INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

RAGA: Rule Acquisition With A Genetic Algorithm

By

Robert Cattral, B.C.S.

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science**

**Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario**

December 2000

© Copyright

2000, Robert Cattral, B.C.S.

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*

Our file *Notre référence*

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-57758-9

Canada

The undersigned hereby recommend to
The faculty of Graduate Studies and Research
acceptance of the thesis,

Rule Acquisition With A Genetic Algorithm

submitted by

Robert Catral

in partial fulfillment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

Jan 15, 2001

Abstract

This thesis describes the implementation and functioning of RAGA (Rule Acquisition with a Genetic Algorithm). RAGA is a genetic algorithm (GA) and genetic programming (GP) hybrid designed for the tasks of supervised and certain types of unsupervised knowledge extraction from large and possibly noisy databases. Applying the GA to the task of data mining required that certain changes be made to the traditional algorithm, particularly with respect to knowledge representation. RAGA uses a system of rules of varying length and complexity, while the traditional GA has chromosomes of fixed length using a binary alphabet. Although it is similar to GP in this respect, RAGA uses mutation operators and does not evolve programs. Unlike both GA and GP, it promotes validity and nonredundancy by intergenerational processing on a fluctuating number of individuals. It also implements a form of elitism that causes a wide exploration of the dataset, and, by making data coverage a component of fitness, it automatically evolves default hierarchies of rules.

Table of Contents

Table of Contents	iv
Table of Figures	viii
Thesis Motivation and Goals	1
Introduction to Data Mining.....	3
Importance of data mining	3
Training and testing sets.....	4
Accommodating changes in data.....	7
Directed versus undirected data mining.....	8
Combining directed and undirected searches.....	11
Confidence and support factors.....	13
Varying Target Levels for Confidence and Support	16
Implementing a new rule.....	17
Existing data mining systems.....	19
Introduction to Genetic Algorithms	21
RAGA: Concept and design.....	22
Problems with existing data mining systems	22
Representation of data	24
Evolving and using a rule hierarchy.....	27
Using confidence and support factors	30
Implementation.....	32
Fitness proportional selection.....	33

Crossover.....	35
Mutation	37
Intergenerational Processing	40
Comparing RAGA to other genetic algorithm based systems	45
Data Mining Sessions.....	45
Definition	45
Session variables	46
Trimming the dataset.....	46
Derived variables.....	47
Variable structure	48
Session Flowcharts	54
User interface and general operation.....	56
Main menu.....	56
Creating a new session	56
Opening an existing session	56
Editing the Current Mining Session	56
Conversion of Existing Datasets	61
Randomizing the dataset	64
Definition and rationale.....	64
Procedure.....	65
Configuring the System.....	66
Rule length	67
Fitness function	72

Rule fitness options	73
Termination condition	79
Random number generator	81
Elitism	82
Crossover and mutation rates	86
Intergenerational processing	87
Miscellaneous options	88
Running a Mining Session	90
Testing the rule hierarchy.....	98
Viewing Results	99
Generating synthetic datasets.....	104
Results	106
Classification of the Polygon Dataset	106
Background	106
Results using C5.0 (See-5)	108
Results using R.....	112
Results using RAGA	114
Classification with additional relationship attributes	116
Miscellaneous datasets	120
Classification of olive oils.....	120
Letter image recognition data.....	121
Previous work.....	122
Undirected data mining experiments.....	123

Future work	125
Conclusion.....	126
Appendix A: File formats.....	128
Appendix B: Polygon dataset.....	130
Appendix C: Multi-target search example	131
Appendix D: Traversing a rule hierarchy	134
References	136

Table of Figures

Figure 1: Steps in the evolutionary process	33
Figure 2: Members in the population represented by fitness	34
Figure 3: Session flowchart.....	54
Figure 4: Single experiment flowchart.....	55
Figure 5: Editing a data mining session	57
Figure 6: Restructuring a data file.....	62
Figure 7: Randomizing the dataset.....	66
Figure 8: System configuration	67
Figure 9: Searching the data.....	90
Figure 10: Manual rule addition.....	94
Figure 11: Viewing results	100
Figure 12: Filtering results	101
Figure 13: Decision tree output from R for classification of polygons.....	112

Thesis Motivation and Goals

The motivation behind developing RAGA, the data mining system being described in this thesis, stems from the interest in having a system or technique that is able to complete various types of data mining tasks within a reasonable period of time. Because there are very few data mining algorithms that make use of genetic-algorithm-based engines, the primary goal is to develop one and experimentally compare it to other well-known algorithms.

In order to measure the value of results generated by RAGA, a number of other data mining systems are tested and compared. The initial experiments in the thesis will test RAGA against other learning systems while obeying the constraints in place for each one. Particularly, for systems that support only 1-place predicate learning, RAGA will not be allowed to act as a relational learner. The primary competitor is C5.0 [Quinlan (1993)], which has been in development for many years and is quite successful at creating decision trees using 1-place predicates. The goal in these first experiments is to have RAGA generate results that compete successfully with this application under these conditions.

If the results from the first set of experiments show that RAGA performs adequately, then the next group will ignore the restrictions found in other systems. Specifically, RAGA will be allowed to act as a relational learner if the conditions are favorable. The goal in these experiments is to show that a system capable of relational learning can generate results that are unattainable using 1-place predicates.

At all stages in development, special considerations are made to ensure that RAGA is flexible enough to handle undirected data mining tasks, which represent a class of problems for which very few algorithms (particularly evolutionary algorithms) exist. Because of the difficulties in both implementing and testing such a system, the algorithm will be tested on synthetic datasets where the underlying rules are known by the author. The final goal is to use RAGA to discover knowledge in this type of domain, and positively compare it to the known rules for the data.

Introduction to Data Mining

Data mining is defined as extracting structured information, such as patterns and regularities, from databases. This information is important because it provides means for understanding the data, including the generation of predictive rules. Although this is clearly defined, it has been used as a catchall term for several methods and procedures within the more general field of *knowledge extraction from data*.

Specifically, data mining refers to the use of methods to extract predictive information only. Some of these methods include generating decision trees, training neural networks, clustering data using algorithms such as nearest neighbor, and rule induction. The process does not include data storage and maintenance (data warehousing), data visualization, or ad hoc query and reporting.

The goal of data mining is to learn from historical data and gain the ability to make prospective decisions in related circumstances. Some types of problems that are addressed with data mining techniques are *classification and segmentation, forecasting, association rule extraction, and sequence detection*.

Importance of data mining

Data mining allows companies to analyze large databases to solve business related decision problems. In the past this approach has been used for applications such as mass mailing campaigns, where customers are selected for mailing based on several criteria.

Under normal circumstances a mass mailing has a 1% customer return, however rules discovered in previous data allow for a greater return. This translates directly into savings in terms of postage, which can re-budgeted for other projects. The measure of improvement is known as the *lift* [Berry, Linoff (1997)].

Factors such as the surge in electronic commerce, the popularity and ease of recording large amounts of sales data, increased computing power and better algorithms have generated a great deal of interest in data mining tools. With the amount of data available and the competitive advantage being realized by those participating, it is expected that the data mining industry will continue to grow at an increasing rate.

Training and testing sets

Independent of the type of search problem, the data mining process generates rules that have to be tested for accuracy. It is not sufficient to simply create a set of rules based on the entire available dataset because they may not be general enough to handle other data from the same problem domain. The important test for a new rule set is to see how it performs on non-analyzed data from the same source. Without this control it is possible to develop rules that perform well for a subset the given problem, but are either too general or too specific to properly address the problem as a whole.

To facilitate the rule judging task there have been several protocols developed, all of which involve splitting up the available data into *training sets* and *testing sets*. The training set is that segment of data used to create the rules, while the testing set is used to

check the predictive value of them. One of the common methods for splitting and testing datasets is called *N-fold cross validation*.

Cross validation refers to a multi-step algorithm that alternates between creating and reviewing rules, and is popular for testing the merit of data mining algorithms. The input required is the entire dataset, and a value for N , which indicates the number of elements in each of the testing segments. The following steps detail the operation of the algorithm:

Retrieve dataset and the value for N

Split the dataset into N segments of approximately equal size (segments labeled 1- N)

Loop through segments (1 to N)

$TS :=$ the segment indexed by the current loop counter value

$TR :=$ the collection of all segments excluding TS

 Create a set of rules (R) using TR

 Test (R) against all elements in TS and record the results

Tally and summarize the results

The result of running the algorithm above is a collection that contains N sets of rules, where the majority of the rules are common between sets. Infrequently found rules are normally contradictory to the rest of the dataset and are given less priority than others. Associated with each set is the accuracy as determined through testing, and the summary statistics provide information on the average accuracy over all rules.

Depending on the size of the dataset the value for N can vary, however there are a number of common choices. One such choice is 50% of the data, where half of the data is used for testing, while the other half is used for training. The problem with this selection is that 50% of the data is often not representative of the entire set, and thus can lead to rules that do not accurately describe the data in general.

Another common choice is called the *leave one out* method. This is the same process, where the value for N is the number of all elements in the dataset, minus one. Although this is a rigorous test of the dataset and algorithm being used, it takes considerably longer than the 50-50 method.

Regardless of the ratio or method chosen, it is important to ensure beforehand that the elements are not ordered in any way. This often requires an additional step referred to as *randomizing the dataset*, which is discussed in detail in the corresponding section on page 64.

Regardless of how comprehensive the testing is, the nature of some domains dictate that the final set of rules will have a limited lifetime. This is especially true in cases where data is constantly in a state of gradual change, as well as data that changes drastically because of a switch in season or other reoccurring event.

Accommodating changes in data

There are several reasons that rules developed for a particular domain can expire or become invalid. The first reason is due to a shift in trend, where certain attributes change gradually over time. This is similar to an effect known as *concept drift*, where details about a subject change a little bit at a time until eventually the basic idea has been completely transformed.

This problem is more prominent in datasets that are constantly being added to, such as complete historical logs. As time progresses and trends change, a prediction based on the entirety of all past experience is not necessarily a good indicator of what is forthcoming. Similarly it is possible that rules will become contradictory to the norm, and contribute negatively to the forecast. One way to address this problem is to expire elements after a set time period, which is equivalent to training with only a recent subset of elements.

This approach works because the variations are small and not significant over limited periods of time. The size of the window depends on the transaction volume and the frequency of change. For the case of concept drift this method works very well, however it is inappropriate for sudden changes caused by cyclic or seasonal trends.

For data that is cyclic, such as the case with seasonal buying patterns, it is important to recognize that certain rules within the set will alternate between valid and invalid depending on the time of year. Further complication to this problem is added by the fact that dates may not be fixed, and there can also be several different dates throughout the

year where different items change in popularity. The effect is that even a stable set of rules can have several subsets that are regularly rotated.

Examples of this are obvious in retail environments, where the shift in season from fall to winter introduces and expires several types of clothing. Similarly, there are a large number of holiday specific related items that are typically only sold for very short periods.

For these cases it is beneficial to have several sets of rules, where one contains elements with little or no variation, and the others are specific to cyclic patterns. While the main set of rules is used constantly, the extra sets are used to provide additional information such as specific supply and demand requirements of particular items during certain times.

Directed versus undirected data mining

During the knowledge discovery process there are several degrees to which the user can help the system by supplying information. The two main categories are known as *directed* and *undirected* data mining.

Directed data mining is also referred to as *supervised learning*. This method attempts to find concept descriptions for classes that are, together with pre-classified examples, supplied to it by a teacher. The problem with supervised learning is that it relies on human experts to create the class, as well as update them periodically to reflect changes in trend.

Unsupervised learning refers to the discovery of knowledge with little or no information supplied by the user, where the goal is to simply find something interesting about the data. To illustrate the difference between these two approaches, consider a system that rates the quality of coffee beans.

In either system the dataset contains information such as the origin, physical description, chemical breakdown, and other objective information concerning the coffee beans.

Unique to the directed classification system is the presence of an additional field, which is a rating of quality by a human expert. In this case it is safe to assume that the rating is not binary, but rather has several degrees of quality.

After examining the attributes and class for each element in the dataset, a supervised system will create predictive rules that determine quality using the basic information only. These rules can afterwards be used to classify coffee beans in the same fashion, but in the absence of the human expert.

In an undirected system the indication of class is not present, and the process is expected to create groups based on the basic information only. In cases where a human expert has already determined the appropriate classes for the domain, one or more of the automatically created classes may correlate to them, however this is not guaranteed. It is possible to have fewer groups because of ambiguity in the data or large overlaps in classes, but it is more likely that a superset of groups will emerge.

The reason for extra groups is that more than one type of classification will occur concurrently. For example, groups can still be formed to describe quality, but other groups may describe attributes such as bean type or family. One of the difficulties involved with undirected data mining is the analysis of the results, where meaningful labels have to be given to each of the groups. Because the system has insufficient information for this task it is left up to the researcher in the post-discovery stage.

It is possible to have more classes than was deemed necessary by the expert because unrecognized patterns and similarities are discovered. During the results analysis these groups can be refined, joined, or separated depending on what granularity is required for the set of classes.

The main disadvantage with undirected data mining is that the process is much more demanding than supervised learning, and takes considerably longer to perform. The tasks range from detecting potentially useful regularities among the data couched in the provided description language to the discovery of concepts through conceptual clustering and constructive induction, and to the further discovery of empirical laws relating concepts constructed by the system. Because of the variety in tasks and extra time required for both configuration and the search itself, undirected data mining is not possible for all projects.

Despite the penalty in time, undirected searches are becoming more widely used for two main reasons. The first is that some domains lack expert advice and opinions, and an appropriate classification scheme simply does not exist. In some cases there are clues as to what classes should exist, but in other cases there is no opinion as to what characteristics should be segmented into groups. The second reason is to try and break away from the bias created when training with a single human expert. As expert systems learn from example, they are credited by mimicking the teacher as closely as possible. Because of the possible difference in opinions between experts it is not always beneficial to use only one teacher for extended periods of time.

Combining directed and undirected searches

As one possible solution to the problem created by using a human expert, unsupervised learning techniques can be employed to create an unbiased look at what characteristics make up different groups. When this type of search concludes, the results can be analyzed and the final groups used as the new set of classes. These classes, together with the associated data, can be used to train a supervised system and create a set of predictive rules.

The instructions for a multi-step search process are as follows:

- 1) Using unsupervised learning, examine the data for relationships. Common attributes that segment the data will become apparent and help to focus the search. This process is also known as *feature selection*.

- 2) Still within the unsupervised learning session, attributes found to organize the elements into common themes will be examined and combined to create a set of classes.
- 3) Once complete, a group of unlabeled classes will exist, which should be analyzed to determine what features or characteristics they represent. It is also important to determine what the coverage of the dataset is, and whether enough groups exist to potentially classify all of the elements. In some cases there may be data that does not fit into any class, however it is also possible to have data that belongs in several classes. It is possible for a single element to belong to several classes because the undirected process allows for elements to be classified based on any number of criteria, and the groups are not in competition with each other.
- 4) Once the classes are analyzed and labeled, some or all of them should be selected for use in the supervised search. Classes that do not structure data in a useful way should be excluded unless they are required for complete coverage. For example, in retail transactions the cashier identification may be part of the data and consequently a class is created for each cashier. Although this may be important for some search goals, it is not likely to be of interest in most cases. With few exceptions, this type of class should be excluded during the creation of predictive rules.
- 5) Using the classes found in the above steps, perform a directed search to create the set of predictive rules used for classification.
- 6) Test the final set of rules against an unseen segment of the original data, and analyze the results.

The advantage to running the multi-step search process detailed in the steps above is that the resulting set of classification rules are for classes that were determined by the algorithm, and not by a human expert. Using this type of procedure minimizes the level of human bias in the results, with exception for the inclusion or exclusion of certain redundant or meaningless (in the context of a particular search) classes.

Confidence and support factors

To determine the quality of a proposed rule there are a combination of statistical and subjective factors to consider. These include the rule accuracy, coverage, how useful and how interesting the rule is within the given domain. Because the latter two features are subjective in nature there is no widely accepted method available for determining them, and for the same reason there is no common scale to rate or compare them. The measures for accuracy and coverage of association rules are known in other areas of information theory as confidence and support factors.

The confidence factor is a measure of the rule accuracy. It is defined to be the percentage of times that the consequent is true given that the antecedent is true. If the consequent is false while the antecedent is true, then the confidence fails for the given rule. If the antecedent is not matched by the given data item, then this item does not contribute to the determination of the confidence of the rule.

The support factor is defined as the rule coverage, which is how often the rule is correctly applied within the entire dataset. This measure is calculated by dividing the number of elements that are correctly answered using the rule by the total number of elements in the set. The basic information conveyed is how often the rule is actually used, which is an important factor to consider when deciding its worth.

There are no perfect values to use when setting confidence and support factors, however there are a few recommended guidelines that should be considered. For a supervised classification system the most often sought goal is to discover a set of rules that have as high a confidence and support as possible. The reason for this is that classification systems ultimately strive for 100% predictive accuracy, and are therefore judged on how many rules can be answered with perfect accuracy. In the case of a dataset that can be classified with no error, has a single rule for each class, and has data that is evenly distributed between all of the classes, the confidence for each rule will be 100% and the number of classes or rules will evenly divide the support.

Although it is possible to search for rules with a lower confidence or support it is not obvious as to why this would be useful in a classification system. One reason is to try to discover large-coverage rules that are within an acceptable margin of error. For example, it may be beneficial in a certain domain to have a single rule that attempts to classify 55% of the elements, even if the accuracy is penalized. Although the confidence is lower than what can be achieved with more refined rules, it can still be acceptable for certain applications. The advantage to this style is that the rule hierarchy remains small.

Having a smaller set of working rules is advantageous only at the time of classification, and serves two basic purposes. The first is that more general and wider-coverage rules help to curb the negative effects of *over-fitting*. The second advantage concerns only scenarios where a human operator will be using the rule set to make predications that lead to informed decisions.

In some cases a human operator must read and understand rules in order to make quick and informed decisions. This type of situation requires that the set of rules be as accurate as possible, but more importantly they must be easy to read and understand. In many cases it is not time-efficient for an operator to scan through a large set of rules. In addition, there is higher probability for incorrect classification, which negates the advantage of having a many more rules to increase predictive accuracy.

In undirected searches the approach must be different because setting confidence and support goals too high will often cause the search to be ineffective. The reason for this is that the rules being sought are not obvious, and those with high values are probably already known. In general, if a rule has both a high confidence and support it is usually an axiom or another type of domain specific knowledge. While this is useful for tasks such as classification, it contributes very little towards the goal of knowledge discovery.

Varying Target Levels for Confidence and Support

When attempting to discover new information from data it is important to remember that the most interesting rules are usually obscure and not easily found. This tends to suggest that a low support factor is desirable for searching out such information, however caution must be used in setting both target values appropriately.

Two problems frequently experienced by analysts stem from searching for rules with too high or too low support [Berry, Linoff (1997)]. It is true that a very low support factor will find obscure information, however if it is too low then the results will be of little use because the event happens so rarely. Even in the case where a rule is 100% confident, if it occurs too infrequently then it is unlikely that anybody can take advantage of it. This is partly because any new rule has to be applied, and this step typically comes with an associated cost. The cost of implementing a new rule is described later in this section.

When the target value for support is set too high then problems occur that are similar to those experienced with a high confidence. Particularly, common associations that are well known and often quite obvious within the domain are found. This type of information is rarely interesting and not likely to have any value.

Finally, when the confidence is too low then rules are typically of little use because the results cannot be guaranteed and the cost of implementing the rule is wasted. Considering all of these factors it is important to realize that there are no fixed values that will work for every run, however a general template can be followed.

Rather than perform a single search using a standard set of values, such as 85% confidence and 10% support as some analysts suggest [Berry, Linoff (1997)], it proves beneficial to search the data several times while varying the levels. Because of the time required to run a single search session it is not reasonable to try every possible pair of values, even within small ranges for each. As one way to tackle this problem, several searches can be performed using variations of the suggested values. The cumulative process will often lead to new rules and other information that is missed during other runs.

For an example of a multiple-target search session where both confidence and support factors are varied in the RAGA environment, please refer to the *Appendix C* on page 131.

Implementing a new rule

The cost of applying and testing a new rule is determined by looking at the effort required to exploit it. For example, if it is recognized that two products are occasionally purchased together then there are several options available to the management.

The first option is to implement policy such that these two items are never put on sale at the same time. The reason for this is that retail promotion sales are intended to entice people into trying a product with the hope that they will purchase it again in the future. Putting items that are known to sell well together may not interest more people than one of the items alone, however it would most likely decrease the combined profit.

A second option involves products that are not heavily dependent on their brand name. In the case that two items sell well together, and the retailer has their own line of each product, then the proprietary versions can be moved close to one another. The reason for this is that retailers typically make more money when selling their own brand, and thus encouraging those to be sold together yields more profit. Similarly, this also works if only one of the products is a brand name and is moved closer to their own brand.

It is important to recognize that each of the above options involves a cost in terms of time, whether it is for planning promotion schedules or relocating items. This represents all or part of the implementation cost. In addition to the expense incurred during implementation of the rule, there can also be unexpected or unforeseen disadvantages.

One possible side effect is that the sale of the displaced item decreases. This happens in cases where a very popular item attracts attention, and other products adjacent to it sell regardless of brand name or quality. The data mining process may have discovered that the two items sell well together, however the profit could amount to less now that the displaced item is less popular.

In summary, the cost of implementing new rules can be significant and should be factored into decisions based on the results from data mining sessions. Monitoring the results from policy change should also be done to determine whether any obvious disadvantage occurs because of the change.

Existing data mining systems

There are several popular data mining systems in use today that use a variety of techniques including statistical and machine learning methods. Some of these are:

CART: Classification and Regression Trees

Breimam, Friedman, Olshen, and Stone (1984)

This technique examines the dataset and creates decision tree classifiers.

C4.5 / C5.0 (See-5)

Quinlan (1993)

This application uses machine-learning techniques to create classifiers in the form of decision trees and rule sets.

R: A Programming Environment for Data Analysis and Graphics

Rick Becker, John Chambers, and Allan Wilks (1998)

Based on the S language, used for the statistical analysis of data.

Includes a variety of functions for data analysis including the creation of decision trees.

The primary limitation that exists when classifying data using the systems above is that they are restricted to 1-place predicates, which means that attributes within the dataset can only be compared to constant values. For example, to discover a rule that compares

the length of an object to its width, a certain amount of preprocessing is required before the search begins.

Although this causes a number of problems, the main fault is that it restricts the boundaries of the search space. Without being able to properly address all of the data in the set, rules are created such that they are non-scalable in many problems. This is the primary reason for the creation of RAGA, and is more completely described on page 22, in the section on *concept and design*.

Introduction to Genetic Algorithms

A Genetic Algorithm [Holland (1975)] is one example of a heuristic search that is used to find solutions to optimization problems. In an optimization problem there are many known solutions, however the task is find the *best* one. Whether one solution is better than another depends on the problem domain, however two common examples are the solution yields a more accurate answer, and the solution represents a process that is faster to execute than another.

GAs are probabilistic heuristics and are not guaranteed to find the optimal solution to any given problem. however they have been used with success since the mid-Nineteen Eighties in a variety of applications. Some of these include biology, engineering, pattern recognition, the physical sciences and the social sciences [Goldberg (1989)].

The strength of a GA is its ability to search large finite spaces by *evolving* complex solutions. These are evolved through the use of a special set of *genetic operations*, where simpler solutions are modified and combined in an environment where they are forced to compete with each other. The entire process is inspired by Darwinian natural selection and the field of genetics.

With some exception, GAs have not been used for the task of data mining. One of the main arguments against using GAs is that the structure is too limited to handle the information required to evolve rule sets of decision trees. The limitations and solutions to various problems are described in the next section.

RAGA: Concept and design

RAGA was created for the purpose of applying Genetic Algorithms to the task of rule induction, and specifically avoiding problems found in other Data Mining systems.

Problems with existing data mining systems

As described briefly in the *Introduction to Data Mining*, there are a number of algorithms and implementations that perform rule induction. The problem with many systems is that they operate using only 1-place predicates, which allows for the comparison of an attribute to a constant value. Depending on the problem domain, solutions of this type may not be scaleable.

A good example of this limitation is shown later during the description of training rules for classification of the polygon dataset, which consists of a number of simple shapes described only by the lengths of each side. Specifically, each of the attributes is an integer value, where three non-zero values represent a triangle, or four non-zero values represent either a square or rectangle. Triangles are more specifically classified as equilateral, isosceles, or other.

Of the three systems used for experimenting on this dataset, which were C5.0 (successor of C4.5), R and RAGA, only RAGA was capable of recognizing relationships between variables when the data was in its raw form. It is still possible for other algorithms to consider these relationships, but this requires that the data first be tailored in a pre-processing stage.

Customizing the data to handle inter-attribute relationships requires adding derived values that express the result of a known operation. For example, in order to compare the length of an object to its width, the addition of a new attribute is required. One possible solution is to create a value for length divided by width. This new variable can be examined by any classification system, and predictions can be made based upon it. One trivial observation is that a value of 1.0 would mean that length equals width.

There are a number of disadvantages to creating new attributes during the pre-processing stage, including a trade-off between an increase in the search space and increased input by the human expert. The trade-off is necessary because the types of relationships as well as the applicable attributes must be determined in advance. For cases where human bias is acceptable, the expert can try to determine beforehand which attributes should be compared, and how. Unfortunately, given the fact that data mining techniques are in place to discover information, this is not always possible because the expert does not expect or understand the data well enough to decide.

In cases where the expert is not able to isolate key attributes or determine what possible relationships should be tested, then a new column must be generated for each pair of attributes using each of the available operations. This increases the size of the search space dramatically, making the process much longer and less likely to succeed.

Furthermore, in cases where more than two attributes can be related, the increase in

search space is exponential. Many of these cases result in a search that cannot be completed in a reasonable period of time.

To summarize, adding relationship attributes during the pre-processing stage is a poor choice because the human expert must decide upon both variables and the operations. This itself is a task in data mining, and although classification systems can take advantage of them, they are still unable to discover them.

For an example of the advantages realized using RAGA in this situation, please see the search sessions using the polygon dataset on page 106.

Representation of data

The way that RAGA represents data is the primary difference between it and other Genetic Algorithm based systems. In a standard GA a fixed length binary string represents each individual. A decoding function exists to convert this string to the appropriate form when required (ex: convert binary string to an integer value). This representation is too restrictive for the predictive rules that are generated by RAGA.

Standard GA representation cannot maintain the information required for each rule. The first problem is the fact that solutions are fixed in length, which limits the structure of rules. Although it is possible to specify an upper bound that is sufficient to hold an entire rule, there are many different sizes of rules within any given set. Accommodating this

within a fixed length structure would require the use of NOP (no-operation) values, and will not be as efficient.

The alphabet used by a genetic algorithm is called the set of *terminals*. In a standard GA there are only two of them, which are the digits zero and one. Variations of GA systems, such as Genetic Programming [Koza (1992)], use a much larger alphabet that is dependent on the problem being solved. In a typical GP system there are numerous terminals that represent both functions and variables.

The type of data being sought by RAGA is known as an association rule, which is of the form: *If* $X_1 \wedge X_2 \wedge \dots \wedge X_n$ *Then* $Y_1 \wedge Y_2 \wedge \dots \wedge Y_m$.

The symbols $X_1 \dots X_n$, and $Y_1 \dots Y_m$ each represent terms within the rule, where a term is a function that either indicates the existence of an attribute, or performs an operation on two or more variables.

Rules are represented within RAGA as dynamic structures that store the information required to present and evaluate the rule. RAGA is similar to GP in this respect because it has both a user-defined number of terminals, and the length of the rule is flexible.

A flexible set of terminals is required in RAGA because different datasets have a different number of attributes, and without accommodation for each of them it is not

possible to fully explore the data. In fact, some datasets require that more variables than attributes be accessible for the purpose of using derived values.

Each rule is represented by a structure that contains several subcomponents:

1. English name
2. Class
3. Length of antecedent
4. Length of consequent
5. List of terms in the antecedent
6. List of terms in the consequent

Within both antecedent and consequent (items #5 and #6) there is a list of terms that makes up the corresponding portion of the rule. Each term is itself a structure because several pieces of data are required to represent it. The list of subcomponents for each term is:

1. Index of variable on left side of function
2. Index of variable on right side of function
3. Function (ex: comparison operator)
4. Constant value (present if the right side index is empty)

Exception: Only item #1 is used to represent Boolean variables

Evolving and using a rule hierarchy

One of the characteristics of RAGA is that it evolves rules in the form of a default hierarchy, with the emphasis placed on creating a strong group rather than strong individual rules. Using the approach, rule sets are kept simple in both individual rule complexity and the total number of rules that are required.

A default hierarchy is a collection of rules that are executed in a particular order.

Whenever a new element is tested against the list, the rule at the top is tried first. If the antecedent correctly matches the data then the rule is used. If the data does not satisfy the antecedent, the next rule in descending order is tested. This continues until a matching rule is found, or the rules in the hierarchy are exhausted.

Using the method means that not every rule has to be 100% confident in order to obtain perfect accuracy overall. The higher levels of the hierarchy play an important coverage-extending role by *protecting* less confident rules, as in the following example:

If (numberOfSides = 4) ^ (length = width) then class = square

If (numberOfSides = 3) then class = triangle

If (numberOfSides > 2) ^ (numberOfSides < 5) then class = rectangle

In this example it is important to realize that the first two steps are independently correct, and can be transposed without affecting the results. The third rule is different because it is

position dependent, and is only able to accurately answer those elements not addressed by the first two rules. If this rule were used in isolation on the same dataset, it would result in a large number of misclassified elements.

Although not obvious in the simple hierarchy shown previously, experimentation has shown that rules at the top of the hierarchy cover most of the data, while those near the bottom often handle exceptional cases. Unlike a set of rules where each exclusively answers a subset of data, these are typically simpler and have fewer constraints because they can rely on the protection and support from others. Although there is often some overlap between elements it is only the first matching rule that is used, and no ambiguity is introduced into the results if this occurs.

Depending on the dataset, this hierarchical approach normally requires fewer and less complex rules to successfully classify the data. This is partly because simpler rules are faster and easier to construct, and also that each one covers a larger area as compared to a very specific rule.

Because maximizing data coverage is one of the goals when evolving a hierarchy, it follows that *over-fitting* the solution set is a potential problem. An over-fit set of rules is one that has a very high accuracy when tested on the original training set, however it is not general enough to accurately answer other data from the same source. This problem is taken seriously in classification tasks, and mechanisms are generally in place to deal with it.

Two popular ways to curb the effect of over-fitting are pruning the results and relaxing the constraints. When a decision tree is pruned it has the effect of grouping similar items together based on their position within the tree. Similarly, relaxing the constraints is done such that more general cases apply to each available class. The problem with these methods of generalization is that they come at the cost of trading-off predictive accuracy. Although generally accepted, this type of solution also creates new problems because some cases simply cannot be classified accurately as a result.

In RAGA, no trade-off is made between predictive accuracy and maintaining more general rules. Instead, the default hierarchy is used to identify and resolve cases of over-fitting. This is possible because rules that answer very few elements are in the lower segment of the hierarchy, and can be easily identified as outliers or other rarely occurring cases. For example, a very unpopular rule might be one that handles only 1 or 2 out of perhaps 5000 data elements.

It is important to note that this type of over-fitting is harmless because these rules are only tried as a last resort. If after all of the general rules are tested unsuccessfully, then the element still might be classified by one of the exception cases. This is not the same as using a default class, where an element is placed into a popular group simply because it could not be identified otherwise.

Assigning a default class is often done as a last resort in order to extend coverage. In a decision tree, the procedure is to simply assume that unclassified elements each belong to the class that holds the majority of elements at the previous node. In many cases this successfully extends the coverage, but the only basis for applying it is that one class is more popular than the others. Obviously, when dealing with subsets of data, this will not always be the case. In RAGA it is still possible to use a default class by setting it as the last rule in the hierarchy. In this position the class will be assumed only if the element fails when tested against every other rule in the hierarchy.

The only drawback to using a default class or limited coverage rules is that in some instances it is less costly to leave the class unknown, rather than to misclassify it. In these cases a default class should not be used, and in RAGA it might involve scaling back the hierarchy such that the bottom segment is not used. For example, arbitrarily choosing to use only the top 80% of rules would leave several items unclassified, but would certainly curtail the problem of over-fitting.

For an example on traversing a rule hierarchy, please see *Appendix D* on page 134.

Using confidence and support factors

In order for a genetic-algorithm-based system to effectively search for solutions, an appropriate fitness function must be implemented. In a data mining system, the ultimate fitness function would act as a measure for how close the rule is to what is being sought.

Because this measure is subjective, particularly in undirected mining tasks, there is no straightforward way to implement such a measure.

With only a limited amount of information available for each potential rule, an effective fitness function has to combine as much of this data as possible. In RAGA, the raw fitness is determined by using the values for confidence and support.

$$fitness = \sqrt{(confidence - confidenceTarget)^2 + (support - supportTarget)^2 * sfAdjustment}$$

This equation converts the values for confidence and support into a single numerical value that increases as each factor approaches the respective target. Basically, the closer the values are to the user specification, the higher the fitness is for the rule.

An optional adjustment can be applied to change the importance of the support factor with respect to the rule confidence. This extension is the last term in the equation, and is called the SF-Adjustment (support factor adjustment).

By default the SF-Adjustment is 1.0, which means that both confidence and support factors are weighed equally when calculating the raw fitness. Depending on the dataset and the domain it might be important that a higher confidence is preferred to a higher support. In these cases entering a decimal value greater than zero but less than one will lower the impact of the support factor. Conversely it is possible to raise the impact of the support factor by entering a multiplier values that is greater than one.

It is important to note that this equation derives that value for raw fitness only, which is later modified based on several problem-specific parameters. These parameters are fully described in the corresponding section on page 72.

One advantage to deriving the raw fitness from the confidence and support factors is that it provides an unbiased subset of the data. It is possible to substitute the fitness function for one that scores rules according to an underlying theme, or relevancy hierarchy. The problem with this approach is that a human expert originally configures it, and then the bias extends through to the final results. When using confidence and support factors, the only human interaction is the specification of the target levels.

Implementation

The genetic engine used by RAGA is a hybrid of GA and GP, with several modifications and additions to the standard models. This section describes the details of the evolutionary functions, and highlights details between RAGA and each of the standard approaches.

Figure 1: Steps in the evolutionary process

Fitness proportional selection

In order for members to be copied from the current generation into the next they must first be selected. Rather than a random selection, which would not necessarily improve the quality of individuals in the next generation, the selection is performed with a bias towards the more fit members. Although not guaranteed it is more likely that the

members with a higher fitness will be selected for, and those with poor fitness will be ignored and discarded.

The selection works by first examining all of the members in the population, and then creating a value that is the sum of all fitness values. Because the fitness values for members differ it means that the range is not evenly distributed, and in fact is more representative of the better individuals. When a random point in the range is chosen it can correspond to any individual, however it is more probable that it belongs to a member with a high fitness.

Figure 2: Members in the population represented by fitness

As can be seen in the previous pie graph, a randomly selected point can be within any of the members, however it is more likely to happen in the individuals that have a higher fitness.

The dilemma with fitness proportion selection is that because the more fit individuals are constantly being preferred, they are usually selected several times within a single generation. When this happens over several generations, a single individual with a high fitness tends to get preferred, and eventually dominates the entire population. This is a very bad scenario because all of the diversity in the population is lost, which means that future generations in the search will provide no benefit whatsoever. This is also referred to as *premature convergence* because the members of the population have converged to a single answer before fully exploring the search space.

Although RAGA uses fitness proportional selection it does not fall prey to premature convergence because unlike standard GA, duplicate rules are not allowed to co-exist within the population.

After a single member has been selected for the next generation there is a probability that another genetic operation will occur. The first of these operations is genetic crossover.

Crossover

Crossover is a technique that is applied to two individuals in an attempt to generate one that is superior. This requires two distinct individuals and a random point in each where

the sequence of terms is split. The rules are broken and then recombined with those from the other rule, which creates two new members.

To illustrate this process, the two rules below were selected for crossover. An arbitrary point was chosen for each rule within the antecedent. In each of the rules below, the bold print indicates the first of the two sections as defined by the crossover-point.

1: If (**A < 5**) ^ (B > 10) ^ (C <= 20) Then (class = 1)

2: If (A > 0) ^ (**C > 2**) ^ (D < 4) Then (class = 2)

When these two rules are crossed over, the following two rules are produced:

3: If (**A < 5**) ^ (A > 0) ^ (C > 2) Then (class = 1)

4: If (B > 10) ^ (C <= 20) ^ (D < 4) Then (class = 2)

In RAGA, this group of rules (1-4) is called a rule *family*. Exactly two of these rules, chosen according to the highest fitness, will be selected for the next generation. This differs from some GA implementations where crossover will indiscriminately select the child-rules and discard the parent-rules.

One of the reasons for this difference is that standard GA systems do not consider the possibility that the child-rules are invalid. Because crossover operates at the term level, with the ability to change both size and structure of rules, it is capable of creating rules

that are not allowed by the restrictions set forth by the user during the configuration stage. When an invalid rule is detected after a crossover operation, it is removed from the family immediately.

The crossover operation does not necessarily generate all of the members in successive generations because it is only used on a probabilistic basis. The probability is defined by the user and is randomized on a per-rule basis. For more information on setting crossover rates in RAGA, please see the corresponding part of the *user interface* section on page 86.

Finally, after the two family members are selected for the next generation, each will be copied pending the possibility of mutation.

Mutation

In RAGA there are two types of mutation, each with a range of possible values. The first type is called micro-mutation, and operates within the terms of each rule. The second is called macro-mutation, which does not affect the specifics of an individual term, but rather is used to add or remove terms from either side of the rule.

The goal of micro-mutation is to modify the terms in an attempt to add variation to the hierarchy. This type of mutation is capable of changing the term variables, constants, or the type of operation.

Example

If $(A < 5) \wedge (B > 10) \wedge (C \leq 20)$ Then (class = 1)

The probability that a micro-mutation will occur is uniform across all variables, constants, and operators within all terms in the population. This operation does not consider terms or rules themselves, or anticipate the effect of the change.

If in the example rule above, the variable *B* in the second term was arbitrarily selected for mutation. After the transformation it becomes the variable *C*, as seen in the following rule:

If $(A < 5) \wedge (C > 10) \wedge (C \leq 20)$ Then (class = 1)

Because the new rule is valid, and mutation is the last step in the evolutionary process, the rule above is copied into the next generation. If however the first term was changed rather than the second, the following rule might have been the result:

If $(B < 5) \wedge (B > 10) \wedge (C \leq 20)$ Then (class = 1)

Because the variable *A* in the first term was changed into the variable *B*, it results in a direct conflict with the second term. With two or more such opposing terms the rule is self-contradictory, and will be discarded immediately.

Macro-mutation is similar in that population members are modified and either moved into the next generation or discarded, however the goal is to experiment with the specialization and generalization of existing rules.

To specialize a rule is to restrict its domain by adding additional constraints. The purpose is to increase the accuracy by lowering the coverage. If it works correctly, then the coverage will be reduced only by elements that were being misclassified beforehand.

To generalize a rule is the opposite of specialization, where constraints are removed in an attempt to increase coverage. If this works correctly, then the constraints are relaxed enough to answer more elements without the penalty of a reduced accuracy.

Example

If (A < 5) ^ (B > 10) ^ (C <= 20) Then (class = 1)	[Original rule]
If (A < 5) ^ (B > 10) ^ (B < 20) ^ (C <= 20) Then (class = 1)	[Specialization]
If (A < 5) ^ (C <= 20) Then (class = 1)	[Generalization]

Both mutation types are used together during the evolutionary process, and can be controlled using varying probabilities. For more information on setting the two mutation rates, please see the corresponding part of the *user interface* section on page 86.

Intergenerational Processing

Due to the fact that the genetic operations behave somewhat randomly it is often the case that rules are modified such that they become invalid afterwards. RAGA handles this situation by using a process named *Intergenerational Processing*, which uses a non-evolutionary approach to modify and replace rules. This is intended to preserve the validity and efficiency of each rule.

Mechanisms of this type are not typically used in Genetic Algorithms, however the idea is similar in some respects to a Genetic Programming variant known as Typed-GP [Montana, 1995]. Typed-GP guarantees that evolved programs will not contain critical errors, thus saving processing time by eliminating worthless members from the population.

A popular argument against using this type of mechanism is that it is *cheating*, where to cheat means that forces other than natural selection influence the members. The first reason for taking this position is that in theory the selection pressure will eliminate worthless members without interference. Secondly, by eliminating individuals that are not selected against naturally, there is a risk that useful genetic material is being lost as a consequence. Regardless of these points, RAGA allows the user to specify one or more options that override the natural selection process.

The genetic engine in RAGA makes the use of this screening available because of the results from many tests that were performed during the software development stage. It

was determined that by allowing the system to remove certain types of rules, not only was there an improvement in the time required to complete the search, but the rule sets were more efficient and performed better on average.

A good example of the gain offered by using this option was observed during the tenth generation of a classification test. Using all of the same data and parameters, except for the intergenerational processing options, two searches were run in parallel. By the tenth generation there were very few valid rules in the population that allowed individuals without restriction. In the competing system, where screening for invalid and ineffectual rules was being performed, many of the rules were sound and in fact remained in place until the end of the search. This same scenario was observed in many tests and was eventually recognized as the norm.

In order to prevent the loss of diversity expected with the non-evolutionary removal of rules, each deleted member is immediately replaced. This replacement is done through the use of rule modification or regeneration.

When a rule is deemed invalid the first step is to modify the rule such that the outcome is valid. The technique used for this is the systematic deletion of terms from antecedent, consequent, or both. If this is not possible because of the rule structure or an excess of similar members in the population, a complete substitution will be made. If after replacing or modifying the rule the result is still invalid, or is a duplicate with another rule in the set, this technique is continually applied until all conditions are satisfied.

After much of the testing and development was complete it was decided that all of the useful constraints would fit into two categories, namely *elimination of extra numeric restrictions* and *elimination of conflicting restrictions*.

Elimination of extra numeric restrictions includes redundant terms as well as logical tautologies. Rules with these types of properties are not actually invalid, however they may be longer than necessary or even completely useless. In many cases this also erodes the confidence of the rule, as well as the support. Some examples of this type of rule are:

1: If $(A < 5) \wedge (B < 10) \wedge (A < 10)$ Then (class = 4)

2: If $(A < 5) \wedge (B < 10)$ Then $(X = 4) \wedge (A < 10)$

3: If C (exists) Then C (exists)

As will be shown in the next step, both rules #1 and #2 can be modified such that they become valid and more efficient, however rule #3 is a simple tautology that will be automatically removed and replaced with a new rule.

Once a rule has been targeted for modification or removal the system checks another user specified setting that biases the system towards removing terms following a particular theme. The two choices are *eliminating more specific rules*, and *eliminating more general rules*. This choice can also be varied and will have different effects depending on the dataset.

In the first rule above there are two restrictions placed on the variable A , making one of them pointless. The decision of which to remove is described by the following logic:

A) Elimination removes more specific rules. RAGA checks the variable type, valid ranges, and comparison operations for each rule to determine which variation is more specific. For these rules the types are integers, and the valid range is 0-25.

After removing the more specific terms, the new rules to replace #1 and #2 are:

1: If $(B < 10) \wedge (A < 10)$ then $(\text{class} = 4)$

2: If $(B < 10)$ then $(X = 4) \wedge (A > 10)$

In both of these cases it was determined that $(A < 5)$ term was the more restrictive, and hence the more specific term. It was removed from the antecedent of both rules.

B) Elimination removes more general rules. This has exactly the opposite effect as the previous option as the $(A < 10)$ term is removed from the two rules as follows:

1: If $(A < 5) \wedge (B < 10)$ Then $(\text{class} = 4)$

2: If $(A < 5) \wedge (B < 10)$ Then $(X = 4)$

Considering the valid integer ranges, these rules are obviously more restrictive than the set derived using the first option.

Elimination of conflicting restrictions includes rules that are invalid because they are self-contradictory. These rules are more detrimental to the entire set because their confidence is always 0%, and they take space in the population as well as consume system resource for nothing. Following are some examples of this type of rule:

1: If $(A < 5) \wedge (B < 10) \wedge (A > 10)$ then $(\text{class} = 4)$

2: If $(A < 5) \wedge (B < 10)$ then $(X = 4) \wedge (A > 10)$

3: If C (exists) Then $\sim C$ (does not exist)

As with the example for the previous option, both rules #1 and #2 can be modified such that they become valid, however rule #3 is a simple contradiction that will be automatically removed and replaced with a new rule.

In each of rules #1 and #2 the variable A is used within more than one term. Although this condition alone does not invalidate a particular rule, the use of it as shown in both examples does. Specifically, the first rule will never be executed because A cannot be both less than 5 and greater than 10. The second rule is a contradiction because it states that A will be greater than 10 if it is first less than 5.

How RAGA modifies the rules is also dependent on the option directing the removal of more specific or more general terms. In the case that the user selects more specific terms, ($A < 5$) will be deleted. In the reverse case where the user selects more general terms, ($A > 10$) will be deleted. Once again, these examples only hold true depending on the allowed range for variable A .

Comparing RAGA to other genetic algorithm based systems

	Standard GA	GP	RAGA
Selection	FPS + others	FPS + others	FPS
Duplicates	Allowed	Allowed	Not allowed
Crossover	Chromosome splice	Tree node exchange	Term level splice
Mutation	Bit level	None	Multi-level
Termination	Generation or success	Generation count	Multi
Function	Approximation, etc	Write programs	Rule induction

Data Mining Sessions

Definition

RAGA uses a *Data Mining Session* to store information related to a particular format in conjunction with a particular dataset. Each session stores the basic structure of the

associated dataset, as well as information on what operations are allowed. Although each session references only a single dataset, it is possible for several sessions to reference each dataset using different sets of operations.

Session variables

A data mining session in RAGA is essentially defined by a group of variables. These are referred to as *session variables*, and are used when referencing attributes within the associated dataset.

In many cases the session variables directly reference the attributes and are not modified or normalized, which means that exactly one session variable is required for each attribute in the dataset. This one-to-one relationship does not always exist because of the possibility that some attributes are not included as variables for reasons such as inefficiency and security.

Trimming the dataset

Inefficiency can stem from the fact that datasets often contain worthless information, such as meaningless or redundant data. One way to help streamline the search is to eliminate any *collinear variable relationships*, which involve multiple variables that are linearly related to each other. These kinds of relationships are often detected by data mining systems because they are easily distinguished, however rarely considered interesting because the relationship is previously known. A second tactic is to discard attributes that have no relationship to the information being explored, such as the

identification of the data entry clerk. Of course, caution must be used when deciding which attributes are truly unrelated because often there are unexpected and important links.

In order to preserve the confidentiality of data it is standard procedure to remove identifying attributes such as the name, or social insurance number. The number of attributes that can be stripped depends heavily on both dataset, and the type of search being done.

Another reason that the relationship between session variables and attributes may not be one-to-one is that new variables are added without a corresponding attribute. These are referred to as *derived variables*.

Derived variables

A derived variable is defined as an informational field that is used by the system, but does not appear in the original dataset. These types of variables are useful in many cases where the data has not been pre-processed to specifically fit a particular application. For example, it may be important in a particular session to realize that the people represented within the dataset are of a certain age, or within a certain age group. Because the life of a dataset can be several years it is much more popular to store the date of birth rather than age of a person at the time of data entry. A simple derived field is the current age of the individual, which is calculated using the date of birth and the current time as reported by the system clock.

Variable structure

Associated with each variable are several pieces of information that dictate what it represents and how it can be manipulated. The first is the variable name, or label, that uniquely identifies it within the session. The label does not have to be descriptive because it is not used for reporting purposes.

The second piece of information is called the *type*, which is used to indicate the type of data being referenced. The available types are: Integer, Real, Percentage, and Boolean. Integer and Real values are the most commonly used, and indicate that the data is numerical in nature with either whole or fractional values. The percentage type is actually a restricted Real value, where additional constraints are enforced by the system. Finally, the Boolean type is used to indicate the existence of a specific attribute within the data.

When using Boolean types it is important to remember that RAGA does not check for specific assignment values such as *true* and *false*, but rather the existence of the corresponding attribute within the element. This type is primarily used in cases where association rules are being sought, such as in a market basket analysis. If the item is present within the transaction then the variable is true, otherwise it is false. In cases where each record contains an attribute that indicates true or false, then an Integer type should be used. The Integer must use two arbitrary numbers to represent the original logical value. The search process will react as expected on condition that these numbers are consistent throughout the data.

An alternative solution to assigning Integers as Boolean indicators is to optimize the dataset by trimming unnecessary attributes. To accomplish this, the attribute being trimmed must be strictly defined as either true or false, otherwise there will be ambiguity and default values introduced into the data. Once this is verified, the dataset can be pre-processed to remove all instances where the attribute indicates a false value. During subsequent searches, RAGA will read the data and record only those cases where the value appears. This pre-processing step has the added benefit of speeding up the search because less time is required to process each record.

The next piece of information stored within a session variable is the *English name*, which is used for reporting purposes only. At any time a rule is printed or shown in its full form, the English name is substituted for variable names. This makes it easier to read and understand the rules produced during the search.

Also stored with each session variable is the *class* that it belongs to. This information is used to group attributes together such that the search process can compare them to one another. When several variables belong to the same class, each can be compared with either another from the same group or a constant value within the allowed range. An example of this is comparing the length of an object to its width. Both of these variables are related within a single record and therefore cannot effectively be compared using a fixed constant value.

In some cases it is important that certain variables are not compared to each other. An example of this arises in examining retail transaction receipts. To compare the prices of individual items may be reasonable, however the price of a single item should never be compared to the price of the total bill. The reason for this is that unless the transaction contains only one item with no tax applied, then the value will always be lower.

Discovering and maintaining these kinds of rules comes at the same cost as other more practical rules, but they contribute nothing of interest to the final result.

With respect to class, the specification is not necessary with Boolean variables. Because these values are used to check only for the presence of an item, it is not possible to develop relationships with other variables. (ie: A Boolean variable is never compared to anything, but rather indicates existence only). As a result, their class is ignored during the search process.

Another important part of each session variable is the *position*, which imposes restrictions on the variable placement within the rule. This control can optionally restrict the variable to existing in only one of the antecedent or consequent, and can also be used to temporarily disable it.

The position of a variable within the rule depends partly on whether the search is directed or undirected. In a simple undirected search where no classes have been defined, each of the variables should be allowed to exist in either the antecedent or the consequent. This

configuration places no restrictions on the search process and relies on very little input from the human expert.

For more directed searches the variables can be restricted to only one side of the comparison. An example of this is the use of pre-defined classes in a classification system. Each of the classes exists only in the consequent and all of the predictive variables exist only in the antecedent.

Some datasets might have several mining session definitions used for a multi-step search. In this case the first configuration would allow all variables on both sides of the comparison. After one or more search sessions the results are compiled and used to create groups or classes. These classes are defined in subsequent mining sessions and used in place of expert classes for more directed searches.

Rule position can also be used to omit one or more variables during a particular search. These types of experiments are performed to see how important certain variables are, and what the impact their omission has on the results. To achieve this, rather than delete the variables from the mining session, the user simply clears both position options. Effectively this means that the variable cannot be placed on either side of the rule and will remain unused. To restore it in a later search the user need only reselect its proper position.

Other numeric properties for Integer, Real, and Percentage type variables must also be specified within the structure. These include the upper and lower bounds, as well as the decimal precision if applicable. An upper bound is the highest possible constant value that a variable can represent, while the lower bound is the smallest possible value. For percentage types these values are normally 0 and 100, respectively. These bounds are often known for the problem domain, and should be entered accurately whenever possible.

If the boundary values are inaccurate the system could be negatively impacted in two ways. The first problem is that the evolutionary process will not consider certain areas within the search space if the range is too restricted. When this occurs, and the bounds are more constrained than found naturally within the data, it follows that some elements cannot be accurately classified.

The second influence is not on the quality of the results, but rather the processing time required generating them. Confining the bounds as much as possible has the effect of narrowing the search space. Restricting the space speeds up search time considerably, and some problems will be unsolvable without first doing so.

The decimal precision, used for Real and Percentage types, specifies the maximum number of decimal places for each value. In certain scientific applications it is necessary to use several places of precision, however it should only be as large as necessary because the search space also increases accordingly.

For each variable there are also a number of comparison operators that must be either allowed or disallowed. These comparisons are used to indicate relationships for Integer, Real, and Percentage types. The set of comparison operators is: *equal to*, *not equal to*, *less than*, *less than or equal to*, *greater than*, and *greater than or equal to*.

In a directed search where the data is being classified as one of several pre-determined classes, these variables are allowed only the *equal to* comparison. With the exception of a small number of datasets there is no valuable relationship between the numerical values that identify different classes, and therefore no reason for other comparison operators. Basically an item can or cannot be identified as belonging to a certain class, and is not subject to being excluded from or narrowed down to a subset of classes. A second reason for the single operator is that the classes themselves are never present in the antecedent, which means that they cannot be compared against other variables for the purpose of classifying data.

Finally, in cases where the variable does not correspond to an attribute and must be defined, the formula or program must be entered into the session. This program code will derive the variable during the search process, and use the data as required.

More information on session variables can be found in *editing the current mining session*, on page 56.

Session Flowcharts

The first chart describes the workings of the program with respect to creating and working with data mining sessions.

Figure 3: Session flowchart

The next chart describes the workings of the program with respect to a single experiment within a data mining session.

Figure 4: Single experiment flowchart

User interface and general operation

Main menu

Creating a new session

Selecting this option will prompt the user for a new session name using a standard *Save As* dialog box. An existing file can be selected, however the contents of that file will be overwritten. To create a new session using an existing one as a template, see *Save a copy*. When a session is created it will be completely empty, and the user is expected to *Edit* the session.

Opening an existing session

Selecting this option will prompt the user for a session using a standard *Open* dialog box. Sessions cannot be created using this method.

Editing the Current Mining Session

A mining session can be modified at any time by using the built in editing facility. This provides a graphical interface for adding and deleting variables, as well as modifying the details for each.

The persons responsible for editing sessions are those familiar with the dataset and the goals of the project with respect to the type of information being sought. An intricate knowledge of the dataset structure is required to map specific fields into variables that are

used during the rule discovery process. In addition it is important to have at least a basic knowledge of the goals because derived variables may be required.

Edit Mining Session				
Session description:	Sample dataset -- Training: 1200 records total, 200 used for testing.			
Associated dataset:	Sample_testing_200.txt	Browse	Save	
List of variables:		Scan dataset	Import	Cancel
class	Add variable	Delete variable	Export	Defaults
colour length width depth shape	<div>Variable parameters:</div> <div>Type: Integer Upper bound: 4</div> <div>English: Class Lower bound: 1</div> <div>Class: cl_class Precision:</div> <div>Rule positioning: ☐ Antecedent Operators: ☒ = ☐ !=</div> <div>☒ Consequent ☐ < ☐ <=</div> <div>Derived variable formula: ☐ > ☐ >=</div>			
# of variables:	6			

Figure 5: Editing a data mining session

The first step in creating a mining session is to name the session, and then choose an associated dataset. The name given to the session should be descriptive enough to distinguish sessions from one another, even if they operate on the same dataset. For example the description might include the name of the dataset, number of records, ratio for testing versus training and other pertinent information.

After the preliminary information is entered, the structure for the dataset must be imported, generated, or entered manually. The structure refers to the list of variables and the properties associated with each one. For information concerning the data associated with this, please see the corresponding section on page 48. The easiest method to build the basic structure is to have it generated by using the *Scan dataset* button.

When the dataset is scanned there are several steps performed for each record within the dataset according to the following algorithm:

For each variable within the current record

Is the variable in the list?

If not, add it.

Examine the variable type (ex: Integer) for compatibility

Is it a Real value, where it was previously classified as an Integer?

If yes, convert it to a Real value with the known precision.

Is the precision greater in the new value?

If yes, increase the precision accordingly.

Does the variable represent a value where it was previously classified as Boolean?

If yes, present a warning to the user.

Where applicable, examine the value represented

Does the current minimum value (lower bound) have a higher value?

If yes, adjust the lower bound.

Does the current maximum value (upper bound) have a lower value?

If yes, adjust the upper bound.

It is important to remember that when a dataset is broken into several segments used for training and testing, the variable definitions should be the same for all sets. To achieve this it is necessary to scan the entire dataset before splitting it into the various segments. Afterwards the definition can be copied or exported for use in the smaller sets. If the data available to the user is pre-split, then it can be merged, or the Scan dataset command can be run for the same list over several of the datasets. As each dataset is scanned the variable definitions will be refined.

For variables that are added using this method, the allowed operations will be set to include all except for the *not equal to* (!=) operation. If this operation is required in the set of terminals then it must be selected manually for each variable.

For variables that are not added manually, such as in the case of derived values, the *Add* button must be used to create the variable. This button is not always available, but rather only when a unique variable name is entered at the top of the list. After the identifying label is entered and the button is pressed the new variable will be added. Variables created using this method will have no default values.

A summary of the steps for adding rules manually is as follows:

- 1) Type the unique variable name into the top of the variable list, and press the Add button. The new variable will be put into the list and selected.
- 2) Select the type as being Integer, Real, Percentage, or Boolean.

- 3) Type in the English name that describes the variable.
- 4) Enter the class of variables that the current one belongs to.
- 5) Select whether the rule appears in the antecedent, consequent, or both. Selecting neither of these positions will leave the variable in the list, however it will not be used during the search process. This can also be done to temporarily omit certain variables during mining sessions.
- 6) If not a Boolean type, enter the upper and lower (minimum and maximum) values for the variable.
- 7) If the type is a Real or Percentage, enter the decimal precision.
- 8) Select the comparison operators that are allowed for this variable.
- 9) If it is a derived field then enter the program instructions to be executed.

Pressing the *Defaults* button will select all of the operations except for *not equal to*. This operation is excluded because it increases the search space and often erodes the relevance of the resulting rule set.

The problems experienced with negation operators are described in detail in the section on the Boolean negation operator. This is found in the *System Configuration*, under *Miscellaneous Options*, on page 88.

After all of the variables are entered there are a couple of other options available, including deletion of variables from the list, importing of other variables from a specified file, and exporting the current list to a file.

It is important to remember that any changes made to the mining session will not be saved unless the *Save* button is pressed. If instead the *Cancel* button is used to return to the main menu, any changes made since the last save will be discarded.

Conversion of Existing Datasets

As with other data mining and statistical packages, RAGA uses a proprietary file format to store and work with datasets. Because this format is not compatible with other packages, such as SAS or SPSS, it means that data cannot be freely interchanged.

In order to make use of existing datasets without manually converting the data, there is a facility within RAGA to import and convert records. This process is known as *Data file restructuring*.

Restructure data file

Source file:

Output file:

☒ First line of source file are the column names

☐ Column names in other file

Columns:

☐ Add incomplete records to new file

Lines in source: Errors:

Lines in output:

Status: Enter filenames and press Start

Figure 6: Restructuring a data file

Before the conversion process can take place there are a few fields that must be entered. In addition to the names of source and output files, the user must have some knowledge about the existing format. The important distinction is that some data files have the column headers (also referred to as field/attribute names) in the main file as the first record, while others may use a separate file to store this information. Both of these formats are widely used and also acceptable to the restructuring process.

By default the column headers are expected to be in the first line of the data file. If this is changed then the column header filename must be entered. A file can also be selected by using the Browse button, located directly to the right of this field.

A final option specifiable by the user is what to do with incomplete records. By default, incomplete records are simply ignored. If the *Add incomplete records to new file* button is checked, then the available fields will be recorded and the others simply left blank.

Once the names of the source and output files are entered, as well as the filename for the column headers if necessary, the Start button will become available. After the Start button is pressed the number of lines in the source file will be displayed and the restructuring will begin. While building the new data file the program cannot be interrupted, and the Exit button cannot be pressed. Once complete, information will be displayed concerning the output file status, and the Exit button can be used to return to the main menu.

The first field that will be updated is the number of lines in the output file. This value should be equal to the number of records in the dataset. It will differ by one if the first row was the column header information, and also one more for each incomplete record if they are not added to the new file. The drop-down error list will contain line numbers for records that are incomplete or contain other types of errors. These line numbers correspond directly to those in the source file.

Randomizing the dataset

Definition and rationale

To *randomize* the dataset is to reorganize the records with respect to each other. The data within each record is not changed in any way, but the ordering of records within the set will follow no particular pattern when the process is complete.

Randomization of records is important for the rule discovery process, particularly in the case of classification. Because a dataset is split up into various testing and training sets it is important that no specific ordering is present. If a pre-existing structure underlies a block of records then it will act as an unfair bias for one or more characteristics. If, for example, a dataset is being used to learn rules for fraud detection and the records are pre-sorted in the order of *Fraud detected* and *No Fraud detected*, it is possible that one of the split sets will contain records of only one class. When this occurs it means the entire discovery process is tainted because no counterexamples exist and 100% of the rules will classify for the same result.

Although some datasets might be pre-randomized it is usually not the case. When datasets are ordered it is not always obvious, and normally occurs for a variety of reasons. Records in a dataset might be ordered according to some natural scheme, or due to side effects of processes used during the data compilation phase. For example, when several cashiers submit daily sales to a central computer the resulting set may have an ordering based on particular cashier identification. Although this may not appear to be of

any importance it can still bias the results, and therefore the dataset should be randomized.

Similarly for cases where the data is compiled to generate statistical reports, the process often introduces structure to a dataset because the reports require it. Once the records have been optimized for running these tasks they are no longer suitable for splitting into testing and training sets. This is also true for data that has been previously classified by an expert. Data in this form is often sorted based on the class, and thus useless for splitting into various sets unless previously randomized.

Procedure

Once the option is selected from the menu the user is prompted for a filename. The program uses the following algorithm to create a new *randomized* file:

LineCount := Number of records in the input file

For each record in the input file

 Assign a unique random number between 1 and LineCount

Loop counter (index) from 1 to LineCount

 Find record in the input file where index = the assigned random number

 Read the record at the specified index

 Write the record to the next sequential position in the output file

After the process is complete the software will display a message for the user indicating the number of lines (records) in the output file. If the number of output lines differs from the number of input lines, then some kind of file error has occurred. Errors of this type are outside of the control of this application.

Figure 7: Randomizing the dataset

Configuring the System

When running a mining session it is important to consider the number of options available for varying and fine-tuning specific search parameters. In a standard GA there are several parameters that can be varied, and as RAGA is an expansion of this it also contains different options. This configuration screen allows for input concerning standard options, as well as those specific to the proprietary genetic engine, and finally those related to the type of search being performed.

System configuration			
Maximum number of conjunctions:		Random number generator:	
Initial population:	<input type="text" value="1"/> <input type="text" value="1"/>	☒ Default ☐ RNG #2 ☐ RNG #3	
Entire run:	<input type="text" value="8"/> <input type="text" value="3"/>	Initial seed: <input type="text" value="0"/>	
Select fitness function:		☒ Use elitism Deviation (%): <input type="text" value="10.00"/> ☐ Classification Max copy (%): <input type="text" value="10.00"/> ☒ Use classification elitism	
☒ C/S Distance SF adjustment: <input type="text" value="1.00"/> ☐ Penalty constant: <input type="text"/>		Rule fitness options: ☐ Limit expansion to good rules ☒ Encourage wider rule coverage ☒ Reward shorter rules (parsimonious) ☐ Reward longer rules (more complex) Reward constant: <input type="text" value="0.10"/>	
Termination condition:		Intergeneration processing:	
☒ # of generations <input type="text" value="50"/> ☐ Stop on full data coverage ☐ % of good rules <input type="text" value="80.00"/> Minimum fitness %: <input type="text" value="95.00"/>		☒ Eliminate extra numeric restrictions ☒ Eliminate conflicting restrictions ☒ Elimination removes more specific rules ☐ Elimination removes more general rules	
Crossover rate (%): <input type="text" value="80.00"/>		Mutation rates (%):	
Micro (min,max): <input type="text" value="0.06"/> <input type="text" value="0.75"/>		Macro (min,max): <input type="text" value="0.01"/> <input type="text" value="0.10"/>	
Other options: ☒ Pre-load data into system memory ☐ Allow Boolean negation operator			
<input type="button" value="Save"/> <input type="button" value="Cancel"/> <input type="button" value="Defaults"/> <input type="button" value="Help"/>			

Figure 8: System configuration

Rule length

One of the most important configuration options is choosing the settings required to distinguish between directed and undirected mining tasks. Because multiple options are involved in the distinction they will each be addressed throughout the following section and summarized at the end.

The first option consists of four fields that dictate the maximum length of rules that will be allowed during the search. The length of a rule is broken down into each side, namely the length of the antecedent and the length of the consequent. The length of each side is

actually the number of terms, separated by conjunctions. For example, $(A > 5) \wedge (A < 10) \wedge (B=A)$ has three terms, thus is of length three.

The upper two fields apply only to the initial population, which is the group that is randomly generated before the evolutionary process begins. The fields represent the number of terms in the antecedent and the number of terms in the consequent respectively. The lower two fields also specify the number of terms for each side of the rule, but these come into effect only after the initial population is generated.

The initial population is separate from the rest of the evolutionary process because the randomly generated rules become the basis for discovery of all rules. This is important to consider because a very poor or inflexible set of initial rules often leads to negative overall search results.

The recommended length for both antecedent and consequent during the pre-evolutionary stage is one, meaning that only one term will be considered on each side of the rule. It is obvious that a rule of this structure will not likely lead to interesting information because of its limited scope, however it will be easy for the system to evaluate and score these kinds of rules.

The main reason for this initial restriction is to force the system into creating a large number of very small rules. These small rules are evaluated and compared in the same way that more complex rules are, however it is expected that they have poor confidence

and support values. The key point is that although each rule may not correctly answer a large set of records, the system is able to readily identify good rules with respect to the others in the same set. The rules identified as being superior will be combined during the evolutionary process and will act as *building blocks* for creating larger, more complex rules.

Consider the following set of rules present in the initial population:

$If (A > 5) \wedge (B > 10) Then (C > 50) \wedge (D < 25)$	[Confidence 0.0%, Support 0.0%]
$If (A > 5) Then (C > 50) \wedge (D < 25)$	[Confidence 0.0%, Support 0.0%]
$If (A > 5) \wedge (B > 10) Then (C > 50)$	[Confidence 25.0%, Support 2.0%]
$If (A > 5) Then (C > 50)$	[Confidence 75.0%, Support 8.0%]

Through simple inspection of these rules it is easy to select the important terms, particularly *A* and the relationship it has with *C*. In the first rule the importance of this relationship was obscured by extra terms *B* and *D*, which brought the confidence of the rule to zero. Rules with no confidence are very unlikely to be selected for further evolution, and thus the important terms would be lost before the next generation.

Similarly in the second rule the consequent was corrupted by the *D* variable, which once again hides the fact that interesting terms exist within the rule. In the third rule, with the removal of the comparison with *D* in the consequent, the rule now has some confidence and support. With these values it is possible that the rule will be selected for evolution

and be positively refined in future generations, however the influence of the comparison with B could have had a more negative effect, possibly reducing the confidence to zero.

Finally, in the fourth rule, the confidence and support factors are notably good. Because this rule will almost certainly be selected for future generations it is very likely that the proper relationship between *A* and *C* will eventually be examined. In fact, it is expected that many rules similar to the third in the set will be discovered and discarded during future generations. As more terms are added to further restrict the rule it is hoped that the confidence will increase and the original rule replaced with the better one.

Essentially, the advantage to using small rules is that the important terms and comparisons will not be overlooked because of other randomly generated restrictions. Because these terms are identifiably more positive they will be selected for refinement in future generations.

Another important thing to consider when specifying the maximum length of the consequent is what type of search is being performed, particularly in the case of classifying data according to pre-defined classes. Most classification systems group data into one class only, and therefore the consequent must consist of exactly one term that describes the class. For example:

If (income < 20000) *Then* (class = lower) [correct structure]

If (income > 8500) *Then* (class = lower) [^] (class = middle) [impossible, cannot be both]

The antecedent is different because its length is a measure of how complex the rule is. In general, a rule increases in complexity by adding more constraints through the use of conjunctions. Therefore, the maximum number of conjunctions is a limiting factor that dictates how complex any particular rule can be. This is important to consider with both directed and undirected searches. As with other parameters during a mining session it is possible to perform several searches using different values.

The rules that arise from searches that allow very long antecedents may explain more complex relationships, however it is important to remember that these kinds of rules are more difficult to find. As a rule becomes more complex it is more difficult to refine it, however it does prove beneficial in many cases. Another thing to consider is that more complex rules are not guaranteed to be better, even if they more accurately describe the data. For more information on this refer to the discussion on *implementing a new rule*, found on page 17.

A final thing to consider when varying the maximum length allowed for rules is that longer rules take more time to process. Setting a very high maximum may be useful for searching out obscure rules, but the penalty may be that a solution cannot be found within a reasonable period of time. If a large percentage of the population is comprised of sizeable rules, it follows that discovering even simple rules will be delayed. As was mentioned during discussion on varying confidence and support factors, mining sessions can be performed in parallel using different machines. If the available resources are high

then it might be very useful to run sessions that allow very large rules, however in many circumstances it is best to leave these kinds of experiments to test as a last resort.

Fitness function

Presently there are two choices of fitness functions available to the genetic engine, namely *C/S Distance* and *Classification*. Both of these functions first require that the raw fitness be calculated, which is a combination of the confidence and support factors. The equation is described in detail on page 30.

The first option, *C/S Distance*, can be used for both directed and undirected mining tasks because any combination of fitness modification options can be used. The second fitness function, *classification*, is used solely for the purpose of classifying data into pre-existing classes. Although it uses the same equation as the *C/S Distance*, the target values are fixed at 100% confidence and 100% support. In addition to the fixed values, other adjustments are made to influence more complete coverage of the data set.

The main adjustment to fitness is a bonus awarded to rules that are first in the hierarchy to successfully answer a particular data item. This function is more completely described in the *Encourage wider rule coverage* option, later in this section. The only difference is that the penalty applied to successive rules answering the same data is adjustable, while it is not flexible for the same option under *C/S Distance*.

Rule fitness options

Depending on the dataset and the type of search being performed there are several fitness modifiers that can be used. A fitness modifier is an adjustment made to the rule fitness after it is calculated using the basic function.

The first option is *Limit expansion to good rules*, which is used to control the expansion of rules during the evolutionary process. When this option is not used, rules can grow in size without restriction using either crossover or mutation operations. Growth can occur in either the antecedent or the consequent, and is limited only by the maximum length as specified for each in the configuration options.

The problem with allowing rules to expand in size without restriction is that useless rules, which are labeled as such if they contribute nothing positive to the results, are also allowed to expand and become more complex. Even in the case of rules that have no confidence it is still possible that new terms will be added. This has the potential to impair the search in two ways, specifically an increased processing time as well as an increased number of bad rules. When in use, this option guarantees that each rule must not expand unless it already has a positive confidence.

Unless a rule can be expanded and transformed into something meaningful there is no point in building on it, and it will be discarded from the population. Because large rules will not exist unless they are valid, processing time is not wasted on their evaluation.

Also, because each rule has a solid base with potential for refinement, the system will produce smaller and more useful rules, which leads to a better hierarchy overall.

The next option is *Encourage wider rule coverage*, and refers to a bonus/penalty system that is in place for purposes of data classification. The goal is to reward rules based on the number of elements they correctly answer within the dataset, excluding those answered by other rules. After the confidence, support, and raw fitness is calculated for each rule the algorithm runs as follows:

Iterate over each element in the dataset:

- Find all of the rules that answer the element correctly. Call this the **confident group** - CG.

- Choose the rule in CG that has the highest fitness value.

- Select the rule with the highest fitness (which is before all others in the current hierarchy).

- This rule is awarded credit for answering the element first.

- Penalize all other rules in CG.

- Re-order the full hierarchy according to the adjusted fitness values.

The purpose of this algorithm is to group together rules that simultaneously answer the same data elements. Only the rule with the highest raw fitness is awarded the bonus for answering each element, and the remaining rules are penalized. This adjustment is applied to the raw fitness, and the rules are reorganized within the hierarchy accordingly.

As this process continues through the elements in the dataset the positions of certain rules will become more ingrained with respect to others. Rules with a very high raw fitness will continue to improve because the bonus is more likely to be awarded, and the opposite will occur for rules of lower fitness. Eventually the upper segment of the hierarchy will contain primarily good rules, and the lower will contain poor rules.

At some intermediate point within the hierarchy there will be a set of rules that are still very confident, but the fitness is quite low because of the adjustment. These rules are typically good, however were too often the losers in the competition for bonuses. It is important to recognize that these rules are unrefined, and act as stepping-stones for rules that eventually out-compete them in the hierarchy.

When considering only the top segment of the hierarchy, there is normally another recognizable pattern caused by the fitness adjustment. The pattern is a sequence of rules that alternate between different consequents, or classes. This is most obvious in cases where the rules are well refined and the elements are distributed equally among the classes defined by the expert. The reason for the staggering is that rules with similar behaviour are forced out of their positions despite their raw fitness. In effect this allows other rules to take their place, and prevents a single class from dominating the hierarchy. In GA terminology this technique is referred to as creating *niches*.

This feature is important for classification tasks because it creates an environment that maximizes coverage while minimizing rule overlap. The result is a set with as few rules

as possible, without penalizing the coverage or accuracy. An example of this can be seen in the set of rules evolved by RAGA for the polygon dataset, on page 114.

The final two fitness adjustment options are *Reward shorter rules*, and *Reward longer rules*. This is a bonus that is applied to a rule based solely on the total number of terms present in the antecedent and consequent combined.

The motivation behind using this option is to bias the system towards slightly longer or shorter rules. Because of increased readability and ease of understanding, it is common to favour shorter rules. This is similar to a reward used by Genetic Programming systems, where shorter programs are allotted credit for being *parsimonious*. Basically, the two rules or programs perform exactly the same within the current dataset, however the smaller one is presumed to be better.

The process examines rules in the set that answer the same elements and first examines their structures before selecting one to be put into the rule hierarchy. For this comparison the actual position within the rule hierarchy is ignored, and each is treated equally until the terms are more closely examined.

The amount awarded as a bonus is dependent on the actual length, as well as the *Reward constant*, which is the next field in the option list. The reward constant is an additive value that will be applied to the standard fitness as follows:

MXA := maximum number of terms allowed in the antecedent

NA := actual number of terms in the Antecedent

MXC := maximum number of terms allowed in the consequent

NC := actual number of terms in the consequent

Bonus for shorter rules := (rewardConstant * ((MXA-NA) + (MXC-MC)))

Bonus for longer rules := (rewardConstant * (NA+NC))

As seen in the algorithm above the reward constant is applied differently depending on the rule length preference. For a bias towards longer rules the credit is a simple function of the rule length, while rewarding shorter rules applies a bonus that is relative to the maximum rule length.

As with many of the other options it may be necessary to change the value of the reward constant. This value can be set to different levels depending on the importance of rule length, but some experimentation may be required for datasets of different sizes. For example, very large datasets typically have a lower fitness deviation between rules. In these cases a high reward constant may have such a large impact that the length of a rule could make it appear better than a more confident rule. To avoid this problem it is important to adjust the reward constant such that rule length separates only rules of approximately the same confidence and support.

One reason to bias the system towards smaller rules is the expectation that they will be easier to understand, as well as more efficient to process. This is often true because larger

rules are more likely to contain redundant or unimportant information, and this can obscure the real reasons for the success of the rule.

Complex rules are not only more difficult to read and analyze, but more importantly they may be unnecessarily specialized. For example, if a rule is simply a superset of terms as compared to another shorter rule, but the two rules answer exactly the same elements within the dataset, then the additional term(s) are not contributing useful information. Although it may not be significant with the current data (ie: training set) it is possible that the rule is not general enough to handle future cases (ie: testing set).

It is important to recognize that if the additional terms were useful in even a small way, there would be a change in the results and a shift in the fitness. Because this is not expressed there is little chance that additional specialization will be more effective in new cases. Without having any additional knowledge to help differentiate the two rules it is safer to keep the more general (shorter) rule.

Although much of the discussion to this point emphasizes maintaining a preference for shorter rules, there are some interesting reasons to experiment with biasing the system to the contrary. When a rule has an additional term that does not affect the result, this term is referred to as an *intron* [Levenick, 1991].

Introns have been researched in other GA systems and appear to have a beneficial effect on the process. Although the reasons for this are not completely understood, it is possible

that the advantage only exists in certain problem domains and may not be applicable to RAGA.

Termination condition

Evolutionary algorithms such as the one used in RAGA will continue indefinitely unless a termination condition is specified. For generic problems the evolution normally terminates after a preset number of generations, after which the best individuals are identified using their associated fitness values. Because RAGA has certain recognizable goals it is possible to terminate evolution based on more significant criteria. The options can also be used in conjunction with one another, where the first goal to be satisfied halts the process.

The first option is the default one, where the search stops after a pre-specified number of generations have evolved. It is recommended that this option always be used because it guarantees an upper bound on the amount of processing that will occur over the lifetime of the search. The value most often used by Genetic Programming engines is 50 generations, although this value can be much higher if required. The reason for this value, as compared to thousands of generations used in Genetic Algorithm engines, is that programs and rules such as the ones discovered by RAGA are computationally expensive. A search that consists of a very large number of generations will take a very long time to complete.

The next available option is to have the search automatically halt when it is determined that all of the elements in the dataset are being correctly answered. Although this condition has no bearing in an undirected search it is ideal for classification systems. With the goal of a classification being that all data should be accurately labeled, the halting once this is achieved is a very good way to speed up the process. The only disadvantage observed so far is that occasionally there are a few rules that can be further refined.

The reason for this is that the system will start off with a certain percentage of the dataset covered, and gradually the original rules will become more refined. At some point the coverage will reach a plateau and the system will generate new information in an attempt to discover the missing elements. If there is a new rule created that suddenly answers the missing data then the process will stop at that point. Unfortunately the new rule is probably not as efficient as it would be had it remained within the system for several more generations.

One solution to this is to restart the search and force it to run for a small number of generations (ex: 10) without any other terminating conditions. Rather than allow the initial population to be generated randomly, simply use the results from the previous search. Instead of performing an extensive exploratory search the system will only refine the imported rules and try to evolve a more efficient set using them as the basis.

Finally it is possible to have the evolution halt based on achieving a specified target value for the number of *good* rules in the current set. A good rule in this context is one that has fitness greater than or equal to the value entered into the *minimum fitness* field. This fitness will vary based on the dataset, however 95.0% is often considered to be good if true over at least 80% of the rules. This option would be suitable for undirected searches, but not recommended for classification because of the requirement for accuracy and as much refinement as possible.

Random number generator

A crucial part of any Genetic Algorithm is the pseudo random number generator. The reason for the importance is that the initial population generation and all of the evolutionary functions rely on probabilistic selection techniques. Each of these functions relies heavily on the fact that the numbers are random.

Because of its extensive use there are reasons that one particular algorithm may be better than another, especially depending on the data set. Examples of this are that some generators are provably more random than others, while others are faster but have a more predictable sequence. Selection of one algorithm over another may be due to personal preference or the requirement for the system to generate numbers as fast as possible.

The first random number generator is defined as the default one used by the program compiler. The second and third options are not implemented at this time, however they

can be done so with generating algorithms such as linear congruential [Lehmer (1951)] or multiple recursive [L'Ecuyer, Blouin, Couture (1993)].

The *initial seed* is a value that is basically a *starting point* for the random sequence. Each time the same search is performed the results will be identical unless this number is changed. Leaving the number the same may be useful for the purposes of reproducing results during experimentation, however it is important to change the seed between runs when searching the same dataset.

Elitism

Because Genetic Algorithm based systems are non-deterministic, there is no explicit guarantee that good members remain within the population after each generation. For RAGA, this means that although there is a high probability that good rules will continue to thrive in future generations, it is not impossible that one or more are lost and not rediscovered.

The loss of good rules between generations means that the average or maximum fitness can actually decrease during evolution, which is essentially taking a step backwards in terms of developing a hierarchy. Particularly in the case of classification, where the goal is to discover a set of rules that achieve perfect coverage, losing a good rule is a serious setback. To avoid this potential problem it is necessary to use elitism.

Elitism is a process that guarantees certain rules are copied directly into the new generation without having to be probabilistically selected. These copies are not subject to crossover or mutation, and thus will appear in exactly the same form. Because the rules are at the top of the fitness scale, and they are copied without modification, it is impossible for the maximum fitness to decrease between generations. The calculation for average fitness depends on all of the rules in the set, whether considered elite or not, so it is still possible for this value to decrease between generations.

There are several versions of elitism that are available for use in Genetic Algorithms, however all of the standard forms appeared to be too limited in nature. The main reason for having to re-think elitism was that other forms only preserve the top N members according to fitness, where N is normally one. While some systems benefit from copying these groups of highly fit individuals between generations, RAGA gained very little because of the hierarchical rule structure.

When evolving a hierarchy there are several groups of rules that work together to answer sets of elements. These are also referred to as niches, and because each niche caters to different elements they are not in direct competition with each other. This characteristic made it necessary to expand elitism to ensure that a variable number of rules can be copied to the next generation. This is done in cases where it is necessary that specific rules be copied in order to preserve the current level of confidence and support.

There are two types of elitism that are available to the genetic engine. The first type is the standard elitism, where the members with highest fitness are directly copied into the next generation. This step is done after all of the fitness values have been determined, but prior to selection of any members for the next generation.

When the process starts it records the fitness for the best rule and uses this as a guideline for selecting other elite rules. The *deviation percentage* dictates how much lower another rule's fitness can be while still being considered elite. For example, if the best rule has a fitness of 81%, then a 10% deviance would allow all rules with 71% or above to be considered elite.

Particularly early in the life of the search process the rule set has not had sufficient time to evolve, and therefore it is common to find very low fitness rules at the top of the hierarchy. When poor rules are in this position, combined with an appropriate value for deviance, it means that almost the entire hierarchy can be considered elite. In order to prevent the entire population from being copied there is an upper bound parameter named the *Max copy percentage*. This value is typically low, perhaps 10%, and stipulates the maximum percentage of the population that can be copied using elitism. This is necessary because preserving too much of the population will not allow new members to be created and thus curb the effects of evolution.

The second type of elitism available for use in RAGA is *classification elitism*. This approach has the same goal as standard elitism in that it preserves important members

between generations. The difference between these two methods is defined by what are considered to be important traits in a particular rule.

Classification elitism does not make use of the standard fitness, but rather the basic result of applying the rule. To be considered elite, the rule must correctly answer at least one data element that has not been answered by a rule preceding it in the hierarchy. The reason for this is that there is something unique about the rule, and this feature is important enough that it should be preserved for future refinement.

At first glance it might seem that this form of elitism has a serious disadvantage, where very poor rules are copied into the next generation simply because they alone answer a particular element. Despite the fact that these rules are considered poor in general, it is important to determine what key features allow them to uniquely answer certain elements. These features can either be basic components for general cases, or specific logic that answers outliers or other elements that go against the norm.

Regardless of what is unique about each rule preserved, elitism guarantees that future generations will have a chance to extract what is important. If in a future generation these rules are evolved into better ones, then the original stepping-stone will be discarded, and the hierarchy will have improved.

Unlike the first version described, classification elitism has no upper bound on the number of rules that are copied. The rationale is that no matter how small the confidence

or support is, each unique feature must be fully explored. One foreseeable problem is that the possibility exists for every rule within the set to be copied without modification. In this case the population would remain static between generations and evolution would not occur, meaning that no improvement is possible.

Fortunately this problem is eliminated by RAGA through the use of a flexible population size between generations. By temporarily increasing the maximum number of rules, the system introduces new terms and logic during each generation. This new genetic material is evaluated and given the opportunity to compete with existing rules, which allows the elite members to be replaced by improved versions of themselves.

As the name suggests, classification elitism is used for obtaining the highest possible data coverage when classifying elements into one of several predefined classes. This approach is not recommended for undirected mining tasks because the level of data coverage is of no importance when searching for interesting and obscure rules. The information being sought is usually a very small set of rules that describe a small subset within the full set, and although several areas within the set are explored in parallel, there is no guarantee of significant coverage.

Crossover and mutation rates

Crossover and mutation rates dictate how often the respective operations will be applied during the evolutionary stage in the search process. For a complete description on the use and theory of these genetic operators, please see the *Implementation* section on page 32.

The first two fields are the minimum and maximum values for the percentage that indicates how likely micro-mutation will occur for each term within each rule during the final stage of evolution. The minimum value is the starting point, and throughout the lifespan of the search process it can change with an upper restriction of the maximum value.

Similarly, macro-mutation has two fields that specify a range of values that can change between generations. The minimum value is the lower bound, while the maximum value is the upper bound for the percentage of a macro-mutation occurring on each rule.

If a static crossover or mutation rate is to be used for a particular mining session, then both values in the range should be equal to the desired rate. The mechanism used to vary the rates will not alter the value.

Intergenerational processing

Intergenerational processing refers to a set of constraints that are applied to each rule during the evolutionary process. The purpose of using these options is to conserve system resources by non-evolutionarily removing rules that are deemed invalid or ineffectual.

For a full description please refer to page 40.

The first two options for this component are *Eliminate extra numeric restrictions*, and *Eliminate conflicting restrictions*. The first refers to the removal of redundant constraints

and tautologies, while the second refers to the removal of conflicting constraints and contradictions. Neither, one, or both of these options can be selected during any search session. The behaviour exhibited by these options depends on the next choice, which is an indication of preference by the user.

When a rule is deemed to require modification it means that at least one term must be removed from rule. The choice of terms to be removed is important because it often affects the specificity of the rule. For example, removing the more restrictive of the competing terms will create a more general rule, while removing the less restrictive term will make it more specific. The options that control this choice are *Elimination removes more specific rules*, and *Elimination removes more general rules*.

Only one of these options can be specified and will be used by RAGA to determine which term(s) will be removed from rules that do not meet the requirements as specified by the first two options.

Miscellaneous options

The two final options in the system configuration are not directly related to the genetic engine. The first is a low level option that improves system performance, and the second is a restriction placed on the use of certain rules.

The operating speed for searching with RAGA can be greatly improved by using the *Pre-load data into system memory* option. When selected, RAGA loads the entire dataset into

memory for the duration of the search process, effectively eliminating the need for disk access. If the system memory is large enough to hold all of the data, then there is a substantial increase in speed. This option is always recommended, however cannot be used for very large datasets on some machines.

The last option concerns data mining and the use of a Boolean negation operator. When selected, this operator is included in the set of available terminals for use when creating and evolving rules.

The Boolean negation operator is a modifier for Boolean variables that indicates a particular attribute is not present within the record. This is similar to having a *not equal to* operator for Integer, Real, or Percentage variable types.

Negated Boolean values are not often used because of the increase in search space created by them. Another problem is that there is typically a decrease in the quality of rules generated while negation is allowed. The primary reason for this is that there are many more items not present in certain types of transactions. For example, when considering items purchased from a grocery store there will be perhaps 10 different items on the bill from a choice of over 50000. A rule that specifies a set of items not be in the transaction will be very confident in most cases, but probably not reveal any interesting information. An example of this is:

If ($\sim A15$) $\wedge$ ($\sim B992$) $\wedge$ ($\sim X0091$) Then ($Y \neq 5$)

Regardless of the frequency in which ($Y = 5$) occurs, the three items in the antecedent occur so rarely that the rule is almost always true. Unfortunately in this particular instance there is no information gained whatsoever.

In practice there are certain cases where rather than search for general inclusion rules, the user needs to search for something specific that is not present. In these cases the *Boolean negation* and *not equal to* operators may be useful, but this is very much dependent on the dataset and domain being examined. For example, no benefit will be realized unless most attributes are expected to be present within each element.

Running a Mining Session

The screenshot shows a window titled "Mining session: test.data". At the top, there are input fields for "Population size" (set to 100), "Confidence factor" (set to 100), and "Support factor" (set to 100). A "Progress:" bar is located to the right of these fields. Below the input fields, there is a section labeled "Current list of rules:" with buttons for "Import", "Add", and "Remove". A large empty rectangular area is below this section. To the right of this area, there are several labels with corresponding input fields: "Last generation:", "Good rules:", "Best fitness:", "Worst fitness:", "Average fitness:", "Perfect coverage:", and "Minimum coverage:". At the bottom left, there is a "Session history" label and a checkbox labeled "Suppress population display" which is checked. On the bottom right, there are four buttons: "Start", "Configure", "Go back", and a partially visible "End" button.

Figure 9: Searching the data

Before starting a search of the data there is some information that must be entered. The initial fields are the population size, confidence target and support target values. These are set according to the type and variation of search being done.

The size of the population is not only an upper bound on the number of rules that will be generated, but more importantly it dictates how much logic can be considered for each rule during the evolutionary process. In practice the size of the population is much larger than the number of rules being sought, although this is not always the case. Because it is difficult to estimate the number of useful rules that could be produced during a mining session, there is no way to know beforehand what the optimum population size should be. Typically a starting value such as 100 or 250 rules is used, and then varied for subsequent runs if necessary.

Varying the population size between runs can be beneficial, however the best approach found during experimentation was to maximize the population size. The limiting factor on increasing the size is that the process takes much longer, so the population should be increased only as much as is practical considering the time frame.

The basis for using a larger population is that more genetic material will be available for selection during evolution. Additionally, a larger population has the potential for a greater overall diversity, which is also beneficial for the selection process. Because small

populations lack these traits it means that they simply cannot cover as much of the search space, and thus will not produce robust rules.

Using a very high population size allows the system to more fully explore the search space, but greatly taxes the system resources. The ideal size is typically found through trial and error, where the size is first set very high. After a couple of generations the ETC (estimated time of completion) is checked, and a decision as to whether to continue is made. If the process is expected to take longer than time is available, the population size should be lowered and the process restarted.

The ETC is dependent on the number of rules in the population, as well as the number of elements in the dataset, and finally the nature of the dataset. For searches with very long rules the complexity could be quite high, and it is suggested that several mining sessions be run in parallel on different machines. This configuration requires that the same dataset be referenced in either a common storage, or through identical copies on each workstation.

The settings for both confidence and support factors differ depending on a number of factors and are described in more detail on page 13 (Confidence and Support Factors). The key thing to consider is that these values combine to determine the fitness of a rule being evaluated, and thus guide the search process.

After setting the initial values it is possible to further change how the genetic engine will operate. This feature is displayed when the *Configuration* button is pressed, and is referred to as the *Configuring the System*. Details on this screen can be found in the corresponding section on page 66.

Before the search begins the *Current list of rules* will be empty, but during and after the search it will contain a list of the current rule population. It is possible to add rules to this list before the search begins, which is referred to as *seeding* the population. By doing this it will mean that these rules start in the first generation, and fewer rules will be generated at random. This is useful particularly in cases where previous searches have yielded rules that need refinement.

Seeding the population with pre-existing or non-random rules can be done using two different methods, namely *importing* or *manually adding* rules. Pressing the *Import* button will prompt the user for a file containing previously evolved rules. This file is of the same format generated by RAGA when the *Save* button is pressed to save the current set. After selecting a file the contents will be imported into the current list of rules. If there are rules contained within the file that do not meet the currently specified restrictions, such as length of antecedent or consequent, they will be discarded automatically. After the import is complete the user will be informed as to how many rules were imported and discarded.

The other method used to insert a non-random rule is to press the *Add* button, which brings up a screen used to add rules manually. This screen must be visited one time for each rule to be added.

The screenshot shows a window titled "Add new rule manually" with a close button (X) in the top right corner. Inside the window, there are three input fields: "Left side label", "Operation", and "Right side label or constant". The "Left side label" field is currently open, displaying a list of attributes: "colour", "length", "width", "depth", "shape", and "class". To the right of these fields are two buttons: "Antecedent" and "Consequent". At the bottom of the window, there are two buttons: "Add Rule" and "Cancel".

Figure 10: Manual rule addition

Once in the section used to add rules manually there are a few controls used to select the specifics of the rule. The first field is a drop down box that contains a list of all the attributes available for use on the left hand side of the rule. After selecting the proper variable, the same procedure is used to choose both right side variable and the operator used to compare the two sides. If a constant value is part of the rule then it is used in place of the right side label. This effectively defines a single term within the rule.

After the first term is defined, it is inserted into the rule by pressing the either the *Antecedent* button, or the *Consequent* button. After doing this the term will be put into the corresponding side and cannot be edited. If an incorrect term is added accidentally then pressing the Cancel button will return the user to the previous screen, and the current

rule will be discarded. Subsequent terms added to the same side of the equation will be separated by conjunctions. This process is repeated until the rule is complete.

If the final rule is correct then pressing the *Add rule* button will return the user to the previous screen and add it to the active list. At this time it is also verified against the constraints and restrictions as defined in the system configuration. Rules that do not conform to the set standards will not be added.

After adding rules manually or importing from an existing file the list of rules can be reviewed before pressing the *Start* button. If there is an undesired rule in the list, it can be removed by selecting it and pressing the *Remove* button. Similarly, pressing the *Clear* button can clear the entire list.

Once the search is started the list of rules will be filled to the specified population size, and will be updated after each new generation is evolved. There are also several other fields updated after each generation as follows:

Progress bar: this bar indicates how much of the current generation has been evolved, which provides a quick way of estimating when the current list of rules will be updated with those from the next generation.

Last generation: this value indicates the number of the last generation to evolved, whose rules appear in the active list.

Good rules; this value indicates the percentage of rules considered to be *good*. In this context, a *good* rule is one that is above the level of fitness specified in the system configuration.

The values for *best fitness*, *worst fitness*, and *average fitness* describe the fitness value for each of the corresponding rules in the current population.

The values for *minimum coverage* and *perfect coverage* concern the percentage of data elements that are correctly answered by the current set of rules. As discussed earlier, this measure is important for classification tasks, and is used to determine how much of the entire dataset is being properly classified.

Perfect coverage indicates the percentage of rules that satisfy the two conditions required for a rule to be considered perfect. The conditions are that the confidence be 100%, and that the elements being answered have no overlap with those answered by more senior rules within the hierarchy. The value for support does not affect this measure, however at least one element must be answered correctly and uniquely to satisfy the two conditions.

The measure of minimum coverage is not as strict, however it is also not 100% accurate at all times. The minimum coverage is essentially an estimate of the minimum number of elements that are being correctly answered by the current set of rules. This is not as strict as perfect coverage because overlap with previous answers is not considered. The reason

for the possibly inaccuracy is that the variation in population size between generations can eliminate some data coverage. Because of this it is not possible to determine the absolute coverage until the evolution is complete.

The last field that is updated after each generation is the *Session History*. This stores a log the initial parameters and configuration for the search, as well as details for each generation. Optionally, as specified by the user with the button labeled *Suppress Population Display*, this list will also contain a snapshot of the entire population at the end of each generation. RAGA does not keep track of extinct rules that have been removed from the population, but occasionally this information is of interest. In order to track changes between generations this history must be saved and examined after the search is complete. Control for saving and clearing the history are also available.

After the search session is complete, both the final set of rules and the session history can be browsed. An extra control labeled *Translate* is available for both fields, and is also performed when a rule from either list is double-clicked with the right mouse button. This will parse the terms within the rule and created an English sentence according to the translations specified during the defining of the session.

There are also controls available for saving and clearing the final hierarchy. Saving the final set of rules is just about always done because otherwise the effort devoted to the evolution of the list will be lost.

Saving the list of rules also allows it to be imported later for either refinement or testing. Testing the rules against other datasets is a crucial part of data mining, whether directed or undirected. For a description of using separate testing and training sets, refer to the section on page 4.

Testing the rule hierarchy

The process of testing rules in RAGA is trivial once the proper mining sessions have been defined, and rules are in the active list. Pressing the *Test* button will test the current rule set and report the results after completing. The progress bar is updated throughout the testing procedure to provide an estimate of how long it will take.

The usual method for testing is to first evolve a rule hierarchy using the training set. After this is complete, the list of rules should be saved, and then the current mining session closed. The complimentary session that references the testing data is then opened. This session is compatible with the training session, however it is comprised of unseen data from the same source.

When the user returns to the searching/testing screen the rules must be reloaded into the active list. Once this is complete the hierarchy can be tested against the proper data, and the results compared to those achieved against the training data.

The information presented to the user after testing includes the number of elements correctly and incorrectly answered, and the percentage of data coverage. Also included in

the session history is a list of all elements that were addressed, and which rules had matching antecedents. If a single element is matched by several rules, then each rule will be displayed until the element is correctly answered.

Examining this information is an important part of understanding the rule hierarchy. In some cases the user might add, omit or change the order of some rules in an attempt to better represent the data overall. Experiments such as this can afterwards be tested on the training set, testing set, or the complete set of data.

After evolving, testing, and refining the rule hierarchy it is also possible to view it using the *Results Viewer*. This component is described in the next section.

Viewing Results

After creating one or more rule hierarchies it is possible to view them, or perform one or more post-processing functions using this component.

Figure 11: Viewing results

When this screen is first shown there are no rules in the list. Pressing the *Browse* button, and then selecting the appropriate file when prompted, imports a list of evolved rules. Each rule will be displayed with the confidence, support, fitness, and selection adjustment values as calculated from the final generation during the evolution of the hierarchy.

Individually rules can be selected and either *translated* or *deleted*. Double-clicking the right mouse button on a specific rule will automatically translate it. Translation redisplay the rule in an easy-to-read form based on the English names that are associated with the attributes in the currently open session. Deleting a rule simply removes it from the list.

When one or more rules are re-arranged, modified, deleted from the list it is not a permanent operation. Because the list is stored in local memory, the changes are only committed to disk once the *Save* button is pressed. Pressing this button will prompt the user for a new file, at which time either the original file can be replaced, or a new one specified. Keeping the original file is always recommended because it adds both rollback and traceability to the process.

In addition to operating on individual rules there are group functions that can be performed. The first of these functions is called *Filter rules*, and is used to strip the list of rules that do not meet certain criteria. Once the button is pressed the following screen will appear:

	Minimum	Maximum
Confidence factor:	75	100
Support factor:	1	100
Fitness:	0	100
Selection adjustment:	0.001	100

Figure 12: Filtering results

The user is able to specify both minimum and maximum boundary conditions for each of confidence, support, fitness, and selection adjustment. When the *Filter* button is pressed,

the constraints will be checked for each rule in the entire set. Rules that do not meet the specified requirements will be dropped from the list, and when the control returns to the previous screen, only the subset of rules will be shown.

The second group function available is called *Merge rules*, and does not strictly modify the rules, but rather merges two sets of rules together into a single list. This process starts by pressing the *Merge* button after there is already a list of rules being displayed. When prompted, the user must specify a second file containing rules, and the system will immediately try to insert each of the new rules into the list. Once complete, this process can be repeated and several files can be merged into the current list.

Although duplicate rules will not be inserted into the list, the other restrictions associated with mining sessions do not apply. An example of this is that the allowed lengths for both antecedent and consequent cannot be used to exclude any rule from the new list.

After creating a superset of the original rules there are a number of possible actions to consider. The user may decide to use the new set to seed a brand new search session. This makes it possible to run several searches, combine the results, and finally refine the entire hierarchy at once. Another possibility is to simply test the combined correctness and coverage of several smaller rule sets at once time. The advantage to this over running each separately is that duplicates will automatically be removed and the true data coverage will not be estimated.

The final group function available is called *Sort by Confidence Factor*. This function sorts the entire set of rules in descending order according to the confidence factor for each. The result is a list of rules where the top of the hierarchy is more likely to answer the majority of elements. Rules that answer contradictions and exceptions to other rules will be at the bottom of the list.

Although rule sets are normally evolved in this fashion it is often the case, especially in classification tasks, that the rules are not sorted strictly by confidence. Because both support and the selection adjustment contribute to the fitness for each rule, the confidence is only one of the factors considered for the position within the hierarchy.

It is also important to remember that when merging files, the incoming rules are placed at the bottom of the list and sorting by confidence is recommended before testing the combined sets.

To summarize, the following steps are recommended when merging several sets of rules together for testing or refinement:

1. Browse for the file containing the first set of rules.
2. Merge each additional file into the current set.
3. Filter the rules such that the only restriction is the selection adjustment with a very low value (ex: 0.001). This step is used to trim the dataset of rules that

played no significant role (concerning data coverage) within their original hierarchy.

4. Sort the entire set by confidence.
5. Save the new rule set under a new name.

Generating synthetic datasets

Testing the results obtained from undirected searches is non-trivial, especially for non-experts in the domain being examined. Because of the effort required in testing these rules it is difficult to test undirected data mining algorithms on real datasets.

One way of addressing this problem is to generate a set of random data that conforms to a pre-specified set of known rules. The data can optionally contain noise, incorrect and missing values, as well as non-essential attributes.

Example simple rule set

If $(A > 5) \wedge (B < 20)$ Then $(C > 8) \wedge (C < 12)$

If $(C < 4)$ Then $(B > 15)$

If $(C > 10)$ Then $(A > 4)$

Based on these rules a large set of data can be generated, and this set can be used to test undirected data mining algorithms.

The ultimate goal for an undirected search algorithm is to find the exact set of rules as shown above, however this is only one of many possible outcomes. Assuming that several rules are generated with reasonable accuracy they do not necessarily have to match those used to define the set. Several factors that can influence the search results are erroneous or inaccurate values, an uneven distribution with respect to the defining rules, and the number of elements generated. This last factor is quite important because too little data often means less coverage of the rule space is considered. Results from this type of search often include rules that are too specific and do not have the proper level of support when applied to a superset of the data.

Results

RAGA has been tested using a number of different datasets for both directed and undirected mining tasks. Some of these datasets comes from data mining and machine learning repositories, while others are generated synthetically for the purpose of algorithm testing. This section includes the information and results of these tests.

Classification of the Polygon Dataset

Background

The polygon dataset describes a group of geometric shapes, and was designed to demonstrate the limitations in using classifiers with 1-place predicates.

The dataset consists of 190 differently sized shapes including triangles and rectangles. Each of these polygons is defined by only four attributes (L1-L4), which correspond to the length of the four different sides. The L1 attribute indicates the length of bottom side, while L2-L4 define the remaining sides in clockwise order from the bottom.

One additional attribute, the *type*, is non-predictive and is used to train classifiers. This type can be one of five values as follows: equilateral triangle, isosceles triangle, other triangle, square, and rectangle.

The underlying rules for the polygons in the dataset are:

Class	Name	Rules for inclusion
1	Equilateral triangle	Any three sides equal, and the fourth has length zero.
2	Isosceles triangle	Exactly two sides equal, one non-zero, and one zero.
3	Other triangle	Three non-zero and non-equal sides, one side zero.
4	Square	All four sides equal.
5	Rectangle	Two pairs of opposite sides equal.

All of the shapes in the dataset are fully connected and enclose an area using at least three sides. For triangles, the geometric rule $BC < AC + AB$ is always true, which guarantees that the shape is valid. No polygons are defined using more than one zero-length side, and no noise is present in the data.

For more details on the specifics of this dataset, please refer to *Appendix B* on page 130.

Three different systems were used to generate classifiers for the polygon dataset. These are *C5.0 (See-5)*, *R*, and *RAGA*. Common training and testing sets were used for all three applications in two different tests. In the first test, the training set consists of 190 elements and the same data is used for testing. The second test uses an additional set of 100000 randomly generated legal polygons, and is used to measure the scalability of the rules generated in the first test.

Results using C5.0 (See-5)

When this application was used to generate the decision tree below, several of the software-specific options were tried. These options include the creation of *rulesets*, and several variations of *boosting*, however the results were not as good as the standard configuration and are not shown.

Output

```
Read 190 cases (4 attributes) from Polygon.data
```

```
Decision tree:
```

```
L4 > 4:
: ...L2 <= 3:
:   : ...L4 <= 333:
:   :   : ...L4 <= 75: Equilateral triangle (3)
:   :   :   : L4 > 75:
:   :   :   :   : ...L1 <= 125: Isosceles Triangle (5)
:   :   :   :   :   : L1 > 125:
:   :   :   :   :   :   : ...L3 <= 125: Isosceles Triangle (3)
:   :   :   :   :   :   :   : L3 > 125: Equilateral triangle (2)
:   :   :   : L4 > 333:
:   :   :   :   : ...L3 > 991: Isosceles Triangle (3)
:   :   :   :   :   : L3 <= 991:
:   :   :   :   :   :   : ...L1 <= 501: Other triangle (4)
:   :   :   :   :   :   :   : L1 > 501:
:   :   :   :   :   :   :   :   : ...L4 <= 501: Other triangle (2)
:   :   :   :   :   :   :   :   :   : L4 > 501: Equilateral triangle (3)
:   L2 > 3:
:   : ...L3 <= 0:
:   :   : ...L4 <= 333: Equilateral triangle (10/5)
:   :   :   : L4 > 333:
:   :   :   :   : ...L2 > 991: Isosceles Triangle (3)
```

```

:      :      L2 <= 991:
:      :      :...L1 <= 501: Other triangle (4)
:      :      L1 > 501:
:      :      :...L4 <= 501: Other triangle (2)
:      :      L4 > 501: Equilateral triangle (3)
:      L3 > 0:
:      :...L1 > 0: Square (42/20)
:      L1 <= 0:
:      :...L2 <= 333:
:      :      :...L4 <= 75: Equilateral triangle (3)
:      :      L4 > 75:
:      :      :...L2 <= 125: Isosceles Triangle (4)
:      :      L2 > 125:
:      :      :...L3 <= 125: Isosceles Triangle (2)
:      :      L3 > 125: Equilateral triangle (2)
:      L2 > 333:
:      :...L3 > 991: Isosceles Triangle (2)
:      L3 <= 991:
:      :...L2 <= 501: Other triangle (4)
:      L2 > 501:
:      :...L4 <= 501: Other triangle (2)
:      L4 > 501: Equilateral triangle (3)
L4 <= 4:
:...L4 > 1:
:      :...L1 > 4: Rectangle (2)
:      L1 <= 4:
:      :...L1 <= 1:
:      :      :...L1 > 0: Isosceles Triangle (4/1)
:      :      L1 <= 0:
:      :      :...L2 <= 1: Isosceles Triangle (2)
:      :      L2 > 1: Other triangle (7/1)
:      L1 > 1:

```



```

:      :...L3 <= 1: Other triangle (7/1)
:
:      L3 > 1:
:
:      :...L2 <= 1: Other triangle (6)
:
:      L2 > 1: Square (2)
L4 <= 1:
:...L4 > 0:
:
:      :...L1 <= 1:
:
:      :...L2 <= 1: Equilateral triangle (4/1)
:
:      :      L2 > 1: Isosceles Triangle (2)
:
:      L1 > 1:
:
:      :...L2 > 1: Isosceles Triangle (3)
:
:      L2 <= 1:
:
:      :...L2 <= 0: Isosceles Triangle (3)
:
:      L2 > 0: Rectangle (2)
L4 <= 0:
:...L2 > 991: Isosceles Triangle (3)
:
:      L2 <= 991:
:
:      :...L1 > 333:
:
:      :...L1 <= 501: Other triangle (4)
:
:      L1 > 501:
:
:      :...L3 <= 501: Other triangle (2)
:
:      L3 > 501: Equilateral triangle (3)
L1 <= 333:
:
:      :...L2 <= 4:
:
:      :...L2 <= 1: Isosceles Triangle (3/1)
:
:      L2 > 1:
:
:      :...L1 > 2: Other triangle (4)
:
:      L1 <= 2:
:
:      :...L2 <= 2: Isosceles Triangle (2)
:
:      L2 > 2: Other triangle (2)
L2 > 4:
:
:      :...L2 <= 75: Equilateral triangle (3)

```

```

L2 > 75:
:...L1 <= 125: Isosceles Triangle (5)
    L1 > 125:
        :...L3 <= 125: Isosceles Triangle (2)
            L3 > 125: Equilateral triangle (2)

```

Evaluation on training data (190 cases):

```

Decision Tree
-----
Size      Errors
   46    30 (15.8%)  <<

a      b      c      d      e      <-classified as
-----
35      1                      (a): class Equilateral triangle
 5     49      2              (b): class Isosceles Triangle
                        48      (c): class Other triangle
 1                      24      (d): class Square
                        1      20      4      (e): class Rectangle

```

Analysis

The predictive accuracy of the data when testing against the full training set is 84.2% using a decision tree of 46 levels.

Scalability

At several points within the tree, comparisons are made between different attributes and the constant zero. This is important because it determines the existence of particular side, and is general enough to handle unseen data. The problem is that this knowledge is not sufficient to accurately classify all of the data, and thus is not a general solution.

The result of running the 100000 randomly generated legal polygons is 39.37% correct, and 60.63% misclassified.

Results using R

Similarly to See-5, this application was used to generate a decision tree using the default configuration.

Output

Figure 13: Decision tree output from R for classification of polygons

Number of terminal nodes: 17

Residual mean deviance: 1.728 = 298.9 / 173

Misclassification error rate: 0.4579 = 87 / 190

Legend

X_N = Length of side N

Eq1 = Equilateral triangle

Iso = Isosceles triangle

Tri = Other triangle

Sqr = Square

Rect = Rectangle (Not present because R was unable to distinguish them)

Analysis

As reported by R, there are 17 terminal nodes used by the decision tree to achieve a predictive accuracy of 54.21%. This is a higher error rate than See-5, although the smaller tree size might indicate that the emphasis was placed on creating a more general rule set.

Scalability

At some point within the tree, comparisons are made to the constant 0.5. In some circumstances this was able to accurately distinguish between a square and an equilateral triangle, which share the property of having all equal sides. Because the lengths are all specified using integer values, the comparison was equivalent to checking for the constant zero, which represents the ability to check for the existence of a particular side.

As with See-5, the decision tree relies on constant values and is not general enough to accurately classify every possible polygon.

The result of running the 100000 randomly generated legal polygons is 30.27% correct, and 69.73% misclassified.

Results using RAGA

The options used within RAGA are the defaults, as specified in the configuration for classification tasks.

Output

[C100.00%,S13.16%]: If $(L2 = L3) \wedge (L4 = L1)$ then (Class = 4)

[C100.00%,S7.37%]: If $(L3 > L1) \wedge (L2 < L4) \wedge (L4 = L3)$ then (Class = 2)

[C100.00%,S6.84%]: If $(L3 \geq L1) \wedge (L4 = L2) \wedge (L2 < L1)$ then (Class = 5)

[C100.00%,S6.32%]: If $(L3 = L1) \wedge (L2 > L3)$ then (Class = 5)

[C100.00%,S6.84%]: If $(L4 < L2) \wedge (L2 = L1) \wedge (L1 > L3)$ then (Class = 2)

[C100.00%,S6.32%]: If $(L3 < L1) \wedge (L2 < L1) \wedge (L4 < L1)$ then (Class = 3)

[C100.00%,S6.32%]: If $(L4 > L3) \wedge (L2 < L4) \wedge (L4 > L1)$ then (Class = 3)

[C100.00%,S6.32%]: If $(L1 < L2) \wedge (L4 < L2) \wedge (L2 > L3)$ then (Class = 3)

[C100.00%,S6.32%]: If $(L3 > L1) \wedge (L3 > L2) \wedge (L4 < L3)$ then (Class = 3)

[C100.00%,S4.74%]: If $(L3 > L1) \wedge (L2 = L3) \wedge (L4 < L2)$ then (Class = 2)

[C100.00%,S4.74%]: If $(L2 = L3) \wedge (L4 > L1) \wedge (L4 = L3)$ then (Class = 1)

[C100.00%,S4.74%]: If $(L2 < L4) \wedge (L4 = L1) \wedge (L1 \leq L3)$ then (Class = 1)

[C100.00%,S4.74%]: If $(L4 < L2) \wedge (L2 \leq L1) \wedge (L2 = L3)$ then (Class = 1)

[C100.00%,S4.74%]: If $(L1 \geq L4) \wedge (L4 > L3) \wedge (L3 < L1) \wedge (L4 = L2)$ then (Class = 1)

[C100.00%,S3.16%]: If $(L2 < L4) \wedge (L4 = L1) \wedge (L4 > L3) \wedge (L3 < L1)$ then (Class = 2)

[C100.00%,S2.63%]: If $(L2 < L4) \wedge (L1 = L3) \wedge (L4 < L1) \wedge (L2 \leq L1) \wedge (L2 < L3)$ then (Class = 2)

[C100.00%,S1.58%]: If $(L1 < L3) \wedge (L4 \geq L3) \wedge (L3 < L2) \wedge (L4 = L2)$ then (Class = 2)

[C100.00%,S1.05%]: If $(L4 < L2) \wedge (L2 \leq L1) \wedge (L1 = L3) \wedge (L2 < L3)$ then (Class = 2)

[C100.00%,S2.11%]: If $(L3 < L1) \wedge (L1 < L4) \wedge (L2 = L4)$ then (Class = 2)

Number of rules: 19

Coverage: 100.0%

Misclassification error rate: 0.0%

Analysis

A total of 19 rules are required in order to achieve 100% predictive accuracy over all of the elements. A default class was not used, however the individual rules address enough subsets within the data to account for 100% coverage.

Scalability

As expected, RAGA evolved a set of rules that considers the relationships between attributes, rather than the values of the attributes themselves. In early stages of the evolution, RAGA developed and tested rules such as:

If $(L2 > 1248)$ Then (class = 2)

Due to the brittleness of these rules, they were not able to achieve 100% accuracy and were discarded by RAGA. At the conclusion of the search, only general rules remained.

The result of running the 100000 randomly generated legal polygons is 93.09% correct. and 6.91% misclassified.

Other notes

For this dataset. RAGA did not take advantage of the default hierarchy. Additionally, the test for the constant zero was not explicit, but rather a comparison to other variables. This may not be as efficient, however it is equally accurate in the training cases.

Classification with additional relationship attributes

Although See-5 is incapable of determining what inter-attribute relationships would be useful, an experiment was done to see how well the classification would perform if the functionality did exist. To achieve this, six derived variables were added to the list of useable attributes, as follows:

$R_{L1L2} := L1 / L2.$	Relationship between sides 1 & 2
$R_{L1L3} := L1 / L3.$	Relationship between sides 1 & 3
$R_{L1L4} := L1 / L4.$	Relationship between sides 1 & 4
$R_{L2L3} := L2 / L3.$	Relationship between sides 2 & 3
$R_{L2L4} := L2 / L4.$	Relationship between sides 2 & 4
$R_{L3L4} := L3 / L4.$	Relationship between sides 3 & 4

The value of these derived fields will be a real number that indicates the relationship between the two specified sides, in terms of length. The general rule is:

Value	Relationship
1.0	The two sides are of the same length.
< 1.0	The first side is smaller than the second, fractionally specified.
> 1.0	The first side is larger than the second, as a multiplier.

Output

Read 190 cases (10 attributes) from Polygon.data

Decision tree:

```

R_L1L3 <= 0.8:
: ...R_L1L3 > 0.5: Other triangle (9.8/1.2)
:   R_L1L3 <= 0.5:
:     ...R_L2L4 <= 0.5:
:       ...R_L3L4 > 1.111: Other triangle (2.3/0.2)
:       :   R_L3L4 <= 1.111:
:         :     ...R_L3L4 <= 0.9: Other triangle (4.2/1.4)
:         :       R_L3L4 > 0.9: Isosceles Triangle (16.1/0.5)
:     R_L2L4 > 0.5:
:       ...R_L2L4 > 1:
:         ...R_L2L4 <= 1.5: Other triangle (6.1/0.6)
:         :   R_L2L4 > 1.5:
:           :     ...R_L2L3 <= 1.111: Isosceles Triangle (6.2/0.7)
:           :       R_L2L3 > 1.111: Other triangle (3/0.7)
:         R_L2L4 <= 1:
:           ...R_L2L4 <= 0.8: Other triangle (6.1/0.6)
:           :   R_L2L4 > 0.8:

```



```

:          :...R_L3L4 > 0.5: Equilateral triangle (10.3/1.3)
:          R_L3L4 <= 0.5:
:          :...R_L1L2 <= 0.5: Isosceles Triangle (5.1)
:          R_L1L2 > 0.5: Equilateral triangle (3.2/0.3)
R_L1L3 > 0.8:
:...R_L1L3 > 1:
:    :...R_L1L3 > 2.143: Isosceles Triangle (7.4/0.8)
:    R_L1L3 <= 2.143:
:    :...L2 > 991: Isosceles Triangle (2.3)
:    L2 <= 991:
:    :...L3 <= 1: Isosceles Triangle (4.1/2)
:    L3 > 1: Other triangle (12)
R_L1L3 <= 1:
:...L3 <= 0:
:    :...R_L1L4 <= 1.111:
:    :    :...R_L1L4 <= 0.9: Other triangle (4.8/1.9)
:    :    R_L1L4 > 0.9: Equilateral triangle (5.8/1.5)
:    R_L1L4 > 1.111:
:    :...R_L1L4 <= 2: Other triangle (3.9/1.5)
:    R_L1L4 > 2: Isosceles Triangle (2.4/0.5)
L3 > 0:
:...R_L2L4 <= 0.5:
:    :...R_L3L4 <= 1.5: Equilateral triangle (10.5/0.3)
:    R_L3L4 > 1.5: Isosceles Triangle (5.9/0.7)
R_L2L4 > 0.5:
:...L4 <= 0: Equilateral triangle (8.6/1.6)
:    L4 > 0:
:    :...R_L3L4 > 1: Rectangle (13)
:    R_L3L4 <= 1:
:    :...R_L3L4 <= 0.9: Rectangle (12)
:    R_L3L4 > 0.9: Square (25)

```

Evaluation on training data (190 cases):

Decision Tree

Size	Errors	
25	9 (4.7%)	<<

(a)	(b)	(c)	(d)	(e)	<-classified as
-----	-----	-----	-----	-----	
36					(a): class Equilateral triangle
5	51				(b): class Isosceles Triangle
	4	44			(c): class Other triangle
			25		(d): class Square
				25	(e): class Rectangle

Analysis

A predictive accuracy of 95.3% was achieved using a total of 25 levels in the decision tree. This tree is smaller and more accurate than the last one created using the same application, which lacked the derived relationship variables.

Scalability

Also unlike the first experiment, this decision tree is more scalable because it considers relationships rather than constant values. If the program were capable of determining and working with these relationships automatically, then it would have achieved this improved accuracy during the first experiment.

The result of running the 100000 randomly generated legal polygons is 78.5% correct, and 21.5% misclassified.

Miscellaneous datasets

For the purpose of comparing RAGA to other algorithms, several other publicly available datasets were tested. The dataset descriptions and results from various algorithms are listed below.

Classification of olive oils

This data consists of the percentage composition of 8 fatty acids (palmitic, palmitoleic, stearic, oleic, linoleic, eicosanoic, linolenic, eicosenoic) found in the lipid fraction of 572 Italian olive oils. An analysis of this data is given in [Forina, Armanino, Lanteri, Tiscornia (1983)].

For this dataset, the training set is also used for testing.

Results using R

Number of terminal nodes: 6

Residual mean deviance: $0.1035 = 15.01 / 145$

Misclassification error rate: $0.01987 = 3 / 151$

Results using RAGA

Number of rules: 8

Coverage: 100.0%

Misclassification error rate: 0.0%

Letter image recognition data

This dataset was taken from the StatLog project [Brazdil, Gama (1991-1994)].

The objective is to identify each of a large number of black-and-white rectangular pixel displays as one of the 26 capital letters in the English alphabet. The character images were based on 20 different fonts and each letter within these 20 fonts was randomly distorted to produce a file of 20,000 unique stimuli. Each stimulus was converted into 16 primitive numerical attributes (statistical moments and edge counts), which were then scaled to fit into a range of integer values from 0 through 15.

The training set consists of 15000 elements, and the discovered rule sets are used to test the remaining 5000.

Results in C4.5 (as reported by the StatLog project)

Error rate in training: 4.2%

Error rate in testing: 13.2%

Results in RAGA

Training set Coverage: 100.0%

 Misclassification: 2.9%

Testing set Coverage: 100.0%

 Misclassification: 12.72%

Notes

In this particular experiment, RAGA was only allowed to use 1-place predicates in order to compete on the same level as C4.5. Although RAGA had results comparable, it is expected that they will improve once the 1-place predicate restriction is removed.

Previous work

In previous work [Catral, Oppacher, Deugo (1999)] there were several supervised and unsupervised experiments performed on two datasets. The first set, referred to as the *Mushroom* dataset [Lincoff (1981)], is well known and often used to test the accuracy and efficiency of classification algorithms. The second set, referred to as the *Vehicle Silhouette* dataset [Siebert 1987], was tested against several algorithms in the StatLog project [Brazdil, Gama (1991-1994)].

Mushroom dataset

This set contains 8124 sample descriptions of 23 species of gilled mushrooms in the Agaricus and Lepiota Family. The dataset uses 22 attributes, and classifies each mushroom as either edible (51.8%) or poisonous.

The rules generated by RAGA covered the entire dataset with 100% accuracy, as compared to algorithms STAGGER [Schlimmer, 1987] and HILLARY [Iba, Wogulis, Langley, 1988], which approach 95% classification accuracy after training on 1000 instances.

Vehicle Silhouette dataset

This set contains 846 records that use 18 continuous attributes to classify vehicles by their silhouette, as recorded by roadway cameras at different angles. The four classes are OPEL (25.06%), SAAB (25.65%), BUS (25.77%), VAN (23.52%).

StatLog reports the results for C4.5 as having error rates of 6.5% in training, and 26.6% in testing. In several experiments with RAGA, the training set achieved an accuracy of between 95% and 100%, however the testing accuracy ranged only between 62% and 71%.

For this dataset, the rules generated by RAGA each cover surprisingly few instances, and as a result there were 60-90 rules required for 100% coverage. This difference in the number of rules and predictive accuracy may be attributed to some form of over-fitting, however this analysis was not completed.

As with the Letter Image Recognition dataset, RAGA was allowed only 1-place predicates for this experiment, and the results are expected to change once this restriction is removed.

Undirected data mining experiments

In order to test the undirected data mining process, several experiments were performed using a variety of synthetic datasets. This process is described in the corresponding section on page 104.

Preliminary results indicated that experimentation with undirected data mining was not as straightforward as it was for simple classification. In most experiments there were several rules being sought, however only one was usually found by the conclusion of the search. Further analysis of this revealed the source of this problem, as well as a potential solution.

The reason for the single-rule solution is that the dominant rule is preferred at an early point in the process. This occurs because it has a high fitness value with respect to the others being considered. Unlike classification tasks, rules are not rewarded for uniquely addressing data elements, and therefore a very similar rule will often have a very similar fitness. As the generation evolves, variations of the good rule are created in an attempt to improve it, however the original copies and unwanted variations are not automatically discarded. Because these rules are similar in content the fitness is similar, and as with the primary rule the fitness is higher than other rules being considered. The result is a single rule that essentially dominates the entire population with variations of itself, and does not allow for the exploration of other niches.

To control this problem there were several changes made within RAGA to penalize rules based on similarity. Preliminary testing with these options yielded better rule sets with more coverage, however the results of these tests are not fully documented and will be described in future work.

Future work

The primary short-term focus for RAGA will be experimenting with an improving the ability to perform undirected data mining tasks. This involves the design and implementation of functions to determine similarity between elements, or the recognition of underlying structures within the data.

For classification tasks, a post-search processing section may be added to make hierarchies even more comprehensible. One of the techniques being considered is the joining or rules with disjunctions in order to achieve a higher coverage.

Example

If ($A > 5$) Then ($C > 10$) [support 25%]

If ($B < 10$) Then ($C > 10$) [support 20%]

If ($A > 5$) $\vee$ ($B < 10$) Then ($C > 10$) [support might reach ~45%]

Several user interface changes will also be implemented to make RAGA easier to use and conduct experiments. These include:

- ❑ Separation of directed and undirected mining tasks
- ❑ Improved testing facility to allow stopping and restarting of experiments
- ❑ Automatic cross validation with user-defined segment sizes

Conclusion

This thesis describes the motivation, design, implementation, and testing of RAGA.

RAGA is a system that is capable of performing data mining tasks of varied type and difficulty, and satisfies the goals as set forth at the beginning of the thesis.

The first set of experiments showed that in supervised data mining tasks, RAGA demonstrates the ability to generate a rule hierarchy that performs similarly to the decision tree created by C5.0 for the same dataset. The structure of the hierarchy is similar to that of a decision list, where the rules are used in a specific order, however it is much stronger because higher priority rules actually protect less confident ones.

Considering the short development time for RAGA, when compared to the many years that C5.0 and predecessors have been used, the results produced by RAGA are quite positive. Furthermore, when the same experiments are tried with the 1-place predicate restriction removed, RAGA takes advantage of relationships that are discovered between attributes. This consequently generates scalable rule hierarchies that are unattainable by many other algorithms. Although this advantage may be dependent on the problem domain, the success of RAGA does not depend on this feature alone.

In unsupervised data mining tasks the results are not compared to those of other algorithms, however the preliminary experiments generated rules that semantically match or closely approximate the original defining rules of synthetic datasets. Because this is an

area for which few algorithms exist, these results are quite promising and it is expected that future work in this area will have similarly positive results.

Appendix A: File formats

RAGA uses two separate file formats to load and store either unprocessed data, or rule hierarchies.

Raw (unprocessed) data example

; Test dataset

;

Length=5 Width=8 Coloured Class=1

Length=8 Width=5 Class=2

As seen in this example, lines that begin with the semicolon character are ignored by RAGA, and are useful for recording dataset comments or omitting specific cases. Each non-commented line represents one element. For Integer, Real, and Percentage types, the attribute name is followed by the equal sign (=), which indicates the assignment of the following value. For Boolean types there is no assignment operator because their presence indicates a value of true.

Final output example

[C100.00%,S6.32%,R=12,W=0,F67.38%,A100.00%]: If (A > B) then (Class = 3)

[C100.00%,S2.63%,R=5,W=0,F26.18%,A20.00%]: If (D > 1248) then (Class = 2)

The example shows that each line represents one rule within the hierarchy, and the sequence follows the intended execution order. Although unimportant for future

application, the original confidence and support values are stored, as well as other statistical information derived when the rule was evolved. There is no token used to indicate that a particular line or rule is a comment.

Appendix B: Polygon dataset

Number of attributes: 5 (4 predictive, 1 class)

Number of classes: 5

Legend

L_x = Length of side x , where $x \in \{1, 2, 3, 4\}$

Class = Type {equilateral triangle, isosceles triangle, other triangle, square, rectangle}

Attributes

4 Predictive attributes: $L1$, $L2$, $L3$, $L4$

1 Class attribute: $Type$

Number of records

Class / Description	Count
Equilateral triangles	36
Isosceles triangles	56
Other triangles	48
Squares	25
Rectangles	25
Total	190

Appendix C: Multi-target search example

The list of steps described in this section detail an example of one multi-target search, where the confidence and support factors are varied between runs in an attempt to better explore the data.

In between each of the steps, the most recently evolved hierarchy is first saved, and then cleared. The ultimate goal at the conclusion of these preliminary searches is to combine the hierarchies and refine the entire set of rules.

Although the values chosen for confidence and support are arbitrary, they are relatively close to the suggested starting point of 85% confidence, and 10% support [2].

1) Search the data using a confidence of 80%, and support of 20%.

The target values of 80% and 20% were chosen at random, however they act as a starting point for subsequent searches.

2) Perform step #1 using confidence 80%, support 10%.

The reason for the decreased support is that rules with less popularity will be preferred. Depending on the dataset the rules may not be different than what was evolved during the first step. In some cases however, the results may contain variations including new terms and rules.

3) Confidence 85%, support 20%.

Once again, similar to the first rule, but the confidence is stricter.

4) Confidence 85%, support 10%.

This is a combination of the differences between steps #2 and steps #3. Notice that the results from changing both targets at once does not necessarily have the same effect as changing each separately. This is especially true when the confidence and support factors have different weight with respect to each other.

5) Confidence 85%, support 5%.

Searching for rules with very low support can help to find obscure and possibly interesting associations. Although they may not be useful in isolation, they may still be of advantage towards the bottom levels of a larger hierarchy.

6) Confidence 95%, support 10%.

Searching for rules with a very high confidence may also yield rules that are not useful on their own, however can still positively affect the entire set when combined.

After the six preliminary searches above there will be the same number of saved rule hierarchies. Once a new mining session is started, these individual sets must be loaded in one at a time. The duplicate rules will be removed, and the starting population will be seeded with all of the knowledge obtained in the initial six steps.

In order to force diversity into the system, this injection of predefined rules should not completely dominate the population, but rather complement a group of randomly initialized rules. When the search begins these rules will have a significant advantage, however they will combine and change in ways that can create a better hierarchy overall. This process is referred to as *rule refinement* because in most cases the new rules will be specializations or generalizations of the original set.

Appendix D: Traversing a rule hierarchy

After a rule hierarchy has been evolved, a new element is classified following the appropriate path. The following example is a hierarchy that was trained on a set of polygons that are either standing upright, or lying down on their side. The only predictive attributes are two Integer values that denote length and width, which specify the boundaries of the polygon.

At the top of the hierarchy are the two most popular rules. The reason for the existence of the third rule is that some instances cannot be distinguished (ex: length = width), and this is a popular rule that emerged in an attempt to cover these cases.

Example Hierarchy

- 1: If (length > width) Then (class = upright)
- 2: If (width > length) Then (class = lying down)
- 3: If (length > 3)^(width < 5) then (class = upright)

Example elements to be classified

- 1) Length = 10, Width = 5;

This element matches rule #1, and is classified as upright.

- 2) Length = 10, Width = 20;

This element tries to match against rule #1, but fails. It next tries to match against rule #2, which succeeds and the element is classified as lying down.

3) Length = 4, Width = 4;

This element tries to match against the first two rules, but fails. It successfully matches against rule #3, and is classified as upright.

4) Length = 5, Width = 5;

This element does not match any of the rules in the hierarchy, and therefore cannot be classified. If a default class were being used, then it would be classified as the most popular configuration.

References

- [Berry, Linoff, 1997] Michael J.A. Berry, Gordon Linoff, 1997. Data Mining Techniques: For Marketing, Sales, and Customer Support. Wiley computer publishing.
- [Steinberg, Colla, 1995] D. Steinberg, P.L. Colla, 1995. CART: Tree-Structured Nonparametric Data Analysis. Salford Systems, San Diego, CA.
- [Quinlan, 1993] R.J. Quinlan, 1993. C4.5 is a licensed product that can be acquired from Morgan Kaufmann Publishers, Ann Arbor, Michigan.
- [Becker, Chambers, Wilks, 1998] Rick Becker, John Chambers. Allan Wilks. R: A Programming Environment for Data Analysis and Graphics.
- [Holland, 1975] John H. Holland, 1975. Adaptation in Natural and Artificial Systems. University of Michigan Press (Ann Arbor).
- [Goldberg, 1989] David E. Goldberg, 1989. Genetic Algorithms in Search, Optimization & Machine Learning. Reading, Massachusetts: Addison-Wesley Publishing Company, inc.
- [Montana, 1995] David J. Montana, 1995. Strongly typed genetic programming. Evolutionary computation 3(2).

[Levenick, 1991] J.R. Levenick, 1991. Inserting Introns Improves Genetic Algorithm Success Rate: Taking a Cue from Biology. *Proceedings of the Fourth International Conference on Genetic Algorithms*, University of California, San Diego. Morgan Kaufmann Publishers.

[Lehmer, 1951] D.H. Lehmer, 1951. Mathematical methods in large-scale computing units. In *Proc. 2nd Sympos. on Large-Scale Digital Calculating Machinery*, Cambridge, MA, 1949, pages 141-146. Cambridge, MA, 1951. Harvard University Press.

[L'Ecuyer, Blouin, and Couture, 1993] P. L'Ecuyer, F. Blouin. and R. Couture. 1993. A Search for Good Multiple Recursive Generators. *ACM Transactions on Modeling and Computer Simulation*, 3:87-98.

[Forina, Armanino, Lanteri, Tiscornia, 1983] M. Forina, C. Armanino, S. Lanteri. E. Tiscornia. 1983. Classification of olive oils from their fatty acid composition. In Martens, H. and Russwurm Jr., H. (ed), *Food Research and Data Analysis*, Applied Science Publishers, London, 189-214.

[Brazdil, Gama, 1991-1994] P. Brazdil, J. Gama, 1991-1994. Esprit Project 5170 StatLog. LIACC, University of Porto Rua Campo Alegre 823 4150 Porto, Portugal.

[Koza, 1992] John R. Koza, 1992. *Genetic Programming: On the programming of computers by means of natural selection*. MIT Press, Cambridge, Mass.

[Schlimmer, 1987] J.S. Schlimmer, 1987. Concept Acquisition through Representational Adjustment (Technical Report 87-19). Doctoral dissertation, Department of Information and Computer Science, University of California, Irvine.

[Iba, Wogulis, Langley, 1988] W. Iba, J. Wogulis, P. Langley, 1988. Trading off Simplicity and Coverage in Incremental Concept Learning. Proceedings of the 5th International conference on Machine Learning, 73-39. Morgan Kaufmann, Ann Arbor, Michigan.

[Catral, Oppacher, Deugo, 1999] R. Catral, F. Oppacher, D. Deugo, 1999. Using Genetic Algorithms to Evolve a Rule Hierarchy. Proceedings of the 3rd European Conference on Principles and Practice of Knowledge Discovery in Databases.

[Flockhart, Radcliffe, 1996] I. W. Flockhart, N. J. Radcliffe, 1996. A Genetic Algorithm-Based Approach to Data Mining. Proceedings of the Second International Conference on Knowledge Discovery and Data Mining.

[Siebert, 1987] J. P. Siebert, 1987. Turing Institute Research Memorandum TIRM-87-018 "Vehicle Recognition Using Rule Based Methods". Turing Institute, Glasgow, Scotland.

[Lincoff, 1981] G. H. Lincoff, 1981. The Audubon Society Field Guide to North American Mushrooms. Alfred A. Knopf, New York.