of Policy Analysis and Information Systems*, Vol. 4, No. 2, pp. 125–161, 1980.

R.S. Michalski, J. Larson, Incremental Generation of VL_1 Hypotheses: the underlying methodology and the description of program AQ11, ISG 83-5, UIUCDCS-F-83-905, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, January 1983.

R.S. Michalski, I. Mozetic, J. R. Hong, N. Lavrac, The Multi-Purpose Incremental Learning System AQ15 and its Testing Application to Three Medical Domains, to appear in AAAI, 1986.

R.E. Reinke, Knowledge Acquisition and Refinement Tools for the ADVISE META-EXPERT System, M.S. Thesis, ISG 84-4, UIUCDCS-F-84-921, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, July 1984.

K.A. Spackman, QUIN: Integration of Inferential Operators within a Relational Database, M.S. Thesis, ISG 83-13, UIUCDCS-F-83-917, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, 1983.